Context-aware In-process Crowdworker Recommendation

Junjie Wang[1,2,3], Ye Yang[4], Song Wang[5], Yuanzhe Hu[1,3], Dandan Wang[1,3], Qing Wang[1,2,3,*]
[1]Laboratory for Internet Software Technologies, [2]State Key Laboratory of Computer Sciences,
Institute of Software Chinese Academy of Sciences, Beijing, China;
[3]University of Chinese Academy of Sciences, Beijing, China; *Corresponding author
[4]School of Systems and Enterprises, Stevens Institute of Technology, USA;
[5]Lassonde School of Engineering, York University, Canada;
{junjie,wq}@iscas.ac.cn,ye.yang@stevens.edu,wangsong@eecs.yorku.ca

ABSTRACT

Identifying and optimizing open participation is essential to the success of open software development. Existing studies highlighted the importance of worker recommendation for crowdtesting tasks in order to detect more bugs with fewer workers. However, these studies mainly focus on one-time recommendations with respect to the initial context at the beginning of a new task. This paper argues the need for in-process crowdtesting worker recommendation. We motivate this study through a pilot study, revealing the prevalence of long-sized non-yielding windows, i.e., no new bugs are revealed in consecutive test reports during the process of a crowdtesting task. This indicates the potential opportunity for accelerating crowdtesting by recommending appropriate workers in a dynamic manner, so that the non-yielding windows could be shortened.

To that end, this paper proposes a context-aware in-process crowdworker recommendation approach, `iRec`, to detect more bugs earlier and potentially shorten the non-yielding windows. It consists of three main components: 1) the modeling of dynamic testing context, 2) the learning-based ranking component, and 3) the diversity-based re-ranking component. The evaluation is conducted on 636 crowdtesting tasks from one of the largest crowdtesting platforms, and results show the potential of `iRec` in improving the cost-effectiveness of crowdtesting by saving the cost and shortening the testing process.

ACM Reference Format:
Junjie Wang[1,2,3], Ye Yang[4], Song Wang[5], Yuanzhe Hu[1,3], Dandan Wang[1,3], Qing Wang[1,2,3,*]. 2020. Context-aware In-process Crowdworker Recommendation. In *42nd International Conference on Software Engineering (ICSE '20), May 23–29, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3377811.3380380

1 INTRODUCTION

Abundant internet resources has driven software engineering activities to be more open than ever. Besides free, successful open source software and cheap, on-demand web storage and computation facilities, more and more companies are leveraging on crowdsourced software development to obtain solutions and achieve quality objectives faster, cheaper [1–3]. As an example, uTest has more than 400,000 software experts with diverse expertise spanning more than 200 countries to validate various aspects of digital quality [2].

Various methods and approaches have been proposed to support utilizing crowdtesting to substitute or aid in-house testing for reducing cost, improving quality, and accelerating schedule [19, 24, 53, 66]. One of the most essential functions is to identify appropriate workers for a particular testing task [13, 14, 51, 60]. This is because the shared crowdworker resources, while cheap, are not free. To help identify appropriate workers for crowdtesting tasks, many different approaches have been proposed by modeling the workers' testing environment [51, 60], experience [13, 60], capability [51], or expertise with the task [13, 14, 51], etc. Unfortunately, these approaches have limited applicability for the highly dynamic and volatile crowdtesting processes. They merely provide one-time recommendation at the beginning of a new task, without considering constantly changing context information of ongoing testing processes.

This study aims at filling in this gap and shedding light on the necessity and feasibility of dynamically in-process worker recommendation. From a pilot study conducted on real-world crowdtesting data (Section 2.2), this study first reveals the prevalence of long-sized ***non-yielding windows***, i.e., consecutive testing reports containing no new bugs during crowdtesting process. 84.5% tasks have at least one 10-sized non-yielding window, and an average of 39% of spending is wasted on these non-yielding windows. This indicates the ineffectiveness of current crowdtesting practice because these non-yielding windows would 1) cause wasteful spending of task requesters; 2) potentially delay the progress of crowdtesting. It also implies the potential opportunity for accelerating testing process by recommending appropriate crowdworkers in a dynamic manner, so that the non-yielding windows could be shortened.

This paper proposes a context-aware in-process crowdworker recommendation approach (named `iRec`) to dynamically recommend a diverse set of capable crowdworkers based on various contextual information at a specific point of crowdtesting process, aiming at shortening the non-yielding window and improving bug detection efficiency.

`iRec` consists of three main components: testing context modeling, learning-based ranking, and diversity-based re-ranking. First, the testing context model is constructed in two perspectives, i.e., process context and resource context, to capture the in-process progress-oriented information and crowdworkers' characteristics respectively. Second, a total of 26 features are defined and extracted from both process context and resource context; based on these

ICSE '20, May 23–29, 2020, Seoul, Republic of Korea

ACM ISBN 978-1-4503-7121-6/20/05…$15.00
https://doi.org/10.1145/3377811.3380380

features, the learning-based ranking component learns the probability of crowdworkers being able to detect bugs within specific context. Third, the diversity-based re-ranking component adjusts the ranked list of recommended workers based on the dynamic diversity measurement to potentially reduce duplicate bugs.

iRec is evaluated on 636 crowdtesting tasks (involving 2,404 crowdworkers and 80,200 reports) from one of the largest crowdtesting platforms. Results show that iRec could shorten the non-yielding window by a median of 50% - 58% in different application scenarios, and consequently have potential of saving testing cost by a median of 8% - 12%. It significantly outperforms four commonly-used and state-of-the-art baseline approaches.

This paper makes the following contributions:

- The formation of the in-process crowdworker recommendation problem based on the empirical investigation on real-world crowdtesting data. **This is the first study to explore the in-process worker recommendation problem to the best of our knowledge.**
- The crowdtesting context model which consists of two perspectives, i.e., process context and resource context to facilitate in-process crowdworker recommendation.
- The development of the learning-based ranking method to learn appropriate crowdworkers who can detect bugs in a dynamic manner.
- The development of the diversity-based re-ranking method to adjust the ranked workers to reduce duplicate bugs.
- The evaluation of the proposed iRec on 636 crowdtesting tasks (involving 2,404 crowdworkers and 80,200 reports) from one of the largest crowdsourced testing platforms, with affirmative results[1].

2 BACKGROUND AND MOTIVATION

2.1 Background

In practice, a task requester prepares the task (including the software under test and test requirements), and distributes it online. Crowdworkers can freely sign in their interested tasks and submit testing reports in exchange of monetary prizes. Managers then inspect and verify each report to find the detected bugs. There are different payout schema in crowdtesting [53, 66], e.g., pay by report. As discussed in previous work [51, 53], the cost of a task is positively correlated with the number of received reports.

The following lists important concepts with examples in Table 1:

Test Task is the input to a crowdtesting platform provided by a task requester. It contains a task ID, and a list of test requirements in natural language.

Test Report is the test record submitted by a crowdworker. It contains a report ID, a worker ID (i.e., who submit the report), a task ID (i.e., which task is conducted), the description of how the test was performed and what happened during the test, bug label, duplicate label, and submission time. Specifically, bug label indicates whether the report contains a bug[2]; and duplicate label indicates with which the report is duplicate. Note that, in the following paper, we refer to "bug report" (also short for "bug") as the report whose bug label is *bug*, refer to "test report" (also short for "report") as any submitted report, and refer to "unique bug" as the report whose bug label is *bug* and duplicate label is *null*.

Crowdworker is a registered worker in a crowdtesting platform, and is denoted by worker ID, and his/her device. It is associated with the historical reports he/she submitted. Note that, in our experimental dataset which spans across six months, we did not observe the crowdworkers' device change; thus this paper assumes each crowdworker corresponds to a stable device variable.

Table 1: Important concepts and examples

Test Task	
Task ID	T000012
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
Submission time	Jan 30. 2016 15:32
Crowdworker	
Worker Id	W5124983210
Device	Phone type: *Samsung SN9009* Operating system: *Android 4.4.2* ROM type: *KOT49H.N9009* Network environment: *WIFI*
Historical Reports	R1002948308, R1037948352

2.2 Non-yielding Windows in Crowdtesting Processes

Most open call formats of crowdtesting frequently lead to ad hoc worker behaviors and ineffective outcomes. In some cases, workers may choose tasks they are not good at and end up with finding none bugs. In other cases, many workers with similar experience may submit duplicate bug reports and cause wasteful spending of the task requester. More specifically, an average of 80% duplicate reports are observed in our dataset.

