Characterizing Crowds to Better Optimize Worker Recommendation in Crowdsourced Testing

Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Fellow, IEEE,
Qiang Cui, Miao Xie, Qing Wang, Member, IEEE

Abstract—Crowdsourced testing is an emerging trend, in which test tasks are entrusted to the online crowd workers. Typically, a crowdsourced test task aims to detect as many bugs as possible within a limited budget. However not all crowd workers are equally skilled at finding bugs; Inappropriate workers may miss bugs, or report duplicate bugs, while hiring them requires nontrivial budget. Therefore, it is of great value to recommend a set of appropriate crowd workers for a test task so that more software bugs can be detected with fewer workers.

This paper first presents a new characterization of crowd workers and characterizes them with testing context, capability, and domain knowledge. Based on the characterization, we then propose Multi-Objective Crowd wOrker recoMmendation approach (MOCOM), which aims at recommending a minimum number of crowd workers who could detect the maximum number of bugs for a crowdsourced testing task. Specifically, MOCOM recommends crowd workers by maximizing the bug detection probability of workers, the relevance with the test task, the diversity of workers, and minimizing the test cost. We experimentally evaluate MOCOM on 532 test tasks, and results show that MOCOM significantly outperforms five commonly-used and state-of-the-art baselines. Furthermore, MOCOM can reduce duplicate reports and recommend workers with high relevance and larger bug detection probability; because of this it can find more bugs with fewer workers.

Index Terms—Crowdsourced testing, Crowd worker recommendation, Multi-objective optimization

1 Introduction

Testing is expensive. According to Brooks [1], testing a software system consumes half of the resources of any software projects. Much work in software engineering has focused on the reduction of this significant cost. New software development methodologies have been developed to encourage more and sooner testing (see all the work on agile and continuous deployment methods). In other work, researchers have proposed impressive, but complex formal methods to allow for the better expression of requirements, then the automatic generation of test cases. Such tools, while success in their home domain, require very skilled and very scarce human operators.

An alternative approach, explored by a growing number of researchers, is crowdsouced testing [2]–[4]. It uses the "crowd", which is a large labor source, to conduct many small tasks and do so relatively inexpensively. The promise is that any large testing task can be completed in the required time, just by hiring more members of the crowd.

The problem with crowdsouced testing is optimizing crowd workers' participation [2], [5]–[7]. Crowd resources, while cheap, are not free. Hence, when scaling up crowdsouced testing, it is necessary to maximize the information gain from every member of the crowds. Also, as shown below not all crowd workers are equally skilled at finding bugs. Inappropriate workers may miss bugs, or report duplicated bugs, while hiring them requires nontrivial budgets. Furthermore, because of the *unknownness*, *largeness* and *undefinedness* of the crowd workers [4], we should not involve all the workers in a crowdsourced testing task. Therefore, it is of great value to recommend a set of appropriate crowd workers for a test task so that more software bugs can be detected with fewer workers.

Finding appropriate workers for particular software engineering tasks has long been recognized as being important and invaluable. There are many lines of related studies about worker recommendation, such as bug triage [8]–[17], mentor recommendation [18], and expert recommendation [19]. With the emergence of crowdsourcing, there are several researches focusing on developers recommendation for crowdsourced software development [20]–[23]. The aforementioned studies either recommend one worker or assume the recommended set of workers are independent with each other. However, in crowdsourced testing, a set of workers need to be recommended to accomplish a test task together. Furthermore, the recommended set of workers are dependent on each other because their performance can together influence the final test outcomes.

- *J. Wang, Q. Wang are with Laboratory for Internet Software Technologies, State Key Laboratory of Computer Sciences, Institute of Software Chinese Academy of Sciences, and University of Chinese Academy of Sciences, Beijing, China.*
  Q. Wang is the corresponding author.
  E-mail: {wangjunjie, wq}@itechs.iscas.ac.cn
- *S. Wang is with Electrical and Computer Engineering, University of Waterloo, Canada.*
  E-mail: song.wang@uwaterloo.ca
- *J. Chen and T. Menzies are with Department of Computer Science, North Carolina State University, Raleigh, NC, USA.*
  E-mail: jchen37@ncsu.edu, timm@ieee.org
- *Q. Cui is with Bytedance Inc., Beijing, China.*
  E-mail: cuiqiang1225@gmail.com
- *M. Xie is with Huawei Technologies Co Ltd, Beijing, China.*
  E-mail: 0520shui@163.com

Manuscript received xxx xx, 2018; revised xxx xx, 2018.

Our previous work has proposed three different approaches to recommend a set of crowd workers for crowdsourced testing tasks [5]–[7]. However, each of them has only partially explored the characteristics of crowd workers and the influential factors of bug detection for crowdsouced testing (details are in Section 3.2). As shown in our experiment (Section 6.1), full exploration of crowd worker's characteristics and influential factors will lead to better results ever seen in prior work.

To improve the practice of crowd worker recommendation, we first present a new characterization of crowd workers which can support more effective crowd worker recommendation. In detail, we characterize the crowd workers with three dimensions, i.e., testing context, capability, and domain knowledge. **Testing context** contains the device model of a crowd worker, the operating system and ROM of her/his device, as well as the network environment. **Capability** represents the ability of a crowd worker abstracted from her/his historical testing behaviors. It contains such attributes as the number of reports submitted, the number and percentage of bugs detected. **Domain knowledge** represents the domain experience a crowd worker obtained through performing past testing tasks. We use the descriptive terms extracted from her/his historical submitted reports to represent her/his domain knowledge.

Based on the characterization of crowd workers, we then propose **M**ulti-**O**bjective **C**rowd w**O**rker reco**M**mendation approach (MOCOM), which aims at recommending a minimum set of crowd workers who could help detect the maximum number of bugs for a crowdsourced testing task. Specifically, MOCOM recommends crowd workers by maximizing the bug detection probability of the selected workers, the relevance with a specific test task, and the diversity of workers, and minimizing the test cost. Among the four objectives, we build a machine learning model to learn the **bug detection probability** for each worker. The features utilized in the learner are the capability of crowd workers, in which we also consider the time-related factors to better model the bug detection probability. For the **relevance** with a specific test task, we compute the cosine distance between workers' domain knowledge and the test task's requirements. The **diversity** of crowd workers is measured based on the differences of their testing context and domain knowledge. **Test cost** mainly consists of the reward for crowd workers and is measured based on the number of recommended workers, which is a common practice in real-world crowd testing platforms [3].

Search-Based Software Engineering (SBSE) is one of the most-commonly used techniques for solving the multi-objective optimization problems in software engineering [24], [25]. Hence, we leverage a widely-used search-based algorithm, namely NSGA-II, to optimize the four objectives when recommending crowd workers.

This paper experimentally evaluates MOCOM on 562 test tasks (involving 2,405 crowd workers and 78,738 test reports) from one of the largest Chinese crowdsourced testing platforms. The experimental results show that MOCOM can detect more bugs with fewer crowd workers, in which a median of 24 recommended crowd workers can detect 75% of all the potential bugs. All objectives are necessary for worker recommendation because removing any of the objectives would result in significant performance decline. In addition, our approach also significantly outperforms five commonly-used and state-of-the-art baseline approaches, with 19% to 80% improvement at *BDR@20* (BDR@20 denotes the percentage of bugs detected by the top 20 recommended workers out of all bugs detected in the task).

This paper makes the following contributions:

- We characterize the crowd workers with three dimensions, i.e., testing context, capability, and domain knowledge, which can capture the characteristics of crowd workers and better support the worker recommendation in crowdsourced testing.
- We propose Multi-Objective Crowd wOrker recoMmendation approach (MOCOM), which recommends crowd workers by maximizing the bug detection probability of workers, the relevance with test tasks, and the diversity of workers, and minimizing the test cost.
- We design a machine learning model to learn the bug detection probability of crowd workers on a given test task, which serves as one objective of MOCOM and improves the worker recommendation performance.
- We evaluate our approach on 562 test tasks (involving 2,405 crowd workers and 78,738 test reports) from one of the largest Chinese crowdsourced testing platforms, and the results are promising.

2 Background and Motivation

2.1 Background

This section presents a brief background of crowdsourced testing to help better understand the challenges we meet in real industrial crowdsourced testing practice. Figure 1 presents the overall procedure of crowdsourced testing. The project manager provides a test task for crowdsourced testing, including the software under test and test requirements. The crowdsourced testing task is usually in the format of *open call* and *incentives provision* [4], so a large number of crowd workers can sign in to perform the task based on its test requirements, and are required to submit crowdsourced test reports. The project manager then inspects these submitted test reports, confirm whether it is a bug, debug and fix it. Note that not every test report involves a bug, and different reports might describe the same bug (i.e., duplicate reports).

In order to attract workers, crowdsourced testing tasks are often financially compensated. The commonly-used payout schema includes *paid by participation*, *paid by bug*, and *paid by first bug* [2], [3], [26]. This work is based on the *paid by participation* schema, in which workers who participate in the test task are equally paid. It is a commonly-used payout schema especially for the newly-launched platform since it can encourage crowd worker's participation [2]. In Section 7.3, we will discuss the usefulness of our proposed approach in terms of other payout schema.

Currently, in most crowdsourced testing platforms, before participating in crowdsourced testing, workers need to search proper test tasks to perform from a large number of published test tasks [2], [3]. This mode is ineffective for bug detection because of the following two reasons: first, workers may choose test tasks they are not good at, which

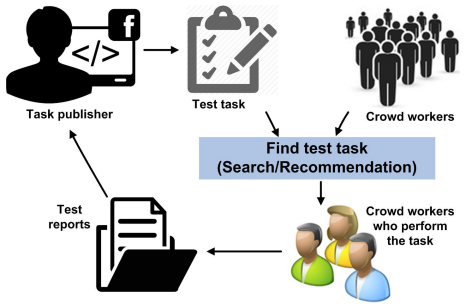


Fig. 1: The procedure of crowdsourced testing

TABLE 1: An example of crowdsourced test task, test report, and crowd worker

Test Task	
Task ID	T000012
Name	IQIYI Testing
Requirement 1	Browse the videos through list mode IQIYI, rank the videos using different conditions, check whether the rank is reasonable.
Requirement 2	Cache the video, check whether the caching list is right.
Test Report	
Report ID	R1002948308
Task ID	T000012
Worker ID	W5124983210
Description	I list the videos according to the popularity. It should be ranked according to the number of views. However, there were many confused rankings, for example, the video "Shibuya century legend" with 130 million views was ranked in front of the video "I went to school" with 230 million views.
Bug label	bug
Duplicate label	R1002948315, R1002948324
Crowd worker	
Worker Id	W5124983210
Context	Phone type: *Samsung SN9009* Operating system: *Android 4.4.2* ROM information: *KOT49H.N9009* Network environment: *WIFI*
Historical Reports	R1002948308, R1037948352

cannot guarantee the quality of testing; secondly, a test task may be conducted by many workers with similar experience or expertise which would result in many duplicated bugs and waste of resources.

In this paper, we suggest a recommendation mode to bridge the gap between crowd workers and test tasks. Our aim is to recommend a set of appropriate workers for a test task. The task publisher can invite these workers on purpose, or attract them with more rewards. In this way, more bugs can be detected with fewer crowd workers and less cost.