To better understand this issue, we examine the bug arrival curve for 636 historical tasks from real-world crowdtesting projects (details are in Section 4.2). We notice that there are frequently *non-yielding windows*, i.e., the flat segments, of the increasing bug arrival curve. Such flat windows correspond to a collection of test reports failing to reveal new bugs, i.e., either no bugs or only duplicate bugs. We refer to the length of a non-yielding window as the number of consecutive test reports.

Figure 1 illustrates the bug arrival curve of an example task with highlighted non-yielding windows (length >10, only for illustration purpose). The non-yielding windows can 1) cause wasteful spending on these non-yielding reports; 2) potentially delay the progress of crowdtesting.

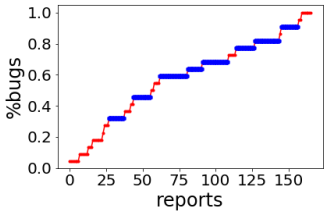
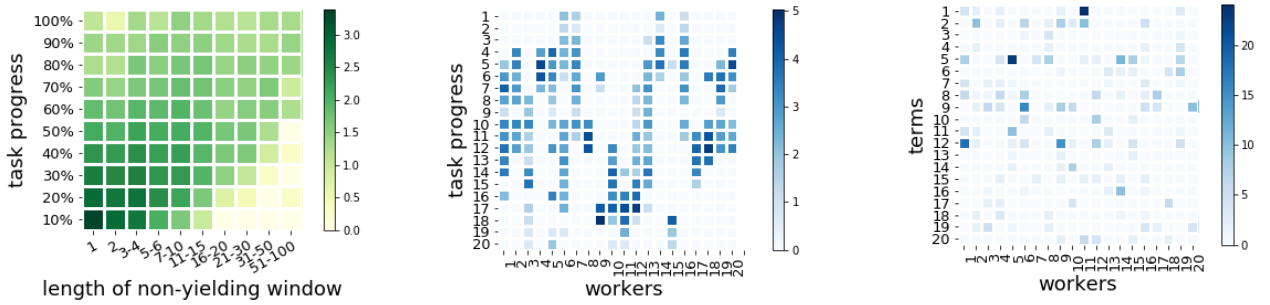


Figure 1: *Bug arrival curve*

[1]https://github.com/wangjunjieISCAS/InProcessRecommendation

[2]In our experimental platform, a report corresponds to either 0 or 1 bug, and there is no report containing more than 1 bug.



(a) Non-yielding windows (b) Crowdworkers' activeness (c) Crowdworkers' preference

Figure 2: Observations based on Baidu dataset

We further investigate this phenomenon and present a summarized view in Figure 2a. The x-axis shows the length of the non-yielding window, while the y-axis shows the relative position of the non-yielding window expressed using the task's progress. We can observe that the long-sized non-yielding window is quite common during crowdtesting process. There are 84.5% (538/636) tasks with at least one 10-sized non-yielding window, 67.8% (431/636) tasks with at least one 15-sized window. Furthermore, these long-sized non-yielding windows mainly take place in the second half of crowdtesting processes. For example, 90.7% (488/538) 10-sized non-yielding windows happened at the latter half of the process.

We further explore the cost waste of these non-yielding windows. Specifically, an average of 39% cost[3] is wasted on these 10- or longer-sized non-yielding windows of all experimental tasks, and an average of 32% cost is wasted on these 15- or longer-sized non-yielding windows. In addition, an average of 33 hours[4] are spent on these 10- or longer-sized non-yielding windows of all experimental tasks.

The prevalence of long-sized non-yielding windows indicates that current workers possibly have similar bug detection capability with previous workers on the same task. In order to break the flatness, we investigate the potential root causes and study if we can learn from the dynamic, underlying contextual information in order to mitigate such situation. This also suggests the unsuitability of existing one-time worker recommendation approaches, and indicates the need for in-process crowdworker recommendation.

2.3 Characterizing CrowdWorker's Bug Detection Capability

This subsection presents more explorations about the characteristics of crowdworkers which can influence their test participation and bug detection performance to motivate the modelings of testing context.

Activeness. Figure 2b shows the distribution of crowdworkers' activity intensity. The x-axis is the random-selected 20 crowdworkers among the top-50 workers ranked by the number of submitted reports, and the y-axis is 20 equal-sized time interval which is obtained by dividing the whole time space. We color-code the blocks, using a darker color to denote a worker submitting more reports during the specific time interval. We can see that the crowdworkers' activities are greatly diversified and not all crowdworkers are equally active in the crowdtesting platform at specific time. Intuitively, the inactive crowdworkers would be less likely to conduct the task, let alone detect bugs.

Preference. Figure 2c shows the distribution of crowdworkers' activity at a finer granularity. The x-axis is the same as Figure 2b, and the y-axis is the random-selected 20 terms (which capture the content under testing) from the top-50 most popular descriptive terms (see Section 3.1 for details). The block in the heat map demonstrates the number of reports which are submitted by the specific worker and contain the specific term. We color-code the blocks, using a darker color to denote a worker submitting reports with corresponding terms more frequently, i.e., worker's preference in different aspects. The differences across columns in the heat map further reveal the diversified preference across workers. Considering there are usually dozens of crowdtesting tasks open in the platform, even if a crowdworker is active, he/she cannot take all tasks. Intuitively, if a crowdworker has a preference on the specific aspects of a task, he/she would show greater willingness in taking the task and further detecting bugs.

Expertise. Similarly, we explore the heat map with the terms from the crowdworkers' *bug reports* (rather than *reports*), we observe a similar trend. Due to space limit, we leave the detailed figure in our website. This indicates the crowdworkers' diversified expertise over different crowdtesting tasks. We also conduct correlation analysis between the number of bug reports (i.e., denoting expertise) and number of reports (i.e., denoting preference) for each pair of the 20 crowdworkers on the top-50 most popular terms, the median coefficients is 0.26 indicating these two types of characteristics are not tightly correlated with each other. *Preference* focuses more on whether a crowdworker would take a specific task, and *expertise* focuses more on whether a crowdworker can detect bugs in the task.

To summarize, the exploration results reveal that workers have greatly diversified activeness, preferences, and expertise, which significantly affect their availability on the platform, choices of tasks, and quality of their submissions. To guarantee the effectiveness of recommendation, a worker is desirable to be active in the platform, and equipped with satisfactory preference and expertise for the given tasks. Thus, all these factors need to be precisely captured

[3]Following previous work [51, 53], we treat the number of reports as the amount of cost.

[4]We measure the duration of each non-yielding window using the time difference between the last and first report's submission time associated with that window.

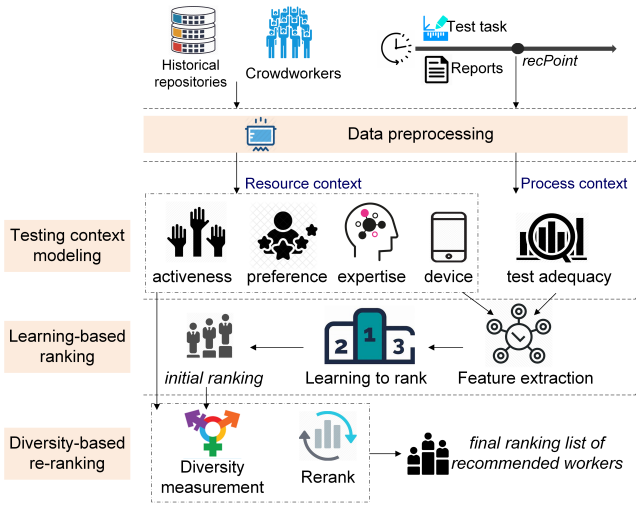


Figure 3: Overview of iRec

and jointly considered within the recommendation approach. Besides, the approach should also consider the diversity among the recommended set of workers so as to reduce duplicates and further improve bug detection performance.