2.2 Important Concepts

We introduce three important concepts in crowdsourced testing: *Test task*, *Test report*, and *Crowd worker*.

Review 4-d A *test task* is the input to a crowdtesting platform provided by a task publisher. It contains task ID, task name, test requirements (mostly written in natural language), and the software under test (not considered in this work). Table 1 shows an example of a test task.

A *test report* is the test outcomes submitted by a crowd worker after the test task is completed. It contains report ID, worker ID (i.e., who submit the report), task ID (i.e., which task is conducted), description of how the test was performed and what happened during the test, bug label, and duplicate label. Table 1 shows an example of a test report. Specifically, the labels are assigned by the project manager to indicate whether the report contains a "bug"[1] (i.e., bug label), and with which the report is "duplicate" (i.e., duplicate label). Note that, in the following paper, we refer to "bug report" (also short for "bug") as the report contains bugs, while refer to "test report" (also short for "report") as any report submitted in the test task (including bug reports and reports without bugs).

A *crowd worker* is a registered worker in the crowdsourced testing platform, and is described by worker ID, her/his context attributes (e.g., device model). The platform also records the worker's historical test reports. A test task can be conducted by hundreds of crowd workers.

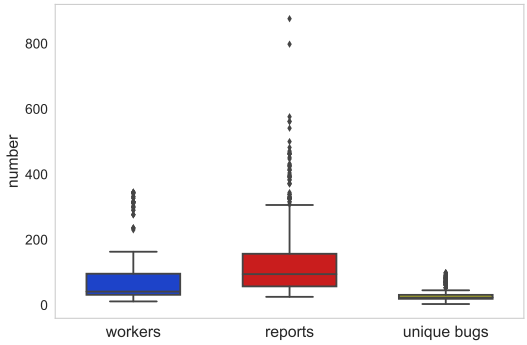


Fig. 2: Statistics of Baidu CrowdTest dataset **Review 4-e**

2.3 Baidu CrowdTest Dataset

We collected crowdsourced testing data from an industrial crowdsourced testing platform, namely Baidu CrowdTest[2]. Baidu CrowdTest was founded in 2011 and has become one of the largest crowdsourced testing platforms in China.

We collected the test tasks that are closed between Jan. 1st 2015 and Aug. 1st 2016. When a crowd worker registers in the platform, he/she is asked to sign a Confidentiality Agreement. It means all the submitted and generated data can be applied to scientific research. Besides, when we collect data from this platform, all information related with the personal privacy (e.g., name, age, phone number, occupation) are encrypted.

In total, we analyzed 562 test tasks, involving 2405 crowd workers and 78738 test reports. For each testing task, we collected its task-related information and all the submitted test reports (see examples in Table 1). We also collected all the involved crowd workers and related information (see examples in Table 1).

Figure 2 shows the statistics for workers participated, reports submitted, and unique bugs (i.e., non-duplicate bug reports) for each test task.

To understand the real-world crowdsourced testing practice, we have conducted an analysis on the collected dataset, and observations are shown in the next subsection.

1. In our experimental platform, each test report can contain zero or one bug.

2. Baidu (baidu.com) is the Chinese largest search service provider. Its crowdsourced testing platform is test.baidu.com.

2.4 Observations and Implications

Based on our collected dataset, we have made the following observations:

1) Although a large number of crowd workers can participate in a test task, not every worker could successfully detect bugs in the task. Figure 3a shows the number of crowd workers participated in each test task, and the number of crowd workers detected bugs in the task. Review 4-f The percentage of workers who have detected bugs among all the involved workers is only 52.6%.

Motivated by this observation, it would be of great value to recommend an appropriate set of candidate workers to perform a test task in order to detect more bugs with fewer workers and less cost. This is especially important when the number of candidate workers is large, which is typical in the context of crowdsourced testing.

2) Different crowd workers might have different bug detection capability. Figure 3b shows the number of bugs detected by each crowd worker. We can see that most workers only detected very few bugs, while there is a small portion of workers who have detected much more bugs than others.

This observation motivates us to look for capable workers, who demonstrate greater capability in history and would be more likely to detect bugs in future.

3) Crowd workers would be more likely to detect bugs when conducting tasks they are familiar with. Figure 3c shows the relationship between workers' familiarity with the test task and their bug detection performance. Given a test task, we first calculate the familiarity between each worker and the test task (i.e., cosine similarity between worker's historical reports and task's requirements). We then average two similarity values, one for the workers who have detected bugs (*sim_p*), and the other for the workers who have not detected bugs (*sim_n*). Following that, we obtain the difference for the two similarity values for each task, i.e., (*sim_p - sim_n*)/*sim_n*. A positive value denotes the familiarity for the workers who have detected bugs is greater than those who have not detected bugs in the task, while a negative value denotes the opposite phenomenon. From Figure 3c, we can see that most projects demonstrate positive values, indicating workers have higher possibility to detect bugs in the test tasks they are familiar with.

This observation motivates us to select workers with expertise relevant to a given test task.

4) For each test task, a number of duplicate reports are reported by different crowd workers, as shown in Figure 3d. An average of 80% reports are duplicates of other reports.

This observation motivates us to decrease the duplicate reports by looking for diverse crowd workers so as to further reduce the waste of cost.

In summary, the above observations motivate the need of recommending a subset of crowd workers for a given test task. They also motivate us to consider workers' capability, their relevance with the test task, as well as the diversity of the selected set of workers.

3 Related Work

In this section, we discuss three areas related to our work, i.e., crowdsouced testing, worker recommendation, test case selection and prioritization.

Review 4-a

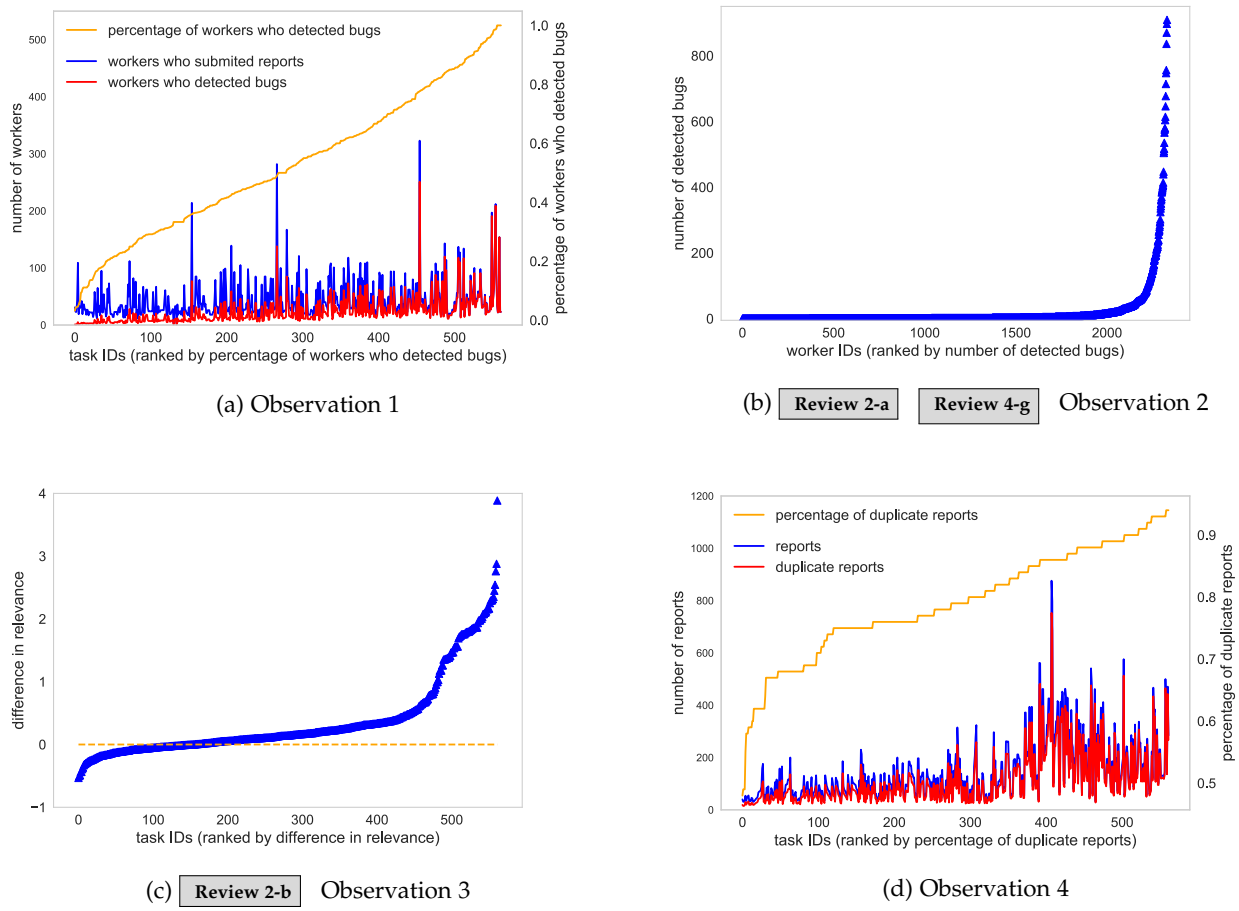
3.1 Crowdsourced Testing

Crowdsourced testing, as a novel practice of software development [2], [27], has been applied to generate test cases [28], measure real-world performance of software products [29], help usability testing [30], detect and reproduce context-related bugs [31]. All the studies above use crowdsourced testing to solve the problems in traditional software testing activities. However, our approach is to solve the new encountered problem in the practice of crowdsourced testing, i.e., worker recommendation for crowdsourced testing task.

Some other studies focus on solving the new encountered problem in the practice of crowdsourced testing. Feng et al. [32], [33] proposed test report prioritization methods for crowdsourced testing. They designed strategies to dynamically select the most risky and diversified test report for inspection in each iteration. Wang et al. [34], [35] proposed to mitigate the distribution difference problem in crowdsourced report classification, through selecting similar data instances from training set to build a classifier for identifying test reports that actually reveal fault. Later, DARS [36] was proposed to leverage deep learning techniques to overcome the data distribution difference across domains in crowdsourced reports classification.

Our previous work has proposed three different approaches to recommend a set of crowd workers for crowdsourced testing tasks. Specifically, Xie et al. [7] proposed Cocoon to recommend a set of crowd workers for a test task. It is designed as a pigeon greedy approach that can achieve the expected test context coverage and maximize testing quality under the context constraint. Cui et al. [5] proposed ExReDiv, a hybrid approach for crowd worker selection, which recommends crowd workers by balancing workers' experience in testing, expertise with the test task, and their diversity in expertise. Meanwhile, Cui et al. [6] proposed MOOSE, which recommends crowd workers by maximizing the coverage of test requirement and the test experience of the selected workers, and minimizing the cost.

Each of the above three approaches has only partially explored the characteristics of crowd workers and the influential factors of bug detection for crowdsouced testing. **First**, Cocoon put more attention on the test context, while concerned less about the workers' characteristics (e.g., capability, past experience) which have found to be critical to the test performance [2], [5], [22], [37]. **Second**, both ExReDiv and MOOSE did not consider the test context of the crowd workers (e.g., the device models) which has proven to be influential to the test outcomes [2], [7], [37]. **Third**, ExReDiv and MOOSE treated the number of detected bugs as a worker's experience; however a worker's experience can also relate with other attributes, such as his submitted reports, number of participated projects, percentage of detected bugs. **Fourth**, none of these three approaches considered the time-related influence (e.g., worker's experience in terms of the past 2 month vs. worker's experience in terms of the past 12 months) on a worker's bug detection performance. However, previous work demonstrated the online developers' recent activities have greater indicative effect on their future behaviors than the activities happened long



(a) Observation 1

(b) Review 2-a Review 4-g Observation 2

(c) Review 2-b Observation 3

(d) Observation 4

Fig. 3: Observations based on Baidu CrowdTest dataset

before [38], [39]. Therefore, the consideration of these time-related factors can potentially improve the worker recommendation performance.

To summarize, this work differs existing studies as follows:

- This work proposes a new characterization of crowd workers which considers worker's test context, capability, and domain knowledge, while previous researches only considered one or two of them.
- This work proposes a new multi-objective worker recommendation approach based on the new characterization of crowd workers, which can lead to better results ever seen in prior work.
- This work models crowd worker's experience based on multiple features, and consider the time-related influence on its experience, which is proven to be effective.

3.2 Worker Recommendation

Software development has become a more and more open activity, where stakeholders can usually come from undefined public. Finding appropriate workers for a particular software engineering task is becoming important and invaluable. There are many lines of related studies for recommending workers for various software engineering tasks, such as bug triage [8]–[17], mentor recommendation [18], and expert recommendation [19].

With the emergence of crowdsourcing, there are several researches focusing on developer recommendation for crowdsourced software development. Yang et al. [20] proposed a novel approach, DR_PSF, to enhance developer recommendation by leveraging personalized source code files. Mao et al. [21] employed content-based recommendation techniques to automatically match tasks and developers during crowdsourced software development. Yang et al. [22] proposed an analytics-based decision support methodology to guide workers recommendation of crowdsourced development.

The aforementioned researches either recommend one worker, or assume the recommended set of workers are independent with each other. However, our work recommends a set of workers who are dependent on each other, because their performance can together influence the final test outcomes.

3.3 Test Case Selection and Prioritization

Within software testing area, another body of previous researches focus on the test case selection and prioritization [40]–[51], which also concerns finding more bugs with less testing cost. However, these approaches can

hardly be used in crowd worker recommendation due to the following reasons. Firstly, most existing researches proposed white-box approaches which rely on structural coverage information in source code (e.g., method coverage, statement coverage, or branch coverage) [45]–[51]. However, the crowdsourced testing platform cannot obtain the source code of the tested apps because of business confidentiality, let alone the coverage information.

Secondly, several recent researches proposed black-box selection approaches which only utilize test cases and do not require the tested program [40]–[43]. They employed the linguistic data of test cases (e.g., their identifier names, comments, string literals), and selected the test cases with different linguistic representation. Such kinds of treatments could not capture the overall characteristics of crowd workers, thus can hardly achieve good bug detection performance.

Another type of black-box selection approaches was to learn the bug revealing probability of a test case, and select these test cases which have a higher chance to trigger bugs [44]. Still, the features used in the machine learning learner were extracted from the code of test cases. While in crowdsourced testing, what we need to handle is crowd workers rather than code.

Although the aforementioned approaches could hardly be used to solve our crowd worker recommendation problem, they motivate us in designing our crowd worker recommendation approach, i.e., select workers with large bug detection probability and with different characteristics. Furthermore, we also select two state-of-the-art approaches ([41], [42]) as the baselines to evaluate MOCOM.

4 Multi-Objective Crowd wOrker recoMmendation approach (MOCOM)

Motivated by the findings in Section 2.4, we model the worker recommendation problem as a multi-objective optimization problem. We propose a Multi-Objective Crowd wOrker recoMmendation approach (MOCOM), which can maximize the bug detection probability of workers, the relevance with the test task, the diversity of workers, and minimize the test cost.

In this section, we first present the characteristics of crowd workers (Section 4.1), followed by the measurement of the four objectives (Section 4.2). Then we present the multi-objective optimization framework for crowd worker recommendation (Section 4.3).

4.1 Characterization of Crowd Worker

In this work, we characterize a crowd worker from three dimensions, i.e., testing context, capability, and domain knowledge. The following subsections illustrate the details of these three dimensions.

4.1.1 Testing Context

Review 4-c **Testing context** represents the hardware (e.g., device model), software (e.g., the operating system of the device), and environment (e.g., network environment) owned by a crowd worker. The reason we consider testing context as one characteristic of a crowd worker is that crowd workers often run the testing task under their specific contexts, which can influence the testing outcomes [2], [7], [37].

We use four attributes to characterize the testing context of a crowd worker. They are the crowd worker's *device model* used to run a test task, the *operating system* of the device model, the *ROM type* of the device model, and the *network environment* under which a test task is run. These attributes are the complete set of testing context recorded by the Baidu CrowdTest platform. Hence, we employ them to characterize a crowd worker and consider them in worker recommendation process. Note that, these attributes are shared by other popular crowdtesting platforms [2], [3], [26], because the platforms need these attributes to reproduce the bugs for the tested app. Even for the crowdtesting platforms that do not have all the four attributes, they can use other similar testing contexts for building the bug probability prediction model, and apply the model in their own cases.

4.1.2 Capability

Capability represents the ability of a crowd worker abstracted from her/his historical testing outcomes. A crowd worker's past performance can reflect her/his capability to a great extent, and has great indicative effect on her/his future bug detection performance [2], [5], [22], [37]. Hence, we consider the capability as an essential dimension to characterize a crowd worker.

We use the following attributes to characterize a crowd worker's capability, i.e.,

- *1) Number of projects which the worker participated in*
- *2) Number of **test** reports submitted by the worker*
- *3) Number of **bug** reports submitted by the worker*
- *4)* Review 2-c *Percentage of **bug** reports submitted by the worker*
  It is computed as the number of bug reports submitted by the worker divided by the number of test reports submitted by the worker.
- *5)* Review 2-d *Degree of duplicate bug reports of the worker*
  It is computed as the *duplicate index* of a worker divided by the number of bug reports submitted by the worker. In it,

$$duplicate\ index = \sum_{r} \frac{1}{number\ of\ r's\ duplicates} \quad (1)$$

  where *r* is the bug report submitted by a worker, and *number of r's duplicates* is the number of duplicates of report r in the test task. For example, worker *W5124983210* in Table 1 submitted two bug reports *R1002948308* and *R1037948352*, where *R1002948308* has 2 duplicates, and *R1037948352* has 6 duplicates. Then percentage of duplicate bugs of worker *W5124983210* is (1/2 + 1/6) / 2. The reason why we do not directly use the number of duplicates divided by the number of bug reports is that we want to not only represent the number of duplicates but also distinguish the degree of duplicates (i.e., using duplicate index).

4.1.3 Domain Knowledge

Domain knowledge represents the domain experience a crowd worker obtained through performing testing tasks. Review 2-e The application under test usually

TABLE 2: Characterization of crowd worker

Testing Context	
Phone type	Samsung SN9009
Operating system	Android 4.4.2
ROM information	KOT49H.N9009
Network environment	WIFI
Capability	
Number of projects participated	15
Number of reports submitted	19
Number of bugs detected	7
Percentage of bugs detected	36.8%
Percentage of duplicate bugs	41.5%
Domain Knowledge	
Descriptive terms extracted from historical reports	< video, list, popularity, rank, view, ...>

come from various domains, and they call for the crowd workers with specific domain knowledge to better explore their functionality [5], [22]. We also observed that crowd workers are more likely to detect bugs when conducting tasks they are familiar with (Section 2.4). This is why we regard domain knowledge as another important criterion to characterize a crowd worker.

Review 3-e We use the "descriptive terms" extracted from a crowd worker's historical submitted reports to represent her/his domain knowledge and represent it as a vector. We present a detailed description about how to obtain the descriptive terms as follows.

We first construct a *descriptive terms list* based on all the tasks in the training dataset (see Section 5.2). We conduct word segmentation and remove stopwords (i.e., *on, the*) to reduce noise. Apart from that, we find that some terms may appear in a large number of reports, while some other terms may appear in only very few documents. Both of them are less predictive and contribute less in modeling the crowd workers. Therefore, we rank the terms according to the number of reports in which a term appears (i.e., document frequency, also known as *df*), and then filter 5% terms with the highest document frequency and 5% terms with the lowest document frequency. Note that, we have conducted experiments with the threshold ranging from 2% to 20%, and results show that with a threshold of 5%, the worker recommendation performance (i.e., bug detection rate at k) can achieve a relative high and stable value. Another note, since the test reports are often short, the term frequency (also known as *tf*), which is another commonly-used metric in information retrieval, is not discriminative, so we only use document frequency to rank the terms. In this way, the final *descriptive terms list* V is formed.

Secondly, we extract the words from a crowd worker's historical submitted reports, and by mapping these words with *descriptive terms list*, we obtain the descriptive terms for representing her/his domain knowledge.

In Table 2, we summarize all the attributes mentioned above. We also present an example of each attribute based on the crowd worker *W5124983210* (in Table 1). We will use these attributes to characterize the crowd workers and measure the objectives.

4.2 Measurement of Four Objectives

Since the purpose of crowd worker recommendation is to help find more bugs with fewer crowd workers, the design of MOCOM considers four objectives.

First, we should recommend the crowd workers with **maximized bug detection probability** [44], since they can potentially improve the bug detection performance.

Second, we should look for the crowd workers with **maximized relevant** expertise with the test task because they have more background knowledge and can increase the bug detection likelihood [5], [6]. This is also because the crowdsourcing task can be *complexity* and *user-driven* [4], hence we should consider the workers' relevance with the task so as to improve the bug detection performance.

Review 2-f Third, we should select a set of crowd workers with *diverse* characteristics [40]–[43], because different workers might explore different areas of the application under testing which would help detect more bugs and reduce duplicate reports.

Last but not least, we should consider the **test cost** which is an essential consideration in crowdsourcing field.

The following subsections illustrate the details of these four objectives.

4.2.1 Objective 1: Maximize Bug Detection Probability of Crowd Workers

We build a machine learning model to learn the **bug detection probability** for each worker. The primary focus of building the model is to determine which features can be utilized for learning the bug detection probability.

Motivated by the findings in Section 2.4, we assume a crowd worker's capability is tightly related with the bug detection probability. So we treat all the crowd worker's capability-related attributes as the features in the machine learning model.

Apart from that, previous work demonstrated the open source developer's recent activity has greater indicative effect on his future behavior than the activity happened long before [38], [39]. Therefore, we further take the time-related factors into consideration, and better model the crowd worker's past experience. Take one of the capability attribute *number of reports submitted* as an example, we also extract the *number of reports submitted in the past 2 weeks*, *number of reports submitted in the past 1 months*, and *number of reports submitted in the past 2 months*. The reason why we use these three time intervals is that our dataset shows more than 75% crowd worker's past activities occur in the past 2 months.