3 APPROACH

Figure 3 shows the overview of the proposed iRec. It can be automatically triggered when the size of non-yielding window exceeding a certain threshold value (i.e., *recThres*) is observed during crowdtesting process, as introduced in Section 2.2. For brevity, we use the term *recPoint* to denote the point of time under recommendation, as illustrated at the top-right corner of Figure 3.

iRec has three main components. First, it models the time-sensitive testing contextual information in two perspectives, i.e., the process context and the resource context, respectively, with respect to the *recPoint* during the crowdtesting process. The process context characterizes the process-oriented information related to the crowdtesting progress of the current task, while resource context reflects the availability and capability factors concerning the competing crowdworker resources in the crowdtesting platform. Second, a learning-based ranking component extracts 26 features from both process context and resource context, and learns the success knowledge of the most appropriate crowdworkers, i.e., the workers with the greatest potential to detect bugs abstracted from historical tasks. Third, a diversity-based re-ranking component adjusts the ranked list of recommended workers by optimizing the worker diversity in order to potentially reduce duplicate bugs.

3.1 Data Preprocessing

To extract the time-sensitive contextual information at *recPoint*, the following data are obtained for further processing (refer to Section 2.1 for more details of these concepts): 1) *test task*: the specific task currently under testing and recommendation; 2) *test reports*: the set of already received reports for this specific task up till the *recPoint*; 3) all registered *crowdworkers* (with historical reports a crowdworker submitted, including reports in this specific task); 4) historical test tasks.

There are two types of textual documents in our data repository: one is test reports and the other is test requirements. Following the existing studies [48, 52], each document goes through standard word segmentation, stopwords removal, with synonym replacement being applied to reduce noise. As an output, each document is represented using a vector of terms.

Descriptive term filtering. After the above steps, we find that some terms may appear in a large number of documents, while some other terms may appear in only very few documents. Both of them are less predictive and contribute less in modeling the testing context. Therefore, we construct a *descriptive terms list* to facilitate the effective modeling. We first preprocess all the documents in the training dataset (see Section 4.3) and obtain the terms of each document. We rank the terms according to the number of documents in which a term appears (i.e., document frequency, also known as *df*), and filter out 5% terms with the highest document frequency and 5% terms with the lowest document frequency (i.e., less predictive terms) following previous work [13, 51]. Note that, since the documents in crowdtesting are often short, the term frequency (also known as *tf*), which is another commonly-used metric in information retrieval [43], is not discriminative, so we only use document frequency to rank the terms. In this way, the final *descriptive terms list* is formed and used to represent each document in the vector space of the descriptive terms.

3.2 Testing Context Modeling

The testing context model is constructed in two perspectives, i.e., process context and resource context, to capture the in-process progress-oriented information and crowdworkers' characteristics respectively.

3.2.1 ***Process Context***. To model the process context of a crowdtesting task, we first represent the task's requirements in the vector space of descriptive terms list and denote it as *task terms vector*. We then use the notion of **test adequacy** to measure the testing progress regarding to what degree each descriptive term of task requirements (i.e., task terms vector) has been tested.

TestAdeq: the degree of testing for each descriptive term t_j in task terms vector. It is measured as follows:

$$TestAdeq(t_j) = \frac{number\ of\ bug\ reports\ with\ t_j}{number\ of\ received\ bug\ reports\ in\ a\ task} \quad (1)$$

where $t_j \in$ *task terms vector*. The larger $TestAdeq(t_j)$, the more adequate of testing for the corresponding aspects of the task. This definition enables the learning of underlying knowledge to match workers' expertise or preference with inadequate-tested terms at a finer granularity.

3.2.2 ***Resource Context***. Based on the observations from Section 2.3, *activeness, preference, and expertise of crowdworkers* are integrated to model the resource context of a general crowdtesting platform. In addition, we include *device of crowdworkers* as a separate dimension of resource context, since several studies reported its diversifying role in crowdtesting environment [51, 60].

1) **Activeness** measures the degree of availability of crowdworkers to represent relative uncertainty associated with inactive crowdworkers. Activeness of a crowdworker w is characterized using the following four attributes :

LastBug: Duration (in hours) between *recPoint* and the time when worker w's last *bug* is submitted.

LastReport: Duration (in hours) between *recPoint* and the time when worker w's last *report* is submitted.

NumBugs-X: Number of bugs submitted by worker w in past X time, e.g., past 2 weeks.

NumReports-X: Number of reports submitted by worker w in past X time, e.g., past 8 hours.

Based on the concepts in Table 1, we can derive the above attributes of worker w from the historical reports submitted by him/her.

2) **Preference** measures to what degree a potential crowdworker might be interested in a candidate task. The higher the preference, the greater the worker's willingness/potential in taking the task/detecting bugs. Preference of a crowdworker w is characterized using the following attribute:

ProbPref: the preference of worker w regarding each descriptive term. In other words, it is the probability of recommending the worker w when aiming at generating a report with specific term t_j. It is measured based on bayes rules [41] as follows:

$$ProbPref(w, t_j) = P(w|t_j) = \frac{tf(w, t_j)}{\sum_{w_k} tf(w_k, t_j)} \cdot \frac{\sum_{w_k} df(w_k)}{df(w)} \quad (2)$$

where $tf(w, t_j)$ is the number of occurrences of t_j in historical reports of worker w, $df(w)$ is the total number of reports submitted by worker w, and k is an iterator over all available crowdworkers at the platform.

As mentioned in Section 3.1, after data preprocessing, each report is expressed with a set of descriptive terms. This attribute can be derived from the crowdworker's historical submitted reports.

3) **Expertise** measures a crowdworker's capability in detecting bugs. When a crowdworker brings in matching expertise required for the given task, he/she would have greater possibility in detecting bugs. Expertise of a crowdworker w is characterized using the following attribute:

ProbExp: the expertise of worker w regarding each descriptive term. It is measured similarly as *ProbPref* as follows:

$$ProbExp(w, t_j) = P(w|t_j) = \frac{tf(w, t_j)}{\sum_{w_k} tf(w_k, t_j)} \cdot \frac{\sum_{w_k} df(w_k)}{df(w)} \quad (3)$$

where $tf(w, t_j)$ is the number of occurrences of t_j in historical *bug reports* of worker w, $df(w)$ is the total number of *bug reports* submitted by worker w, and k is an iterator over all available crowdworkers at the platform.

The difference between *ProbProf* and *ProbExp* is that the former is measured based on worker's submitted *reports*, while the latter is based on worker's submitted *bug reports*, following the motivating studies in Section 2.3. The reason why we characterize expertise in terms of each term is because it enables the more precise matching with the inadequate-tested terms, and the identification of more diverse workers for finding unique bugs in a much-finer granularity.

4) **Device** measures the device-related attributes of the crowdworker which is critical in testing an application and in revealing device-related bugs [56]. Device of a crowdworker w is characterized using all his/her device-related attributes including: ***Phone type*** used to run the testing task, ***Operating system*** of the device model, ***ROM type*** of the phone, ***Network environment*** under which a task is run. These are necessary to reproduce the bugs for the software under test, shared among various crowdtesting platforms [19, 66].

3.3 Learning-based Ranking

Based on the dynamic testing context model, a learning-based ranking method is developed to derive the ranks of crowdworkers based on their probability of detecting bugs with respect to a particular testing context.

3.3.1 ***Feature Extraction.*** 26 features are extracted based on the process context and resource context for the learning model, as summarized in Table 2. Features 1-12 capture the ***activeness*** of a crowdworker. Previous work demonstrated the developer's recent activity has greater indicative effect on his/her future behavior than the activity happened long before [51, 69], so we extract the activeness-related features with varying time intervals. Features 13-19 capture the matching degree between a crowdworker's ***preference*** and the inadequate-tested aspects of the task. Features 20-26 capture the matching degree between the a crowdworker's ***expertise*** and the inadequate-tested aspects of the task. Note that, since the learning-based ranking method focuses on learning and matching the crowdworker's bug detection capability related to the descriptive terms of a task, we do not include the *device* dimension of resource context.

The first group of 12 features can be calculated directly based on the activeness attributes defined in the previous section. The second and third group of features are obtained in a similar way by examining the similarities. For brevity, we only present the details to produce the third group of features, i.e. 20-26.