In this way, the original one attribute (i.e., *number of reports submitted*) can yield four features in the machine learning model, and the original five capability attributes in Table 2 generate 20 features in our learning model (demonstrated in Table 3).

Furthermore, we employ another time-related feature in our machine learning model. It is the time interval between the crowd worker's last submission on this platform and the test task's publishing time, measured in number of days. Intuitively, the longer this time interval is, the less likely the crowd worker would take part in this task.

In summary, we list all the aforementioned 21 features which are used in our machine learning model in Table 3. We employ Logistic Regression as our machine learning model, which is widely reported as effective in many different classfication tasks in software engineering [36], [52]. **Review 3-b** Based on the logistic regression model

TABLE 3: Feature for machine learning model

Number of projects in the past	Number of projects in past *2 weeks*
Number of projects in past *1 month*	Number of projects in past *2 month*
Number of reports in the past	Number of reports in past *2 weeks*
Number of reports in past *1 month*	Number of reports in past *2 month*
Number of bugs in the past	Number of bugs in past *2 weeks*
Number of bugs in past *1 month*	Number of bugs in past *2 month*
Percentage of bugs in the past	Percentage of bugs in past *2 weeks*
Percentage of bugs in past *1 month*	Percentage of bugs in past *2 month*
Percentage of duplicates in the past	Percentage of dup. in past *2 weeks*
Percentage of dup. in past *1 month*	Percentage of dup. in past *2 month*
Time interval between last submission and task publish time (in days)	

trained on the training dataset, given a task in the testing dataset, we can obtain its bug detection probability of all candidate workers. Specifically, for a set of candidate crowd workers (i.e., one solution in Section 4.3), we add up their bug detection probability on the given test task and consider the summation as the bug detection probability of the test task.

4.2.2 Objective 2: Maximize Relevance of Crowd Workers with Test Task

For relevance objective, we need to measure the relevance between the candidate crowd workers and a test task.

We use the similarity between a worker's domain knowledge and a test task to denote the relevance. It is computed based on the cosine similarity between the descriptive terms of the crowd worker's domain knowledge and the descriptive terms of the test task's requirements. A larger similarity value denotes the worker's domain knowledge is more tightly relevant with the test task. The reason why we use cosine similarity is that past researches have demonstrated its effectiveness in high-dimensional textual data [13], [34], [53], which is exactly our case.

Given a specific test task, to obtain the relevance for a set of candidate crowd workers (i.e., one solution in Section 4.3), we first combine the domain knowledge of all selected workers as a unified vector, then compute the cosine similarity between the unified vector and a test task's requirements.

4.2.3 Objective 3: Maximize Diversity of Crowd Workers

For the diversity objective, we need to measure the diversity of a set of selected crowd workers. **Review 4-h** Objective 2 (i.e., maximize relevance) aims at finding workers who are familiar with a test task. Apart from that, the nature of software testing calls for diverse workers who can help explore various parts of the app and reduce duplicate reports. Hence objective 3 (i.e., maximize diversity) aims at finding workers with diverse background. Note that, although these two objectives seems conflicting with each other, the multi-objective optimization framework utilized in this work can help reach a balance between relevance maximization and diversity maximization.

An important goal for any multi-objective optimizer is *diversity*, i.e. generating solutions that span the space of possibilities, which is widely recognized [24], [54]. Hence, many optimization algorithms have specialized diversity operators built in to their core operation. For example, the NSGA-II optimizer used in this work employs a novel space pruning operator which strives to spread out the answers that it generates [24]. Note that NSGA-II's diversity is *diversity in objective space*.

The alternative to *objective diversity* in the output space (i.e., exploring diverse solutions) is *attribute diversity* in the input space (i.e., finding diverse workers within a solution). The reason why we need attribute diversity is that NSGA-II's objective diversity operator was incomplete for our purposes, because the observation in Section 2.4 motivates us to recommend a diverse set of workers (i.e., attribute diversity). In addition, experiment results (Section 6.3) show that if attribute diversity is disabled, then the performance would decline sharply.

To explore the attribute diversity, we use count-based method to measure it, and count how many different attribute values appeared in the selected set of crowd workers (i.e., one solution in Section 4.3).

Remember that the crowd workers are characterized by three dimensions: testing context, capability, and domain knowledge. For capability dimension, it is unreasonable to consider the diversity because we require all the selected workers are capable, rather than some of them are capable while others are not. Accordingly, we compute the diversity based on other two dimensions, i.e., crowd workers' testing context and domain knowledge.

Review 4-b Specifically, for testing context, we count how many different phone types, operating systems, ROM types, and network environments contained in the set of workers. For domain knowledge, we count how many different terms appeared in the domain knowledge of the workers.

Note that, testing context only has four attributes, while the domain knowledge has thousands of attributes (i.e, the number of unique descriptive terms). In order to eliminate the influence of different number of attributes on the diversity measurement, we compute the diversity respectively for testing context and domain knowledge, then obtain the final diversity values using a weight parameter. We have experimented with different weights, it turns out a weight of 0.5 (i.e., testing context and domain knowledge are equally treated) can obtain relative good and stable performance. So we use this weight in the evaluation.

4.2.4 Objective 4: Minimize Test Cost

The cost is an unavoidable objective when recommending workers for the crowdsourced tasks. The most important cost in crowdsourced testing is the reward for workers. We suppose all the workers who participate in a test task are equally paid, which is a common practice in real-world crowdsourced testing platforms [3]. In this way, the cost for a set of selected workers (i.e., one solution in Section 4.3) is measured as its size.

4.3 Multi-Objective Optimization Framework

We have mentioned in Section 4.2 that MOCOM needs to optimize four objectives. Obviously, it is difficult to get optimal results for all objectives at the same time. For example, to maximize bug detection probability, we might need to hire more crowd works, thus, sacrifice the fourth objective, i.e., minimize test cost. Our proposed MOCOM seeks a *Pareto front* (or set of solutions). Solutions outside Pareto front cannot dominate (better than, under all objectives) any solutions within the front.

MOCOM uses *NSGA-II* algorithm (i.e., Non-dominated Sorting Genetic Algorithm-II) to optimize the aforementioned four objectives. NSGA-II is a widely used multi-objective optimizer in and out of Software Engineering area. According to [54], more than 65% optimization techniques in software analysis are based on Genetic Algorithm (for problems with single objective), or NSGA-II (for problems with multiple objectives). For more details of NSGA-II algorithm, please see [55].

In our crowd worker recommendation scenario, a Pareto front represents the optimal trade-off between the four objectives determined by NSGA-II. The tester can then inspect a Pareto front to find the best compromise between having a crowd worker selection that balances bug detection probability, relevance, diversity, and test cost or alternatively having a crowd worker selection that maximizes one/two/three objective/s penalizing the remain one/s.

MOCOM has the the following four steps:

1) Solution encoding. Like other recommendation problems [6], [24], we encode each worker as a binary variable. If the worker is selected, the value is one; otherwise, the value is zero. The solution is represented as a vector of binary variables, whose length equals to the number of candidate crowd workers. The solution space for the crowd worker recommendation problem is the set of all possible combinations whether each crowd worker is selected or not.

2) Initialization. The starting population is initialized randomly, i.e, randomly selecting K (K is the size of initial population) solutions among all possible solutions (i.e., the solution space). We set K as 200 as recommended by [56].

3) Genetic operators. For the evolution of binary encoding for the solutions, we exploit standard operators as described in [24]. We use single point crossover, bitflip mutation to produce the next generation. We use binary tournament as the selection operator, in which two solutions are randomly chosen and the fitter of the two will survive in the next population.

4) Fitness functions. Since our goal is to optimize the four considered objectives, each candidate solution is evaluated by our objective functions described in Section 4.2. For bug detection probability, relevance, and diversity, the larger these values are, the faster the convergence of a solution is. The test cost benefits from the smaller values.

5 Experiment Design

5.1 Research Questions

Our evaluation addresses the following research questions:

- **RQ1**: How effective is MOCOM in crowd worker recommendation?

For RQ1, we first present some general views of our approach for worker recommendation, measured in bug detection rate. To further demonstrate the advantages of MOCOM, we then compare its performance with 5 commonly-used and state-of-the-art baseline methods (details are in Section 5.4).

- **RQ2**: Review 4-i How effective is the machine learning model in predicting the bug detection probability?

We build a machine learning model to better obtain the bug detection probability, which serves as one objective in MOCOM. RQ2 aims at investigating the effectiveness of our machine learning model on predicting the bug detection probability considering the predicted probability with the actual bug detection results.

- **RQ3**: Review 4-j What is the contribution of each objective to the overall approach?

In this work, four objectives are utilized for facilitating crowd worker recommendation (details are in Section 4.2). Among them, the objective of cost is indispensable, which cannot be removed. RQ3 explores the performance of MOCOM when removing each of the other three objectives in order to investigate the contribution of each objective.

- **RQ4**: Do the results of MOCOM achieve high quality?

RQ4 is to evaluate the quality of Pareto fronts produced by our search-based approach, which can further demonstrate the effectiveness of our approach. We apply three commonly-used quality indicators, i.e., HyperVolume (*HV*), Inverted Generational Distance (*IGD*), and Generalized Spread (*GS*) (see Section 5.3).

5.2 Experimental Setup

Our experiments are conducted on crowdsourced reports from the repositories of Baidu CrowdTest platform (details are in Section 2.3). Review 4-k To simulate the usage of MOCOM in practice, we employ a commonly-used longitudinal data setup [14], [16]. That is, all the 562 experimental test tasks were sorted in the chronological order, and then divided into 20 non-overlapped and equally sized folds with each fold having 28 test tasks (the last fold has 30 tasks).

Review 3-c We then employ the former *N-1* folds as the training dataset to train MOCOM and use the test tasks in the *N*th fold as the testing dataset to evaluate the performance of worker recommendation. We experiment *N* from 11 to 20 to ensure a relative stable performance because a too small training dataset could not reach an effective model. Note that, what varies in the different experiments is the size of training set which goes from 10 to 19 folds, while the testing set always contains one fold of test tasks.

The role of the training dataset is extracting the capability and domain knowledge of the crowd workers based on their historical submitted reports, and building the machine learning model for predicting bug detection probability. For each test task in the testing dataset, we run MOCOM and baseline methods to recommend a set of crowd workers, and evaluate their performance. In total, we have 282 test tasks (i.e., $9 * 28 + 30$) to evaluate MOCOM.

We configured NSGA-II with the setting of *initial population* = 200, *maximum fitness evaluation (i.e., number of runs)* = 20,000 as recommended by [56].

5.3 Evaluation Metrics

Given a test task, we measure the performance of a worker recommendation approach based on whether it can find the "appropriate" workers who can detect bugs. Following previous studies [5]–[7], we use the commonly-used bug detection rate for the evaluation.