Table 2: Features for learning to rank

Category	ID	Feature
Activeness indexing	1	LastBug
	2	LastReport
	3-7	NumBugs-8 hours, NumBugs-24 hours, NumBugs-1 week, NumBugs-2 week, NumBugs-all (i.e., in the past)
	8-12	NumReports-8 hours, NumReports-24 hours, NumReports-1 week, NumReports-2 week, NumReports-all (i.e., in the past)
Preference matching	13-14	Partial-ordered cosine similarity, partial-ordered euclidean similarity between worker's preference and test adequacy
	15-19	Partial-ordered jaccard similarity between worker's preference and test adequacy with the cutoff threshold of 0.0, *0.1, 0.2, 0.3, 0.4*
Expertise matching	20-21	Partial-ordered cosine similarity, partial-ordered euclidean similarity between worker's expertise and test adequacy
	22-26	Partial-ordered jaccard similarity between worker's expertise and test adequacy with the cutoff threshold of 0.0, *0.1, 0.2, 0.3, 0.4*

Previous work has proven extracting features from different perspectives can help improve the learning performance [9, 26, 40], so we extract the similarity-related features from different viewpoints. Cosine similarity, euclidean similarity, and jaccard similarity

are the three commonly-used similarity measurements and have proven to be efficient in previous researches [16, 17, 48, 52], therefore we utilize all these three similarities for feature extraction. In addition, a crowdworker might have extra expertise beyond the task's requirements (i.e., the test adequacy), to alleviate the potential bias introduced by the unrelated expertise, we define the partial-ordered similarity to constrain the similarity matching only on the descriptive terms within the task terms vector.

Partial-ordered cosine similarity (POCosSim) is calculated as the cosine similarity between test adequacy and a worker's expertise, with the similarity matching constraint only on terms appeared in task terms vector.

$$POCosSim = \frac{\sum x_i * y_i}{\sqrt{\sum x_i^2}\sqrt{\sum y_i^2}} \tag{4}$$

,where x_i is 1.0 - $TestAdeq(t_i)$, y_i is $ProbExp(w, t_i)$, and t_i is the ith descriptive term in task terms vector.

Partial-ordered euclidean similarity (POEucSim) is calculated as the euclidean similarity between test adequacy and a worker's expertise, with a minor modification on the distance calculation.

$$POEucSim = \begin{cases} \sqrt{\sum (x_i - y_i)^2}, & if \quad x_i >= y_i \\ 0, & if \quad x_i < y_i, \end{cases} \tag{5}$$

,where x_i and y_i is the same as in *POCosSim*.

Partial-ordered jaccard similarity with the cutoff threshold of θ (POJacSim) is calculated as the modified jaccard similarity between test adequacy and a worker's expertise based on the set of terms whose probabilistic values are larger than θ.

$$POJacSim = \frac{A \cap B}{A} \tag{6}$$

,where A is a set of descriptive terms whose (1.0 -$TestAdeq(t_i)$) is larger than θ, and B is a set of descriptive terms whose $ProbExp(w, t_i)$ is larger than θ.

3.3.2 ***Ranking***. We employ LambdaMART, which is the state-of-the-art learning to rank algorithm and reported as effective in many learning tasks of SE [58, 68].

Model training. For every task in the training dataset, at each *recPoint*, we first obtain the process context of the task and resource context for all crowdworkers, then extract the features for each crowdworker in Table 2. We treat the crowdworkers who submitted new bugs after *recPoint* (not duplicate with the submitted reports) as positive instances and label them as 1. As reported by existing work that unbalanced data could significantly affect the model performance [45, 46], to make our dataset balanced, we randomly sample an equal number of crowdworkers (who didn't submit bugs in the specific task) with the positive instances and label them as 0. The instances close to the boundary between the positive and negative regions can easily bring noise to the machine learner, therefore, to facilitate the generation of more effective learning model, we choose crowdworkers who are different from the positive instances [10, 40], i.e., to select those majority instances which are away from the boundary.

Ranking based on trained model. At the *recPoint*, we first obtain the process context and resource context for all crowdworkers, extract the features in Table 2, and apply the trained model to predict the bug detection probability of each crowdworker. We sort the crowdworkers based on the predicted probability in a descending order, and treat a ranked list of higher-ranked *recNum* crowdworkers (*recNum* is an input parameter since usually only a small set of crowdworkers is considered for recommendation) as the output of the learning-based ranking component, i.e., *initial ranking* in Figure 3.

3.4 Diversity-based Re-ranking

To produce less duplicate reports and improve the bug detection performance, as discussed in Section 2.3, we develop a diversity-based re-ranking method to adjust the initial ranking of crowdworkers to optimize the diversity among crowdworkers.

3.4.1 ***Diversity Measurement***. We first measure the diversity delta of a worker with respect to current re-ranked list of workers S (see Sec. 3.4.2 for details) in two dimensions, i.e., expertise diversity delta and device diversity delta.

Expertise diversity delta gives higher score to these workers who have most different expertise from the workers in the current re-ranked list.

$$ExpDiv(w, S) = \sum_{t_j} ProfExp(w, t_j) \times \prod_{w_k \in S} (1.0 - ProfExp(w_k, t_j)) \tag{7}$$

where the later part (i.e., $\prod$) estimates the extent to which t_j is tested by the workers on current re-ranked list.

Device diversity delta gives higher scores to these workers who can bring more new device's attributes (e.g., phone type, operating system, etc.) to those of the workers on current re-ranked list, so as to facilitate the exploration in new testing environment.

$$DevDiv(w, S) = (w's\ attributes) - \cup_{w_k \in S}(w_k's\ attributes) \tag{8}$$

where $w's\ attributes$ is a set of attributes of $w's\ device$, i.e., Samsung SN9009, Android 4.4.2, KOT49H.N9009, WIFI as in Table 1.

3.4.2 ***Re-ranking***. Suppose we have a ranked list of recommended workers (w_1 - w_{recNum}) produced by the learning-based ranking method, and an empty list of re-ranked list S, the re-ranking algorithm first moves w_1 to S, then executes the following steps iteratively (suppose current re-ranked list having r workers): ① Calculate $ExpDiv(w, S)$, $DevDiv(w, S)$ for the remaining workers in ranked list; ② Sort the workers respectively based on $ExpDiv(w, S)$ and $DevDiv(w, S)$ descending, and obtain the expertise index $expI(w)$ and device index $expI(w)$ (e.g., $expI(w) = 1$ for the worker with the largest $ExpDiv(w, S)$); ③ Obtain the combined diversity for each worker by $ExpI(w) + divRatio \times DevI(w)$ (where *divRatio* is an input parameter denoting the relative weight of device diversity compared with expertise diversity), and move the worker with the smallest value into S. The reason why we use *index* rather *the original value* for the combined diversity is to alleviate the influence of extreme value.

4 EXPERIMENT DESIGN

4.1 Research Questions

- **RQ1**: (Performance Evaluation) How effective is `iRec` for crowdworker recommendation?

For RQ1, we first present some general views of `iRec` for worker recommendation. To further demonstrate its advantages, we then

compare its performance with four state-of-the-art and commonly-used baseline methods (details are in Section 4.5).

- **RQ2**: (Context Sensitivity) To what degree iRec is sensitive to different categories of context?

The basis of this work is the characterization of the test context model (details are in Section 3.2). RQ2 examines the performance of iRec when removing different sub-category of the context, to understand the context sensitivity of recommendation.

- **RQ3**: (Diversity Gain) How much is the diversity gain by introducing the re-ranking method in recommendation?

Besides the learning-based ranking component, we further design a diversity-based re-ranking component to adjust the original ranking. RQ3 aims at examining its role in recommendation.

4.2 Dataset

We collected crowdtesting data from Baidu[5] crowdtesting platform, which is one of the largest industrial crowdtesting platform.

We collected the crowdtesting tasks that are closed between May. 1st 2017 and Nov. 1st 2017. In total, there are 636 mobile application testing tasks from various domains (details are in our website), involving 2,404 crowdworkers and 80,200 submitted reports. For each testing task, we collected its task-related information, all the submitted test reports and related information, e.g., submitter, device, etc. The minimum, average, and maximum number of reports (*and unique bugs*) per task are 20 (*3*), 126 (*24*), and 876 (*98*) respectively.