Bug Detection Rate at k (*BDR@k*) is the percentage of bugs detected by the recommended k crowd workers in a test task out of all bugs historically detected in the specific

task. Formally, given a set of recommended k workers (i.e., W) and a test task (i.e., T), the *BDR@k* is defined as follows:

$$BDR@k = \frac{\#bugs\ detected\ by\ workers\ in\ W}{\#all\ bugs\ of\ T} \quad (2)$$

Note that, "bugs" here are referred as no duplicate bugs. We inspect the Pareto front produced by our approach (Section 4.3), and find the recommended worker set under different k values. Since a smaller subset is usually preferred in crowd worker recommendation due to the limited budget, we obtain *BDR@k* when k is 3, 5, 10, 20, 50, and 100.

To compare the *BDR@k* values of the different worker recommendation approaches, we additionally use two metrics.

Firstly, we employ $\%\Delta$ which is the percent difference between the *BDR@k* of two approaches.

$$\%\Delta(\mu_1, \mu_2) = \frac{\mu_2 - \mu_1}{\mu_1} \quad (3)$$

where μ_1 and μ_2 are the *BDR@k* values of two worker recommendation approaches.

Secondly, we use the Mann-Whitney U statistical test [52] to determine if the difference between the *BDR@k* is statistically significant. The Mann-Whitney U test has the advantage that the sample populations need not be normally distributed (non-parametric). If the p-value of the test is below 0.05, then the difference is considered statistically significant.

In addition, we apply HyperVolume (*HV*), Inverted Generational Distance (*IGD*), and Generalized Spread (*GS*) to evaluate the quality of Pareto fronts produced by our search-based approach (see Section 4.3), which have been widely used in existing Search-Based Software Engineering studies [24], [54], [57]. These three quality indicators compare the results of the algorithm with the reference Pareto front, which consists of best solution.

HyperVolume (*HV*) is the combination of convergence and diversity indicator. It calculates the volume covered by the non-dominated set of solutions from an algorithm. A higher value of HV demonstrates a better convergence as well as diversity; *higher* values of HV are *better*. **Inverted Generational Distance (*IGD*)** is a performance indicator. It computes the average distance between set of non-dominated solutions from the algorithm and the reference Pareto set. A lower IGD indicates the result is closer to the reference pareto front of a specific problem; i.e. *lower* values of IGD are *better*. **Generalized Spread (*GS*)** is a diversity indicator. It computes the extent of spread for the non-dominated solutions found by the algorithm. A lower value of GS shows that the results have a better distribution; i.e. *lower* values of GS are *better*. Due to the limited space, for details about the three quality indicators, please refer to [24].

5.4 Ground Truth and Baselines

The **Ground Truth** of bug detection performance is obtained based on the historical crowd workers who participated in test tasks. In detail, we first rank the crowd workers based on their submitted reports in chronological order, then obtain the *BDR@k* based on this order. **Review 3-g** For example, *BDR@3* is based on all bugs detected by the first three crowd workers who participated in test tasks. The maximum value of *BDR@k* of ground truth is 1.00 because we suppose all the bugs have been detected when a test task is closed.

To further explore the performance of MOCOM, we compare MOCOM with 5 commonly-used and state-of-the-art baselines.

ExReDiv [5]: This is a weight-based crowd worker recommendation approach. It linearly combines experience strategy (i.e., select experienced workers), relevance strategy (i.e., select workers with expertise relevant to the test task), and diversity strategy (i.e., select diverse workers).

MOOSE [6]: This is a multi-objective crowd worker recommendation, which can maximize the coverage of test requirement, maximize the test experience of the selected crowd workers, and minimize the cost.

Cocoon [7]: This crowd worker recommendation approach is based on maximizing the testing quality under the test context coverage constraint. In it, the testing quality of each worker is measured based on the number of bugs reported in history.

STRING [41]: This approach is designed for black-box test case selection. It uses string distances on the text of test cases for comparing and prioritizing test cases. In detail, this approach first converts each test case into a string of text, and greedily selects one test case which is farthest from the set of already-selected test cases. In our crowd worker selection scenario, we treat a crowd worker as a test case and consider her/his historical submitted reports as the content of test case.

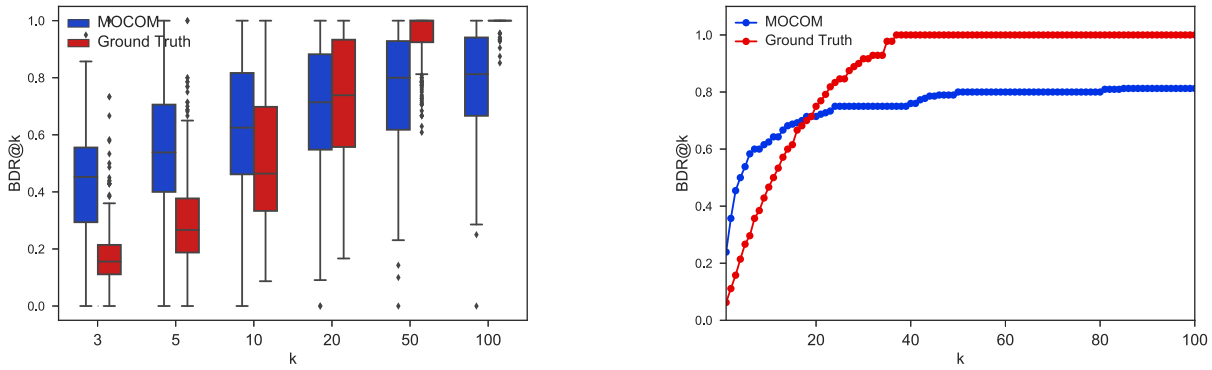
TOPIC [42]: This is another popular black-box test case selection approach. It represents the test cases using the linguistic data of test cases (i.e., their identifier names, comments, and string literals), and applies topic modeling to the linguistic data to model the functionality of each test case. Then it gives high priority to the test cases which test different functionality of the system under test. In our crowd worker selection scenario, we also treat a crowd worker as a test case and consider her/his historical submitted reports as the linguistic content. **Review 3-h** Although *STRING* and *TOPIC* are designed for test case selection and consider different aspects from other three worker recommendation baselines, out of curiosity, we want to investigate whether test case selection approaches can also do the crowd worker recommendation problem.

6 Results and Analysis

6.1 Answering RQ1: Effectiveness of MOCOM

We first present some general views of the performance for our proposed MOCOM for worker recommendation, measured in bug detection rate (*BDR@k*).

Figure 4a demonstrates *BDR@k* under six representative k values. We can see the median *BDR@k* is 0.46 when k is 3, the median *BDR@k* is 0.62 when k is 10, and the median *BDR@k* is 0.70 when k is 20. To put it another way, with 3 recommended workers, the median for the percentage of detected bugs is 46%. In addition, a median of 62% of all



(a) Bug detection rate for projects

(b) Median bug detection rate curve

Fig. 4: Performance of MOCOM (RQ1)

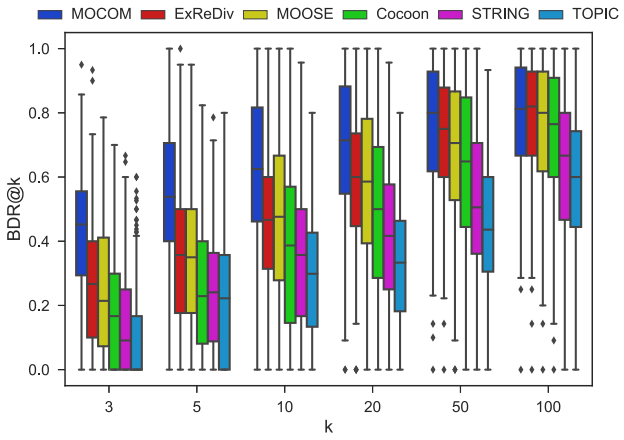


Fig. 5: Performance comparison with baselines (RQ1)

TABLE 4: Results of $\%\Delta$ for median BDR@k of baselines(RQ1)

	k=3	k=5	k=10	k=20	k=50	k=100
MOCOM vs. ExReDiv	69%	50%	33%	19%	6%	0%
MOCOM vs. MOOSE	111%	53%	31%	21%	13%	1%
MOCOM vs. Cocoon	171%	135%	61%	42%	23%	6%
MOCOM vs. STRING	397%	123%	75%	71%	58%	21%
MOCOM vs. TOPIC	INF	607%	56%	80%	71%	36%

TABLE 5: Results of Mann-Whitney U Test (RQ1)

	k=3	k=5	k=10	k=20	k=50	k=100
MOCOM vs. ExReDiv	0.000**	0.000**	0.000**	0.000**	0.035**	0.630
MOCOM vs. MOOSE	0.000**	0.000**	0.000**	0.000**	0.000**	0.188
MOCOM vs. Cocoon	0.000**	0.000**	0.000**	0.000**	0.000**	0.008**
MOCOM vs. STRING	0.000**	0.000**	0.000**	0.000**	0.000**	0.000**
MOCOM vs. TOPIC	0.000**	0.000**	0.000**	0.000**	0.000**	0.000**

bugs can be detected with 10 recommended workers, and a median of 70% of all bugs can be detected with 20 recommended workers. This indicates the effectiveness of our approach.

Figure 4b shows *BDR@k* curve when k increases from 1 to 100. We can easily observe that *BDR@k* of MOCOM increases rapidly and reaches 75% when a median of 24 crowd workers are employed. This means with the crowd workers recommended by our proposed MOCOM, only a median of 24 workers are needed to detect 75% of all potential bugs.

In Figure 4a and 4b, we also present the *BDR@k* of *ground truth* (see Section 5.4). We can observe that when k is below 20, the bug detection rate (*BDR@k*) achieved by our proposed MOCOM is much higher than the *BDR@k* of groundtruth. This implies our proposed approach can detect more bugs with few crowd workers, which is the main concern and contribution of this work. We also notice that *BDR@k* of MOCOM is lower than *BDR@k* of groundtruth when k is larger than 20. We will explain the detailed reason in Section 7.1.

To further demonstrate the advantages of MOCOM, we then present the comparison between MOCOM and the five commonly-used and state-of-the-art baseline methods (details are in Section 5.4).

Figure 5 presents the *BDR@k* values of our propose MOCOM and the five baselines. We can easily observe that our proposed MOCOM is much more effective, considering the bug detection performance (*BDR@k*) of these recommended workers. This is particularly true when the k is smaller than 50.

Table 4 demonstrates the results of $\%\Delta$ between our proposed MOCOM and the five baselines. *BDR@5* of MOCOM has 50% to 607% improvement compared with the baselines, while *BDR@20* of MOCOM undergoes 19% to 80% improvement compared with the baselines.

We additionally conduct Mann-Whitney U Test for *BDR@k* between our proposed MOCOM and the five baseline approaches. Results show that for k is 3, 5, 10, 20, and 50, the *p-value* between our proposed MOCOM and each of the five baselines are all below 0.05 (details are in Table 5). This signifies that the bug detection performance of the crowd workers recommended by our approach is significantly better than existing approaches, which further indicates the advantages of our approach over the five commonly-used and state-of-the-art baseline methods.

Review 3-i Furthermore, unsurprisingly, the two baseline methods which are originally designed for test case selection (i.e., *STRING* and *TOPIC*) perform bad for crowd worker recommendation. And the three baseline methods

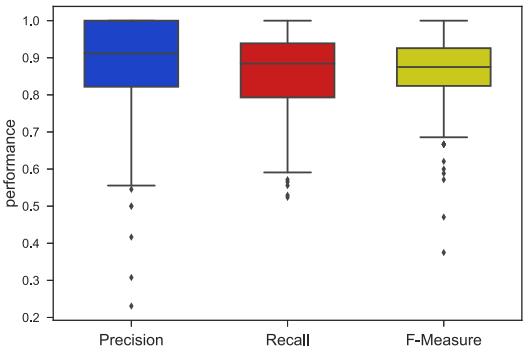


Fig. 6: Machine learner performance for bug detection probability (RQ2)

which are proposed specifically for crowd worker recommendation (i.e., *ExReDiv*, *MOOSE*, and *Cocoon*) perform better. This indicates, once again, the need of designing approach for crowd worker recommendation exclusively, because the characteristics of crowd workers are different from other objects (e.g., test case) in software testing context.

Among the three baselines for crowd worker recommendation (i.e., *ExReDiv*, *MOOSE*, and *Cocoon*), we can observe that *ExReDiv* is a little better than the other two. This might occur *ExReDiv* also considers the crowd worker's relevance with the test task, and the diversity among the selected crowd workers, although not as comprehensive as our proposed MOCOM. Moreover, experimental results show that relevance and diversity are important factors which should be considered in crowd worker recommendation (details are in Section 6.3).

6.2 Answering RQ2: Effectiveness of Machine Learning Model for Bug Detection Probability

This research question aims at investigating the effectiveness of the machine learning model in predicting the bug detection probability, which serves as one objective of MOCOM (Section 4.2.1). We use the three most commonly-used metrics (i.e., Precision, Recall, and F-Measure) [52] to measure the effectiveness of prediction. We treat the worker with a predicted probability greater than 0.5 as a bug finder, otherwise as not a bug finder. In the historical submitted reports, we can obtain who have detected bugs, and who have not. Hence, the three metrics are computed based on the worker's predicted bug detection results and the actual bug detection results.

Figure 6 presents the effectiveness for predicting the bug detection probability. We can easily see that our machine learning model can achieve high precision, recall, and f-measure. Specifically, the median precision is about 0.91, the median recall is about 0.89, and the median f-measure is 0.89. Furthermore, in 75% of the experimental projects, our machine learning model can achieve the precision of 0.82, the recall of 0.79, and the f-measure of 0.82. This implies that our machine learning model can predict the crowd worker's bug detection probability with high accuracy. Therefore, we can use the predicted bug detection probability as one objective in our multi-objective crowd worker recommendation approach.

TABLE 6: Feature importance rank(RQ2)

Rank	Feature	Rank	Feature
1	Number of bugs in past *2 weeks*	12	Number of projects in past *1 month*
2	Number of bugs in past *1 month*	13	Number of projects in the past
3	Percentage of bugs in past *2 weeks*	14	Number of reports in past *2 month*
4	Number of bugs in past *2 month*	15	Number of projects in past *2 month*
5	Number of projects in past *2 weeks*	16	Number of reports in past *1 month*
6	Number of reports in past *2 weeks*	17	Percentage of dup. in past *1 month*
7	Percentage of bugs in past *1 month*	18	Number of reports in the past
8	Number of bugs in the past	19	Percentage of bugs in the past
9	Percentage of dup. in past *2 weeks*	20	Percentage of dup. in past *2 month*
10	Time interval between last submission and task publish time	21	Percentage of dup. in the past
11	Percentage of bugs in past *2 month*		

We further explore the relative importance of different features in our machine learning model. We first obtain the Information Gain [52] for each feature in every project, then treat it as its rank and compute the average rank across all the experimental projects for each feature. Table 6 presents the rank of the features.

Generally speaking, the features which capture the more recent activities of crowd workers are ranked much higher. For example, the feature *number of bugs in past 2 weeks (1st rank)* is ranked higher than *number of bugs in the past (8th rank)*, while the former is about the workers' activity in the past 2 weeks and the latter is about the workers' activity in the whole past. This indicates the need of considering the time-related factors when modeling crowd worker's capability in predicting bug detection probability.

In addition, the features which relate with bug detection activity are ranked higher than these about general activities. For example, the feature *number of bugs in past 2 weeks (1st rank)* and *percentage of bugs in the past 2 weeks (3rd rank)* are ranked higher than *number of projects in past 2 weeks (5th rank)* and *number of reports in past 2 weeks (6th rank)*. This implies, compared with general activities in the crowdsourced testing platform, the past bug detection activity can better model the crowd worker's capability and bug detection probability.

6.3 Answering RQ3: Contribution of Each Objective

This research question is to evaluate the contribution of each objective, i.e., whether each of the applied objectives is necessary for our crowd worker recommendation. For the crowd worker recommendation problem, the objective of cost is indispensable, which cannot be removed. Hence, we remove each of the other three objectives, run MOCOM with the remaining objectives, and evaluate the bug detection rate (*BDR@k*).

Figure 7 presents the *BDR@k* under different settings, where *ALL* denotes using all the four objectives (i.e., our proposed MOCOM), *noCAP* denotes the recommendation without the objective *maximizing bug detection probability of workers*, *noREV* denotes the recommendation without the objective *maximizing relevance with the test task*, and *noDIV* denotes the recommendation without the objective *maximizing diversity of workers*.

We can easily observe that without any of the three objectives, the bug detection performance (*BDR@k*) would decline dramatically. This is particularly true when k is smaller

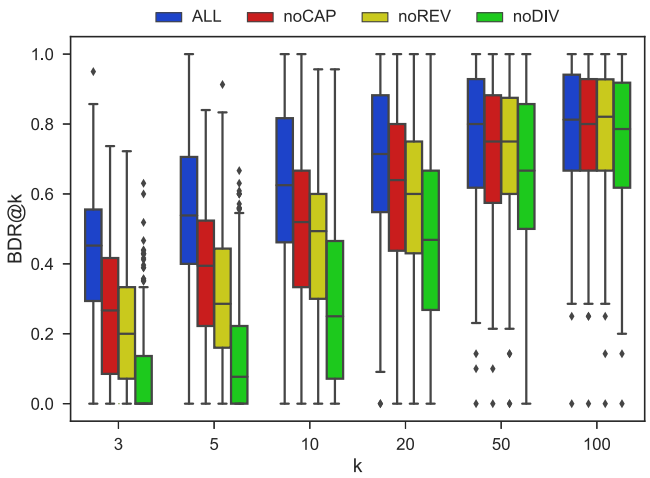


Fig. 7: Performance of MOCOM under different objectives (RQ3)

TABLE 7: Results of %Δ for BDR@k of different objectives (RQ3)

	k=3	k=5	k=10	k=20	k=50	k=100
ALL vs. noCap	69%	36%	20%	11%	6%	1%
ALL vs. noREV	126%	88%	26%	19%	6%	-1%
ALL vs. noDIV	INF	282%	119%	53%	23%	8%

TABLE 8: Results of Mann-Whitney U Test (RQ3)

	k=3	k=5	k=10	k=20	k=50	k=100
ALL vs. noCap	0.000**	0.000**	0.000**	0.000**	0.038**	0.556
ALL vs. noREV	0.000**	0.000**	0.000**	0.000**	0.011**	0.415
ALL vs. noDIV	0.000**	0.000**	0.000**	0.000**	0.000**	0.066

than 50. This indicates all the objectives are necessary for recommending an appropriate set of crowd workers.

Specifically, Table 7 demonstrates the results of %Δ for MOCOM with different objectives. *BDR@5* of our approach with all four objectives has 36% to 282% improvement compared with the recommendation with three objectives, while *BDR@20* of our approach with all four objectives undergoes 11% to 53% improvement compared with the recommendation with three objectives.

We additionally conduct Mann-Whitney U Test for *BDR@k* between the recommendation under all objectives (i.e., *ALL*) and the recommendation under partial objectives (i.e., *noCap*, *noREV*, and *noDIV*). Results show that when k is equal to 3, 5, 10, 20, and 50, the *p-value* between all objectives and partial objectives are all below 0.05 (details are in Table 8). This signifies that the bug detection performance between using all objectives and partial objectives is significantly different, which further indicates all the objectives are necessary, and they together contribute to the worker recommendation performance.

Furthermore, the bug detection performance would undergo the most dramatic decline without the diversity objective (i.e., *noDIV*). This might occur without considering diversity, the selected workers tend to possess similar background and would report duplicate bugs so as to influence their bug detection rate. This proves, once again, the importance of diversity in software testing context [40]–[43].

Review 4-1 Note that, one can easily generate a mistaken perception that bug detection probability is the most "bug-related" feature in our approach. Thus one may feel confused that *noCAP* (i.e., recommendation without the bug probability objective) can still results in a "still quite good" model. However, the bug detection probability in our model denotes the general ability of bug detection, rather than the specialized ability of detecting bugs for a specific task in testing dataset. This is because the features, utilized to build the machine learning model for capability prediction (in Table 3), all involve the general past activities of the crowd workers. This is why we include the second objective "relevance of crowd workers with test task" to help select the workers who are more capable for a specific task. The results of *noCAP* is better than *noREV* indicates that the general capability of bug detection contributes less to worker recommendation, while the specialized background with the task contributes more to worker recommendation.

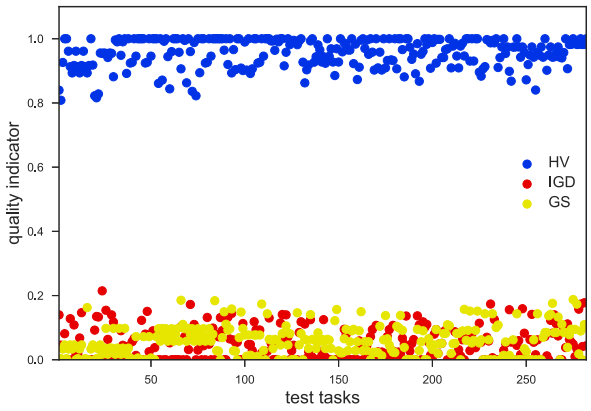


Fig. 8: Quality indicators (RQ4)

6.4 Answering RQ4: Quality of Optimization

Since MOCOM is a search-based approach, which produces Pareto fronts, this research question is to evaluate the quality of Pareto front, i.e., the quality of optimization. Three commonly-used quality indicators, i.e., HyperVolume (*HV*), Inverted Generational Distance (*IGD*), and Generalized Spread (*GS*) [24], are applied. For each test task, we present the value of each quality indicator obtained by MOCOM in Figure 8.

We can see that most projects have very high *HV* values, very low *IGD* values and very low *GS* values. The average *HV* is 0.95, the average *IGD* is 0.04, and the average *GS* is 0.05. This denotes our optimization has achieved high quality.

Existing researches on test case selection and worker selection achieve similar results [6], [57]. This further suggests that the results of MOCOM have high quality.