4.3 Experimental Setup

To simulate the usage of iRec in practice, we employ a commonly-used longitudinal data setup [44, 48, 53]. That is, all the 636 experimental tasks were sorted in the chronological order, and then divided into 21 equally sized folds with each fold having 30 tasks (the last fold has 36 tasks). We then employ the former *N-1* folds as the training dataset to train iRec and use the tasks in the *Nth* fold as the testing dataset to evaluate the performance of worker recommendation. We experiment N from 12 to 20 to ensure a relatively stable performance because a too small training dataset could not reach an effective model.

For each task in the testing dataset, at the triggered *recPoint* (see Section 3), we run iRec and other approaches to recommend crowdworkers. We experimented *recThres* from 3 to 12; and due to space limit, we only present the results with four representative *recThres* (i.e., 3, 5, 8, and 10) and leave others on our website. The size of the experimental dataset (i.e., number of total *recPoint*) under the four *recThres* are 676, 479, 345, and 278 respectively.

For the parameter *divRatio*, we tune the optimal value based on the training dataset. In detail, for every candidate parameter value (we experiment from 0.1 to 0.9), we obtain the *FirstHit* (see Section 4.4) of the recommendation result on the training set and calculate the median value. We treat the parameter value, under which the smallest median value is obtained, as the best one. The parameter *recNum* is tuned in the same way.

4.4 Evaluation Metrics

Given a crowdtesting task, we measure the performance of worker recommendation approach based on whether it can find the "right" workers who can detect bugs, and how early it can find the first one. Following previous studies, we use the commonly-used bug detection rate [13, 14, 51] for the evaluation.

Bug Detection Rate at k (BDR@k) is the percentage of unique bugs detected by the recommended k crowdworkers out of all unique bugs historically detected after the *recPoint* for the specific task. Since a smaller subset is preferred in crowdworker recommendation, we obtain *BDR@k* when k is 3, 5, 10, and 20.

Besides, as our in-process recommendation aims at shortening the non-yielding windows, we define another metric to intuitively measure how early the first bug can be detected.

FirstHit is the rank of the first occurrence, after *recPoint*, where a worker from the recommended list actually submitted a unique bug to the specific task.

To further demonstrate the superiority of our proposed approach, we perform one-tailed Mann Whitney U test [38] between our proposed iRec and other approaches. We include the Bonferroni correction [57] to counteract the impact of multiple hypothesis tests. Besides the *p-value* for signifying the significance of the test, we also present the *Cliff's delta* to demonstrate the effect size of the test. We use the commonly-used criteria to interpret the effectiveness levels, i.e., Large (0.474-1.0), Median (0.33-0.474), Small (0.147-0.33), and Negligible (-1, 0.147) (see details in [12]).

4.5 Ground Truth and Baselines

The **Ground Truth** of bug detection of a given task is obtained based on the historical crowdworkers who participated in the task after the *recPoint*. In detail, we first rank the crowdworkers based on their submitted reports in chronological order, then obtain the *BDR@k* and *FirstHit* based on this order.

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

MOCOM [51]: This is a multi-objective crowdworker recommendation approach by maximizing the bug detection probability of workers, the relevance with the test task, the diversity of workers, and minimizing the test cost.

ExReDiv [13]: This is a weight-based crowdworker recommendation approach that linearly combines experience strategy, relevance strategy, and diversity strategy.

MOOSE [14]: This is a multi-objective crowdworker recommendation, which can maximize the coverage of test requirement, maximize the test experience of workers, and minimize the cost.

Cocoon [60]: This crowdworker recommendation approach is designed to maximize the testing quality (measured in worker's historical submitted bugs) under the test coverage constraint.

For each baseline, we conduct worker recommendation before the task begins; then at each *recPoint*, we first obtain the set of worker who have submitted reports in the specific task (denoted as white list workers), and use the recommended workers minus the white list workers as the final set of recommended workers. Note that, the reason why take out the white list workers is because 99% crowdworkers only participated one time in a crowdtesting task in our experimental dataset; and without the white list, the performance would be worse.

[5] test.baidu.com

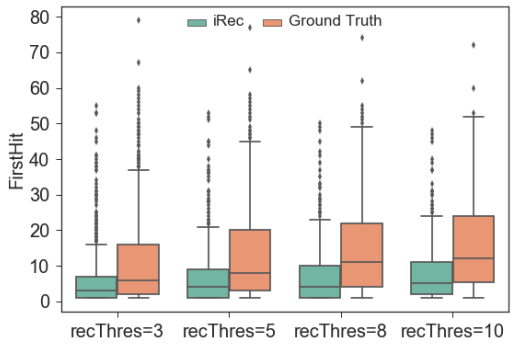


Figure 4: Performance of iRec for FirstHit

5 RESULTS AND ANALYSIS

5.1 Answering RQ1: Performance Evaluation

Figure 4 demonstrates the *FirstHit* of worker recommendation under four representative *recThres* (i.e., *recThres*-sized non-yielding window is observed in Section 3), i.e., 3, 5, 8, and 10. We can easily see that for all four *recThres*, *FirstHit* of iRec is significantly (p-value is 0.00) and substantially (Cliff's delta is 0.25-0.39) better than current practice of crowdtesting. When *recThres* is 5, the median *FirstHit* of iRec and *Ground Truth* are respectively 4 and 8, indicating our proposed approach can shorten the non-yielding window by 50%. For other application scenarios (i.e., *recThres* is 3, 8, and 10), iRec can shorten the non-yielding window by 50% to 58%.

Figure 5 demonstrates the *BDR@k* of worker recommendation under four representative *recThres*. iRec significantly (p-value is 0.00) and substantially (Cliff's delta is 0.24-0.39) outperforms current practice of crowdtesting for BDR@k (k is 3, 5, 10, and 20). When *recThres* is 5, a median of 50% remaining bugs can be detected with the first 10 recommended crowdworkers by our proposed iRec, with 400% improvement compared with current practice of crowdtesting (50% vs. 10%). Besides, a median of 78% remaining bugs can be detected with the first 20 recommended crowdworkers by iRec, with 160% improvement compared with current practice (78% vs. 30%). This again indicates the effectiveness of our approach not only for the power in finding the first "right" workers, but also in terms of the bug detection with the set of recommended workers.

We also notice that for a larger *recThres*, the advantage of iRec over current practice is larger. In detail, when *recThres* is 3, iRec can improve the current practice by 87% (75% vs. 40%) for *BDR@20*, and when *recThres* is 8, the improvement is 460% (80% vs. 14%). This holds true for other metrics. A larger *recThres* might indicate the task is getting tough because no new bugs are reported in quite a long time, and our proposed iRec can help the task get out of the dilemma with new bugs submitted very soon.

Furthermore, for the *recPoint* with larger *FirstHit of Ground Truth*, our proposed approach can shorten the non-yielding window in a larger extent (due to space limit, see the figure on our website). For example, for the *recPoint* whose *FirstHit of Ground Truth* is larger than 3 (*recThres* is 5), iRec can shorten the non-yielding window by 64% on median (5 vs. 14), while the improvement is 50% (4 vs. 8) in the whole dataset. This further indicates the effectiveness of our approach since for *recPoint* with a larger *FirstHit of Ground Truth*, it is in higher demand for an efficient worker recommendation so that the "right" worker can come soon.

In the following paper, we use the experimental setting when *recThres* is 5 for further analysis and comparison due to space limit.