7 Discussion

7.1 Further Exploration of Results

In Section 6.1, we show that when k is below 20, the bug detection rate (*BDR@k*) achieved by MOCOM is much larger than that of ground truth, which implies the effectiveness of our approach. However, we also notice that *BDR@k* of

MOCOM is smaller than *BDR@k* of ground truth when k is larger than 20. In addition, the median *BDR@k* of ground truth can achieve 1.00, while the median *BDR@k* of MOCOM can only reach 0.80 even 100 workers are employed (Figure 4b). The possible reasons for this phenomenon are as follows.

Review 3-a Firstly, as our evaluation is conducted on the historical reports, we assume that the historical submitted bugs are the total number of bugs. This is why *BDR@k* of ground truth can achieve 1.00.

Review 3-d Secondly, we find that there are newcomers in some projects who do not have historical data. For these workers, we cannot model their capability and domain knowledge. Under this situation, our approach would not recommend them to perform test tasks. This is the cold-start problem in recommendation [58], which has not been well solved. In our experimental dataset, there are 128 test tasks whose *BDR@100* values are less than 0.80. Among these 128 test tasks, 43 (33.5%) tasks have newcomers, who would not be recommended by our approach. However they have detected 5% of total bugs in the ground truth.

To mitigate the impact of newcomers, we plan to incorporate the static attributes of crowd workers (e.g., occupation, interest) to help model the crowds. In addition, we suggest the project manager employ our recommended workers to find the 80% bugs with most of the budgets. Meanwhile, the same test task can also be delivered to the newcomers or other crowds with the leaving tiny proportion of budgets. With this varying pricing mechanism, our work recommendation approach can play a better role, and the crowdsourced task can be tested in a more cost-effective way.

Thirdly, there are some bug-finders who did not follow the mechanism we designed in this work. For example, we found several bug-finders did not submit any reports in the past six months, or their past experience are not tightly related with the task's test requirements. In this case, our approach has very low probability of recommending them to perform the test task. These outliers are common in recommendation problems, and because of this, almost all the recommendation problems can not achieve 100% recall [58]. We will explore other influential factors to better improve the recommendation results.

7.2 Benefits of MOCOM

Review 3-a As discussed in previous section, our approach can find a median of 75% bugs with fewer crowd workers, but might fail to find all bugs. Despite of this, we believe the value of our approach is finding more bugs, earlier (see Figure 4b). This is important because the goal of testing optimization in many circumstances is to shut down the testing process early (thereby saving the resources that would have otherwise been spent). For the teams handling the bugs, the goal is often not "find all bugs" but "find as many bugs using least resources as possible" (which, in our case, is the number of crowd workers) [59].

This section then discusses the benefits of MOCOM in terms of the four objectives. **Review 4-m** Section 7.2.1 will present the effectiveness of MOCOM in reducing cost when detecting equal number of bugs; Section 7.2.2 will show its effectiveness in reducing duplicate reports; Section 7.2.3 and 7.2.4 will demonstrate its effectiveness in recommending workers with high bug detection probability and relevance.

7.2.1 Reducing Cost

Figure 9a demonstrates the consumed cost of MOCOM and *ground truth* when detecting equal number of bugs. We can see that our proposed MOCOM consumes less cost than ground truth (i.e., current crowdsourced testing practice). For example, when detecting 15 bugs, with our MOCOM, the platform can save an average of 33% costs (i.e., (30-20)/30). The reduced cost is a tremendous figure when considering the large number of tasks delivered in a crowdtesting platform.

7.2.2 Reducing Duplicate Reports

Figure 9b demonstrates the *duplicate percentage* of MOCOM and *ground truth* in terms of the top k recommended crowd workers. *Duplicate percentage* is computed by the number of duplicates pairs divided by the total number of reports pairs. Note that, in Section 6.1, we conclude that our proposed MOCOM is effective especially when k is less than 20; thus we only present the results with k from 3 to 20 in this subsection and the next two subsections.

We can easily see that with MOCOM, the percentage of duplicate reports is less than current crowdsourced testing practice. This indicates the design of our approach (i.e., considering diversity) can help reduce duplicate reports so as to improve the cost-effectiveness.

7.2.3 Increasing Bug Detection Probability

Figure 9c presents the bug detection probability of the crowd workers (Section 4.2.1) for MOCOM and *ground truth* in terms of the top k recommended crowd workers. We can see that the recommended crowd workers have higher bug detection probability than current crowdsourced testing practice. This implies our approach can recommend crowd workers with higher bug detection probability so as to detect more bugs with less cost (see results in Section 6.1).

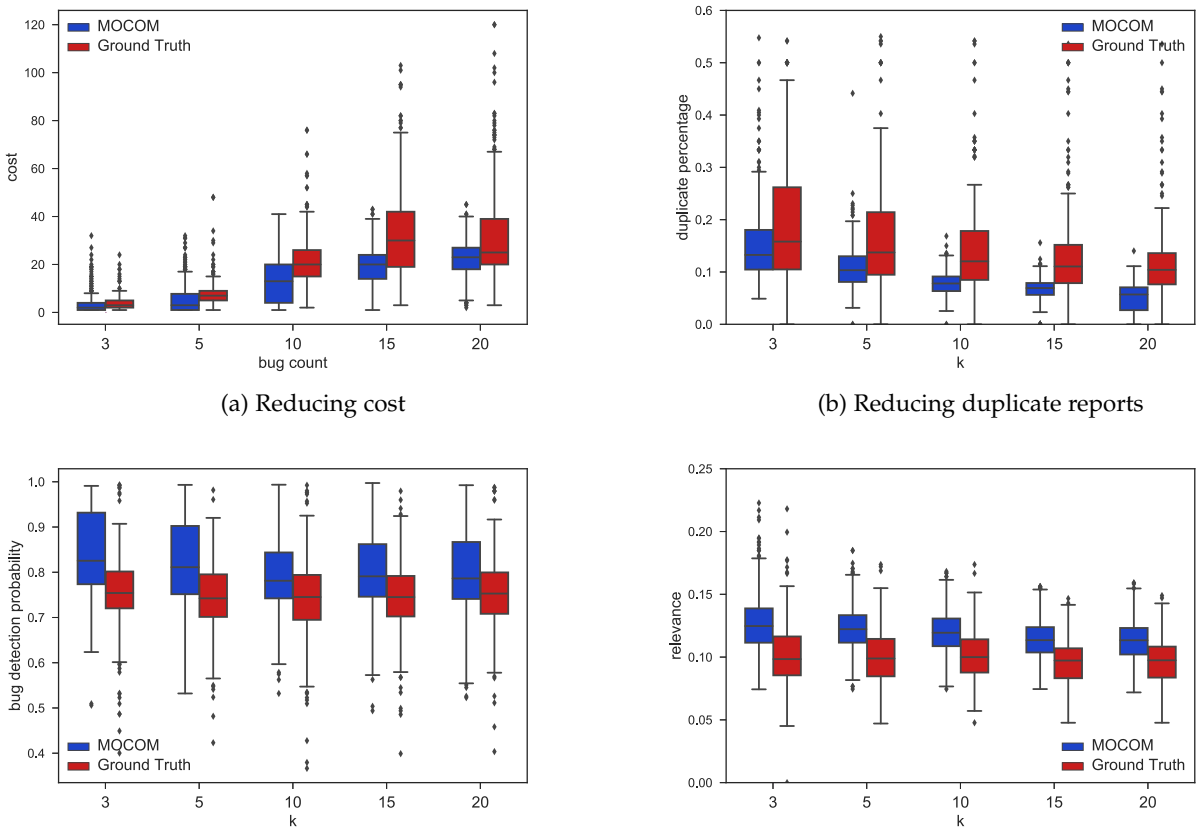
7.2.4 Increasing Relevance

Figure 9d shows the relevance of crowd workers with the test task (Section 4.2.2) for MOCOM and *ground truth* in terms of the top k recommended crowd workers. We can see that the recommended crowd workers have higher relevance with the test task than current crowdsourced testing practice. This suggests our approach can recommend crowd workers who have higher relevance with the test task so that they can detect more bugs (see results in Section 6.1).

7.3 Usefulness in Terms of Payout Schema

This section discusses whether our proposed MOCOM still works with other types of payout schema.

Paid by participation. Our proposed approach is based on the Baidu CrowdTest payout schema in which workers are equally paid when they submit reports in a test task. It is a commonly-used payout schema especially for the newly-launched platform because it can encourage crowd worker's participation [2]. Evaluation results show that



(a) Reducing cost

(b) Reducing duplicate reports

(c) Increasing bug detection probability

(d) Increasing relevance

Fig. 9: Benefits of MOCOM

MOCOM detects more bugs with fewer crowd workers (i.e., less cost) thus improves the cost-effectiveness of current crowdsourced testing practice.

Review 3-f **Paid by bug**. In this schema, only those crowd workers who detect bugs are paid (no matter whether it is a duplicate bug). It is also a commonly-used payout schema [2]. Because the reduced cost is measured by the number of workers in our current evaluation, for this payout schema, the reduced cost might not remain the same as current evaluation results. However, our evaluation results have also showed that with the recommended crowd workers, the number of duplicate reports is reduced (see Section 7.2.2). Since the duplicate reports also need to be paid, the reduction in duplicates can help save cost, as well as decrease the effort to manage these duplicates. Hence, in this schema, with our proposed approach, the costs-effectiveness of crowdsourced testing can also be improved.

Paid by first bug. In this schema, the crowd workers who detect the first bug are paid (the following duplicates would not be paid). It is another popular payout schema [26], in which any worker recommendation approach would not save cost because the total cost is the number of bugs contained in the software system. However, as our approach can detect more bugs with fewer crowd workers, for this payout schema, it means that, with our recommended crowd workers, bugs can be reported earlier than current crowdsourced testing practice. This is important since a large quantity of software are developed under agile model which calls for rapid iteration [60]. In addition, the reduction in duplicate reports by our approach can also help decrease the effort to manage the duplicates.

7.4 Threats to Validity

The threats to external validity concern the generality of this study. First, our experimental data consists of 562 test tasks collected from one of the largest crowdsourced testing platforms in China. The results of our study may not generalize beyond this environment where our experiments were conducted. However, the size of this dataset relatively reduces this threat.

The internal validity mainly concerns the implementation of baselines. Since the original implementation of STRING baseline and TOPIC baseline are not released, we have reimplemented our own version. We have strictly followed the procedures described in their work to alleviate this threat. These two approaches are designed for test case selection, while our aim is to select crowd workers which is different from test case. The original approaches treat test cases as strings of text, and we employ the text of worker's historical reports which is the most similar attribute in our context. Review 3-j In addition, we model the diversity of testing context based on the number of attributes. A refined modeling of diversity might improve the performance of worker recommendation, and we will explore other modeling manner of diversity in future.

The main threat to construct validity in this study involves the four objectives in multi-objective formulation. These four objectives are designed from different aspects: such as the bug detection probability of workers, the relevance with the test task, the diversity of workers, and the test cost. Although other objectives may also influence bug detection in crowdsourced testing, we have obtained promising results with these four objectives. Nevertheless, exploration of other objectives would further address this threat.