Comparison with Baselines. Figure 6 demonstrates the comparison results with four baselines. Overall, our proposed iRec significantly (p-value is 0.00) and substantially (Cliff's delta is 0.16-0.23) outperforms the four baselines in terms of *FirstHit* and *BDR@k* (k is 3, 5, 10, and 20). Specifically, iRec can improve the best baseline *MOCOM* by 60% (4 vs. 10) for median *FirstHit*; and the improvement is infinite for median *BDR@k* (e.g., 78% vs. 0 for BDR@20). This is because all the baselines are designed to recommend a set of workers before the task begins and don't consider various context information of the crowdtesting process. Besides, the aforementioned baseline approaches do not explicitly consider the activeness of crowdworkers which is another cause of performance decline. Furthermore, the baselines' performance are similar to each other which is also due to their limitations of lacking contextual details in one-time worker recommendation

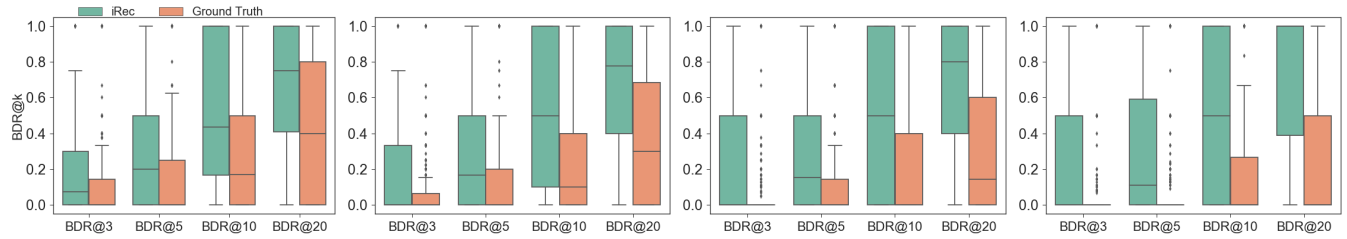
5.2 Answering RQ2: Context Sensitivity

Figure 7 shows the comparison results between iRec and its six variants. Specifically, *noAct*, *noPref*, *noExp*, and *noDev* are different variants of iRec without activeness, preference, expertise, and device context respectively. Because process context cannot be removed, *noProc* denotes using the process context at the beginning of a task. We additionally present *noRsr* which denotes using the resource context at the beginning of the task to further demonstrate the necessity of precise context modeling.

We can see that without any type of the resource context (i.e., *noAct*, *noPref*, *noExp*, and *noDev*), the recommendation performance would undergo a decline in both *FirstHit* and *BDR@k*. Without activeness-related context, the *FirstHit* of the recommended workers undergoes a largest variation, i.e., the most sensitive context for recommendation. This might be because this dimension of features is the only one for capturing time-related information, and without them, the model would lack important clues for the crowdworkers' time-series behavior. Preference-related context exerts a slightly larger influence on the recommendation performance than expertise-related context, although they are modeled similarly. This might because many crowdworkers submitted reports but didn't report bugs, so preference-related context is more informative than experience-related context, thus we can build more effective learning model. The lower performance of *noProc* and *noRsr* compared with iRec further indicates the necessity of the precise context modeling.

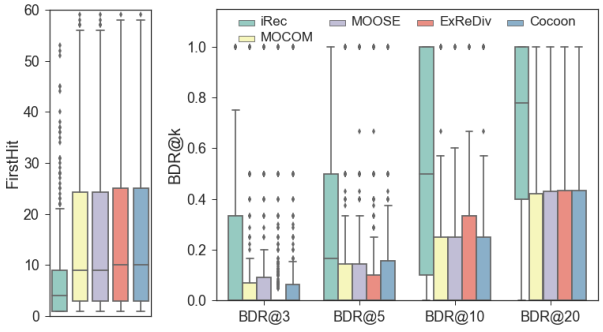
5.3 Answering RQ3: Diversity Gain

Table 3 first demonstrates the average performance of iRec and iRec *without re-rank*, followed by the distribution of performance increase and decrease of iRec compared with iRec *without re-rank* in all *recPoint*. We can see that with the re-ranking component, the average performance can be improved by 12% to 19%. Specifically, the re-ranking can increase the *BDR@10* in 25% cases, and decrease it in 15% cases. This is because there are large amount of duplicate bugs and increasing the diversity of recommended workers



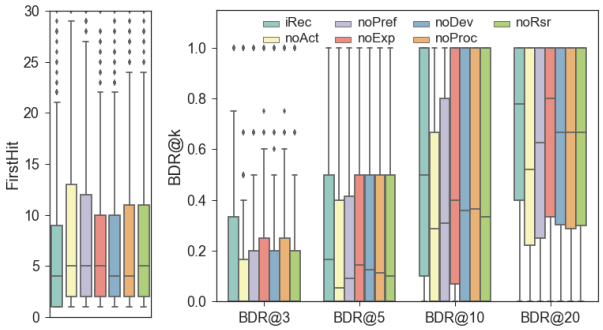
(a) BDR@k for *recThres*=3 (b) BDR@k for *recThres*=5 (c) BDR@k for *recThres*=8 (d) BDR@k for *recThres*=10

Figure 5: Performance of iRec for BDR@k



(a) FirstHit (b) BDR@k

Figure 6: Performance comparison with baselines



(a) FirstHit (b) BDR@k

Figure 7: Context sensitivity

can help decrease the duplicate bugs so as to increase the unique bugs. Furthermore, we can observe that there are more points with performance increase than those with decrease for *BDR@k* with larger *k*. This makes sense because if a crowdworker contributes less to the diversity, he/she would be moved backward so that more unique bugs can be detected earlier; and the larger of examined *k*, the larger possibility for duplicate bugs in terms of the original list, and more room for improvement.

Although the average value for all metrics are increased with re-rank, we admit that the re-ranking component can not always improve the performance. This might be because sometimes the workers ranked earlier are not always those who can detect bugs, and when the re-ranking moves back the similar workers who can actually detect bugs, the bug detection performance would decline.

Table 3: Role of re-ranking

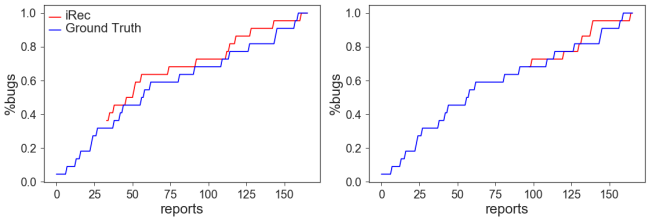
	FirstHit	BDR@3	BDR@5	BDR@10	BDR@20
Average performance					
iRec	7.21	21%	32%	48%	67%
iRec without re-rank	8.35	18%	26%	41%	59%
improvement	13.5%	17.1%	19.6%	15.9%	12.3%
Recommending points					
performance increase	39%	15%	20%	25%	26%
performance decrease	27%	9%	12%	15%	12%

Future work would design more effective re-ranking algorithm to tackle the negative effect on the recommendation performance.

6 DISCUSSION

6.1 Benefits of In-process Recommendation

In-process worker recommendation has great potential to facilitate talent identification and utilization for complex, intelligence-intensive tasks. As presented in the previous sections, the proposed iRec established the crowdtesting context model at a dynamic, finer granularity, and constructed two methods to rank and re-rank the most suitable workers based on dynamic testing progress. In this section, we discuss with more details about why practitioners should care about such kind of in-process crowdworker recommendation.



(a) at report#32 (b) at report#86

Figure 8: Illustrative examples of iRec

We utilize illustrative examples to demonstrate the benefits of the application of iRec. Figure 8 demonstrates two typical bug detection curve using iRec for two *recPoint* of the task in Figure 1. We can easily see that with iRec, not only the current non-yielding window can be shortened, but also the following bug detection efficiency can be improved with the recommended set of workers. In detail, in Figure 8a, we can clearly see that with the recommended workers, the bug detection curve can rise quickly, i.e., with equal

number of workers, more bugs can be detected. Also note that, in real-world application of iRec, the in-process recommendation can be conducted dynamically following the new bug detection curve so that the bug detection performance can be further improved. In Figure 8b, although the bug detection curve can not always dominate the current practice, the first "right" worker can be found earlier than current practice. Similarly, with the dynamic recommendation, the current practice of bug detection can be improved.

Table 4: Reduced cost with iRec

	recThres=3	*recThres*=5	*recThres*=8	*recThres*=10
1st-quarter	4.8%	4.2%	2.7%	2.8%
median	12.1%	9.8%	8.6%	8.1%
3rd-quarter	21.3%	18.6%	16.7%	16.4%

Based on the metrics in Section 4.4 that are applied for single *recPoint*, we further measure the ***reduced cost*** for each crowdtesting task if equipped with iRec for in-process crowdworker recommendation. It is measured based on the number of reduced report, i.e., the difference of *FirstHit* value between iRec and *Ground Truth*, following previous work [51, 53]. For a crowdtesting task with multiple *recPoint*, we simply add up the reduced cost of each *recPoint*. As shown in Table 4, a median of 8% to 12% cost can be reduced, indicating about 10% cost can be saved if equipped with our proposed approach for in-process crowdworker recommendation. Note that, this figure is calculated by simply summing up the reduced cost of single *recPoint* based on the offline evaluation scenario adopted in this work. However, as shown in Figure 8, in real-world practice, the recommendation can be conducted based on the bug arrival curve after the prior recommendation; and the reduced cost should be further improved. Therefore, crowdtesting managers could benefit tremendously from actionable insights offered by in-process recommendation systems like iRec.

6.2 Implication of In-process Recommendation

Nevertheless, in-process crowdworker recommendation is a complicated, systematic, human-centered problem. By nature, it is more difficult to model than the one-time crowdworker recommendation at the beginning of the task. This is because the non-yielding windows are scattered in the crowdtesting process. Although the overall non-yielding reports are in quite large number, some of the non-yielding windows are not long enough to apply the recommendation approach or let the recommendation approach work efficiently. Our observation reveals that an average of 39% cost is wasted on these long-sized non-yielding windows (see Section 2.2), but the reduced cost by our approach is only about 10% which is far less than the ideal condition. From one point of view, this is because the front part of the non-yielding window (i.e., *recPoint* in Section 3) could not be saved because it is needed for determining whether to conduct the worker recommendation. And from another point of view, there is still room for performance improvement.

On the other hand, the true effect of in-process recommendation depends on the potential delays due to interactions between the testing manager, the platform, and the recommended workers. The longer the delays are, the less the benefit can take effect. It is critical for crowdtesting platforms, when deploying in-process recommendation systems, to consider how to better streamline the recommendation communication and confirmation functions, in order to minimize the potential delays in bridging the best workers with the tasks under test. For example, the platform may employ instant synchronous messaging service for recommendation communication, and innovate rewarding system to attract more in-process recruitment. More human factor-centered research is needed along this direction to explore systematic approaches for facilitating the adoption of in-process recommendation systems.

6.3 Threats to Validity

First, following existing work [51, 53], we use the number of crowdtesting reports as the amount of cost when measuring the reduced cost. As discussed in [53], the reduced cost is equal with or positively correlated with the number of reduced reports for all the three typical payout schemas.

Second, the recommendation is triggered by the non-yielding window, which is obtained based on report's attributes. In crowdtesting process, each report would be inspected and triaged with these two attributes (i.e., bug label and duplicate label) so as to better manage the reported bugs and facilitate bug fixing [18, 67]. This can be done manually or with automatic tool support (e.g., [48, 49]). Therefore, we assume our designed methods can be easily adopted in the crowdtesting platform.

Third, we evaluate iRec in terms of each recommending point, and sum up the single performance as the overall reduced cost. This is limited by the offline evaluation, which is quite common choice of previous worker recommendation approaches in SE [8, 23, 27, 44, 61]. In real-world practice, iRec can be applied dynamically based on the new bug arrival curve formed by the prior recommended crowdworkers. We assume when applied online, the reduction of cost should be larger because the later recommendation can be based on the results of prior recommendation which is proven to be efficient compared with current practice.

Fourth, for the generalizability of our approach, a recent systematic review [66] has shown current crowdtesting services are dominated by functional, usability, and security test of mobile applications. The dataset used in our study is largely representative of this trend, with 632 functional and usability test tasks spanning across 12 application domains (e.g., music, sport). The proposed approach is based on dynamically constructing the testing context model using NLP techniques and learning-based ranking, which is independent of different testing types. We believe that the proposed approach is generally applicable to supporting other testing types such as security and performance testing, since more sophisticated skillsets reflecting these specialty testing may be implicitly represented by corresponding descriptive terms learned in the dynamic context. Therefore, the learning and ranking components will not be affected and can be reused. Further verification on other testing types or scenarios is planned as our future work.

7 RELATED WORK

Crowdtesting has been applied to facilitate many testing tasks, e.g., test case generation [11], usability testing [22], software performance analysis [37], software bug detection and reproduction [21]. There were dozens of approaches focusing on the new encountered problems in crowdtesting, e.g., crowdtesting reports prioritization

[16, 17, 28], reports summarization [24], reports classification [48–50, 52], automatic report generation [30], crowdworker recommendation [13, 14, 51, 60], crowdtesting management [53], etc.

There were many lines of related studies for recommending workers for various software engineering tasks, such as bug triage [6, 7, 27, 34, 39, 44, 54, 55, 59, 61, 65], code reviewer recommendation [15, 23, 64], expert recommendation [8, 32], developer recommendation for crowdsourced software development [29, 33, 62, 63], worker recommendation for general crowdsourcing tasks [5, 31, 42], etc. The aforementioned studies either recommended one worker or assumed the recommended set of workers are independent of each other, which is not applicable for testing activity.

Several studies explored worker recommendation for crowdtesting tasks by modeling the workers' testing environment [51, 60], experience [13, 60], capability [51], expertise with the task [13, 14, 51], etc. However, these existing worker recommendation solutions only apply at the beginning of the task, and do not consider the dynamic nature of crowdtesting process.

The need for context in software engineering is officially proposed by Prof. Gail Murphy in 2018 [35, 36], and she stated that the lack of context in software engineering tools would limit the effectiveness of software development. Context-related information has been utilized in various software development activities, e.g., code recommendation [20], software documentation [4], static analysis [25, 47], etc. This work provides new insights about how to model and utilize the context information in open environment.

8 CONCLUSIONS

Open software development processes, e.g. crowdtesting, are highly dynamic, distributed, and concurrent. Existing worker recommendation studies largely overlooked the dynamic and progressive nature of crowdtesting process. This paper proposed a context-aware in-process crowdworker recommendation approach, `iRec`, to bridge this gap. Built on top of a fine-grained context model, `iRec` can dynamically learn a ranked list of capable and diverse workers from historical and ongoing contextual information at any specific point of crowdtesting process. The evaluation results demonstrate its potential benefits in shortening the non-yielding window, improving bug detection efficiency, and reducing testing cost.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under grant No.2018YFB1403400, the National Natural Science Foundation of China under grant No.61602450, No.61432001. We would like to thank the testers in Baidu for their extensive efforts in supporting this work.