8 Conclusion

In crowdsourced testing, it is of great value to recommend a set of appropriate crowd workers for a test task so that more software bugs can be detected with fewer workers. We first present a new characterization of crowd workers which can support more effective crowd worker recommendation. We characterize the crowd workers with three dimensions, i.e., testing context, capability, and domain knowledge. Based on the characterization, we then propose Multi-Objective Crowd wOrker recoMmendation approach (MOCOM), which aims at recommending a minimum number of crowd workers who could detect the maximum number of bugs for a crowdsourced testing task. Specifically, MOCOM recommends crowd workers by maximizing the bug detection probability of workers, the relevance with the test task, the diversity of workers, and minimizing the test cost.

We experimentally evaluate our approach on 562 test tasks (involving 2,405 crowd workers and 78,738 test reports) from one of the Chinese largest crowdsourced testing platforms. The experimental results show that our MOCOM can detect more bugs with fewer crowd workers, in which a median of 24 crowd workers can detect 75% of all potential bugs. All the objectives are necessary for worker recommendation because removing any of the objectives would result in a significant performance decline. In addition, our approach also significantly outperforms five commonly-used and state-of-the-art baseline methods, with 19% to 80% improvement at *BDR@20*.

Acknowledgments

This work is supported by the National Natural Science Foundation of China under grant No.61602450, No.61432001, and China Scholarship Council. We would like to thank the testers in Baidu for their extensive efforts in supporting this work.

References

[1] F. Brooks, Jr., *The Mythical Man-Month: Essays on Software Engineering*. MA: Addison-Wesley, 1995.
[2] X. Zhang, Y. Feng, D. Liu, Z. Chen, and B. Xu, "Research progress of crowdsourced software testing," *Journal of Software*, vol. 29(1), pp. 69–88, 2018.
[3] http://www.softwaretestinghelp.com/crowdsourced-testing-companies/.
[4] M. Hosseini, K. Phalp, J. Taylor, and R. Ali, "The four pillars of crowdsourcing: A reference model," in *Research Challenges in Information Science (RCIS), 2014 IEEE Eighth International Conference on*. IEEE, 2014, pp. 1–12.
[5] Q. Cui, J. Wang, G. Yang, M. Xie, Q. Wang, and M. Li, "Who should be selected to perform a task in crowdsourced testing?" in *COMPSAC'17*, 2017, pp. 75–84.
[6] Q. Cui, S. Wang, J. Wang, Y. Hu, Q. Wang, and M. Li, "Multi-objective crowd worker selection in crowdsourced testing," in *SEKE'17*, 2017, pp. 218–223.
[7] M. Xie, Q. Wang, G. Yang, and M. Li, "Cocoon: Crowdsourced testing quality maximization under context coverage constraint," in *ISSRE'17*, 2017, pp. 316–327.
[8] J. Anvik, L. Hiew, and G. C. Murphy, "Who should fix this bug?" in *ICSE'06*, 2006, pp. 361–370.
[9] G. Jeong, S. Kim, and T. Zimmermann, "Improving bug triage with bug tossing graphs," in *FSE'09*, 2009, pp. 111–120.
[10] J. Xuan, H. Jiang, Z. Ren, and W. Zou, "Developer prioritization in bug repositories," in *ICSE'12*, 2012, pp. 25–35.
[11] S. Wang, W. Zhang, Y. Yang, and Q. Wang, "Devnet: exploring developer collaboration in heterogeneous networks of bug repositories," in *ESEM'13*, 2013, pp. 193–202.
[12] W. Zhang, S. Wang, Y. Yang, and Q. Wang, "Heterogeneous network analysis of developer contribution in bug repositories," in *CSC'13*, 2013, pp. 98–105.
[13] S. Wang, W. Zhang, and Q. Wang, "Fixercache: Unsupervised caching active developers for diverse bug triage," in *ESEM'14*, 2014, p. 25.
[14] P. Bhattacharya and I. Neamtiu, "Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging," in *ICSM'10*, 2010, pp. 1–10.
[15] D. Matter, A. Kuhn, and O. Nierstrasz, "Assigning bug reports using a vocabulary-based expertise model of developers," in *MSR'09*, pp. 131–140.
[16] A. Tamrawi, T. T. Nguyen, J. M. Al-Kofahi, and T. N. Nguyen, "Fuzzy set and cache-based approach for bug triaging," in *FSE'11*, pp. 365–375.
[17] H. Naguib, N. Narayan, B. Brügge, and D. Helal, "Bug report assignee recommendation using activity profiles," in *MSR'13*, pp. 22–30.
[18] G. Canfora, M. Di Penta, R. Oliveto, and S. Panichella, "Who is going to mentor newcomers in open source projects?" in *FSE'12*, p. 44.
[19] D. Ma, D. Schuler, T. Zimmermann, and J. Sillito, "Expert recommendation with usage expertise," in *ICSM'09*, pp. 535–538.
[20] H. Yang, X. Sun, B. Li, and Y. Duan, "DR_PSF: Enhancing developer recommendation by leveraging personalized source-code files," in *COMPSAC'16*, vol. 1, 2016, pp. 239–244.
[21] K. Mao, Y. Yang, Q. Wang, Y. Jia, and M. Harman, "Developer recommendation for crowdsourced software development tasks," in *SOSE'15*, 2015, pp. 347–356.
[22] Y. Yang, M. R. Karim, R. Saremi, and G. Ruhe, "Who should take this task?: Dynamic decision support for crowd workers," in *ESEM'16*, 2016, p. 8.
[23] B. Ye and Y. Wang, "CrowdRec: Trust-aware worker recommendation in crowdsourcing environments," in *ICWS'16*, 2016, pp. 1–8.
[24] S. Wang, S. Ali, T. Yue, Y. Li, and M. Liaaen, "A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering," in *ICSE'16*, 2016, pp. 631–642.
[25] C. Henard, M. Papadakis, M. Harman, and Y. L. Traon, "Combining multi-objective search and constraint solving for configuring large software product lines," in *ICSE'15*, pp. 517–528.
[26] https://help.utest.com/submitting-bug-reports-utest/.
[27] N. Leicht, I. Blohm, and J. M. Leimeister, "Leveraging the power of the crowd for software testing," *IEEE Software*, vol. 34, no. 2, pp. 62–69, 2017.
[28] N. Chen and S. Kim, "Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles," in *ASE'12*, pp. 140–149.
[29] R. Musson, J. Richards, D. Fisher, C. Bird, B. Bussone, and S. Ganguly, "Leveraging the crowd: how 48,000 users helped improve lync performance," *IEEE software*, vol. 30, no. 4, pp. 38–45, 2013.
[30] V. H. Gomide, P. A. Valle, J. O. Ferreira, J. R. Barbosa, A. F. Da Rocha, and T. Barbosa, "Affective crowdsourcing applied to usability testing," *International Journal of Computer Scienceand Information Technologies*, vol. 5, no. 1, pp. 575–579, 2014.
[31] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *Mobile Software Engineering and Systems (MOBILESoft), 2016 IEEE/ACM International Conference on*, 2016, pp. 88–99.
[32] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu, "Test report prioritization to assist crowdsourced testing," in *FSE'15*, pp. 225–236.

[33] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, "Multi-objective test report prioritization using image understanding," in *ASE'16*, pp. 202–213.

[34] J. Wang, Q. Cui, Q. Wang, and S. Wang, "Towards effectively test report classification to assist crowdsourced testing," in *ESEM'16*, p. 6.

[35] J. Wang, S. Wang, Q. Cui, and Q. Wang, "Local-based active classification of test report to assist crowdsourced testing," in *ASE'16*, pp. 190–201.

[36] J. Wang, Q. Cui, S. Wang, and Q. Wang, "Domain adaptation for test report classification in crowdsourced testing," in *ICSE-SEIP'17*, pp. 83–92.

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

[38] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in oss community," in *ICSE'12*, pp. 518–528.

[39] ——, "Who will stay in the floss community? modeling participant's initial behavior," *IEEE Transactions on Software Engineering*, vol. 41, no. 1, pp. 82–99, 2015.

[40] Y. Ledru, A. Petrenko, and S. Boroday, "Using string distances for test case prioritisation," in *ASE'09*, 2009, pp. 510–514.

[41] Y. Ledru, A. Petrenko, S. Boroday, and N. Mandran, "Prioritizing test cases with string distances," *Automated Software Engineering*, vol. 19, no. 1, pp. 65–95, 2012.

[42] S. W. Thomas, H. Hemmati, A. E. Hassan, and D. Blostein, "Static test case prioritization using topic models," *Empirical Software Engineering*, vol. 19, no. 1, pp. 182–212, 2014.

[43] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, L. Zhang, and B. Xie, "Test case prioritization for compilers: A text-vector based approach," in *ICST'16*, pp. 266–277.

[44] J. Chen, Y. Bai, D. Hao, Y. Xiong, H. Zhang, and B. Xie, "Learning to prioritize test programs for compiler testing," in *ICSE'17*, 2017, pp. 700–711.

[45] S. Wang, J. Nam, and L. Tan, "QTEP: quality-aware test case prioritization," in *FSE'17*, 2017, pp. 523–534.

[46] G. Rothermel, R. H. Untch, C. Chu, and M. r. J. Harrold, "Prioritizing test cases for regression testing," *TSE'11*, vol. 27, no. 10, pp. 929–948, 2011.

[47] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *STVR'12*, vol. 22, no. 2, pp. 67–120, 2012.

[48] Z. Li, M. Harman, and R. M. Hierons, "Search algorithms for regression test case prioritization," *TSE'07*, vol. 33, no. 4, pp. 225–237, 2007.

[49] G. Rothermel, R. H. Untch, C. Chu, and M. r. J. Harrold, "Test case prioritization: An empirical study," in *ICSM'99*, 1999, pp. 179–188.

[50] H. Mei, D. Hao, L. Zhang, L. Zhang, J. Zhou, and G. Rothermel, "A static approach to prioritizing junit test cases," *TSE'12*, vol. 38, no. 6, pp. 1258–1275, 2012.

[51] A. Shi, T. Yung, A. Gyori, and D. Marinov, "Comparing and combining test-suite reduction and regression test select ion," in *FSE'15*, pp. 237–247.

[52] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[53] H. Rocha, M. T. Valente, H. Marques-Neto, and G. C. Murphy, "An empirical study on recommendations of similar bugs," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, 2016, pp. 46–56.

[54] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Computing Surveys (CSUR)*, vol. 45, no. 1, p. 11, 2012.

[55] K. Deb, S. Agrawal, A. Pratap, and T. Meyarivan, "A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-II," in *International Conference on Parallel Problem Solving From Nature*. Springer, 2000, pp. 849–858.

[56] J. H. Holland, *Genetic algorithms*. Scientific American, 1992.

[57] M. G. Epitropakis, S. Yoo, M. Harman, and E. K. Burke, "Empirical evaluation of pareto efficient multi-objective regression test case prioritisation," in *ISSTA 2015*, 2015, pp. 234–245.

[58] F. Ricci, L. Rokach, B. Shapira, and P. B. Kantor, *Recommender systems handbook*. Springer, 2015.

[59] S. Elbaum, G. Rothermel, and J. Penix, "Techniques for improving regression testing in continuous integration development environments," in *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2014, 2014, pp. 235–245.

[60] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.