REFERENCES

[1] 2019. https://www.topcoder.com/.
[2] 2019. https://www.applause.com/.
[3] 2019. https://www.testbird.com/.
[4] Emad Aghajani. 2018. Context-Aware Software Documentation. In *2018 IEEE International Conference on Software Maintenance and Evolution, ICSME 2018, Madrid, Spain, September 23-29, 2018*. 727–731.
[5] Eiman Aldahari, Vivek Shandilya, and Sajjan G. Shiva. 2018. Crowdsourcing Multi-Objective Recommendation System. In *Companion of the The Web Conference 2018 on The Web Conference 2018, WWW 2018, Lyon , France, April 23-27, 2018*. 1371–1379.
[6] John Anvik, Lyndon Hiew, and Gail C Murphy. 2006. Who should fix this bug?. In *ICSE'06*. 361–370.
[7] Pamela Bhattacharya and Iulian Neamtiu. 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *ICSM'10*. 1–10.
[8] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is going to mentor newcomers in open source projects?. In *FSE'12*. 44.
[9] Zherui Cao, Yuan Tian, Tien-Duy B. Le, and David Lo. 2018. Rule-based specification mining leveraging learning to rank. *Autom. Softw. Eng.* 25, 3 (2018), 501–530.
[10] Di Chen, Wei Fu, Rahul Krishna, and Tim Menzies. 2018. Applications of psychological science for actionable analytics. In *Proceedings of the 2018 ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/SIGSOFT*. 456–467.
[11] Ning Chen and Sunghun Kim. 2012. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *ASE'12*. 140–149.
[12] N. Cliff. 2014. *Ordinal methods for behavioral data analysis*. Psychology Press.
[13] Qiang Cui, Junjie Wang, Guowei Yang, Miao Xie, Qing Wang, and Mingshu Li. 2017. Who Should Be Selected to Perform a Task in Crowdsourced Testing?. In *COMPSAC'17*. 75–84.
[14] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-Objective Crowd Worker Selection in Crowdsourced Testing. In *SEKE'17*. 218–223.
[15] Yuanrui Fan, Xin Xia, David Lo, and Shanping Li. 2018. Early prediction of merged code changes to prioritize reviewing tasks. *Empirical Software Engineering* 23, 6 (2018), 3346–3393.
[16] Yang Feng, Zhenyu Chen, James A Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing. In *FSE'15*. 225–236.
[17] Yang Feng, James A Jones, Zhenyu Chen, and Chunrong Fang. 2016. Multi-objective test report prioritization using image understanding. In *ASE'16*. 202–213.
[18] Ruizhi Gao, Yabin Wang, Yang Feng, Zhenyu Chen, and W Eric Wong. 2018. Successes, challenges, and rethinking–an industrial investigation on crowdsourced mobile application testing. *Empirical Software Engineering* (2018), 1–25.
[19] Ruizhi Gao, Yabin Wang, Yang Feng, Zhenyu Chen, and W. Eric Wong. 2019. Successes, challenges, and rethinking - an industrial investigation on crowdsourced mobile application testing. *Empirical Software Engineering* 24, 2 (2019), 537–561.
[20] Marko Gasparic, Gail C. Murphy, and Francesco Ricci. 2017. A context model for IDE-based recommendation systems. *Journal of Systems and Software* 128 (2017), 200–219.
[21] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 88–99.
[22] Victor HM Gomide, Pedro A Valle, José O Ferreira, José RG Barbosa, Adson F Da Rocha, and TMGdA Barbosa. 2014. Affective crowdsourcing applied to usability testing. *International Journal of Computer Scienceand Information Technologies* 5, 1 (2014), 575–579.
[23] Christoph Hannebauer, Michael Patalas, Sebastian Stünkel, and Volker Gruhn. 2016. Automatically Recommending Code Reviewers Based on Their Expertise: An Empirical Comparison. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE 2016)*. 99–110.
[24] Rui Hao, Yang Feng, James Jones, Yuying Li, and Zhenyu Chen. 2019. CTRAS: Crowdsourced test report aggregation and summarization. In *ICSE'2019*. 921–932.
[25] Kihong Heo, Hakjoo Oh, and Hongseok Yang. 2019. Resource-aware program analysis via online abstraction coarsening. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 94–104.
[26] Qiao Huang, Emad Shihab, Xin Xia, David Lo, and Shanping Li. 2018. Identifying self-admitted technical debt in open source projects using text mining. *Empirical Software Engineering* 23, 1 (2018), 418–451.
[27] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *FSE'09*. 111–120.
[28] He Jiang, Xin Chen, Tieke He, Zhenyu Chen, and Xiaochen Li. 2018. Fuzzy Clustering of Crowdsourced Test Reports for Apps. *ACM Transactions on Internet Technology* 18, 2 (2018), 18.
[29] Muhammad Rezaul Karim, Ye Yang, David Messinger, and Guenther Ruhe. 2018. Learn or Earn? - Intelligent Task Recommendation for Competitive Crowdsourced Software Development. In *51st Hawaii International Conference on System Sciences, HICSS 2018, Hilton Waikoloa Village, Hawaii, USA, January 3-6, 2018*. 1–10.
[30] Di Liu, Xiaofang Zhang, Yang Feng, and James A. Jones. 2018. Generating descriptions for screenshots to assist crowdsourced testing. In *25th International Conference on Software Analysis, Evolution and Reengineering, SANER*. 492–496.
[31] Zheng Liu and Lei Chen. 2017. Worker Recommendation for Crowdsourced Q&A Services: A Triple-Factor Aware Approach. *PVLDB* 11, 3 (2017), 380–392.
[32] David Ma, David Schuler, Thomas Zimmermann, and Jonathan Sillito. 2009. Expert recommendation with usage expertise. In *ICSM'09*. 535–538.

[33] Ke Mao, Ye Yang, Qing Wang, Yue Jia, and Mark Harman. 2015. Developer recommendation for crowdsourced software development tasks. In *SOSE'15*. 347–356.

[34] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In *MSR'09*. 131–140.

[35] Gail C. Murphy. 2018. The Need for Context in Software Engineering (IEEE CS Harlan Mills Award Keynote). In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE 2018)*. 5–5.

[36] Gail C. Murphy. 2019. Beyond integrated development environments: adding *context* to software development. In *Proceedings of the 41st International Conference on Software Engineering: New Ideas and Emerging Results, ICSE (NIER) 2019, Montreal, QC, Canada, May 29-31, 2019*. 73–76.

[37] R. Musson, J. Richards, D. Fisher, C. Bird, B. Bussone, and S. Ganguly. 2013. Leveraging the Crowd: How 48,000 Users Helped Improve Lync Performance. *IEEE Software* 30, 4 (2013), 38–45.

[38] Nadim Nachar et al. 2008. The Mann-Whitney U: A test for assessing whether two independent samples come from the same distribution. *Tutorials in quantitative Methods for Psychology* 4, 1 (2008), 13–20.

[39] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. 2013. Bug report assignee recommendation using activity profiles. In *MSR'13*. 22–30.

[40] Haoran Niu, Iman Keivanloo, and Ying Zou. 2017. Learning to rank code examples for code search engines. *Empirical Software Engineering* 22, 1 (2017), 259–291.

[41] Thomas Oberlin and Rangasami L. Kashyap. 1973. Bayes Decision Rules Based on Objective Priors. *IEEE Trans. Systems, Man, and Cybernetics* 3, 4 (1973), 359–364.

[42] Mejdl S. Safran and Dunren Che. 2019. Efficient Learning-Based Recommendation Algorithms for Top-*N* Tasks and Top-*N* Workers in Large-Scale Crowdsourcing Systems. *ACM Trans. Inf. Syst.* 37, 1 (2019), 2:1–2:46.

[43] Gerard Salton and Michael McGill. 1984. *Introduction to Modern Information Retrieval*. McGraw-Hill Book Company.

[44] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M Al-Kofahi, and Tien N Nguyen. 2011. Fuzzy set and cache-based approach for bug triaging. In *FSE'11*. 365–375.

[45] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online Defect Prediction for Imbalanced Data. In *ICSE'15*. 99–108.

[46] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *TSE'16* 43 (2016), 1–18.

[47] Junjie Wang, Bihuan Chen, Lei Wei, and Yang Liu. 2019. Superion: grammar-aware greybox fuzzing. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 724–735.

[48] Junjie Wang, Qiang Cui, Qing Wang, and Song Wang. 2016. Towards effectively test report classification to assist crowdsourced testing. In *ESEM'16*. 6.

[49] Junjie Wang, Qiang Cui, Song Wang, and Qing Wang. 2017. Domain adaptation for test report classification in crowdsourced testing. In *ICSE-SEIP'17*. 83–92.

[50] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information & Software Technology* 110 (2019), 139–155.

[51] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2019. Characterizing Crowds to Better Optimize Worker Recommendation in Crowdsourced Testing. *IEEE Transactions on Software Engineering* (2019).

[52] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *ASE'16*. 190–201.

[53] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. iSENSE: Completion-Aware Crowdtesting Management. In *ICSE'2019*. 932–943.

[54] Song Wang, Wen Zhang, and Qing Wang. 2014. FixerCache: Unsupervised caching active developers for diverse bug triage. In *ESEM'14*. 25.

[55] Song Wang, Wen Zhang, Ye Yang, and Qing Wang. 2013. DevNet: exploring developer collaboration in heterogeneous networks of bug repositories. In *ESEM'13*. 193–202.

[56] Lili Wei, Yepang Liu, and Shing-Chi Cheung. 2019. Pivot: learning API-device correlations to facilitate Android compatibility issue detection. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*. 878–888.

[57] Eric W Weisstein. 2004. Bonferroni correction. (2004).

[58] Qiang Wu, Christopher J. C. Burges, Krysta M. Svore, and Jianfeng Gao. 2010. Adapting boosting for information retrieval measures. *Information Retrieval* 13, 3 (01 Jun 2010), 254–270.

[59] Xin Xia, David Lo, Ying Ding, Jafar M. Al-Kofahi, Tien N. Nguyen, and Xinyu Wang. 2017. Improving Automated Bug Triaging with Specialized Topic Model. *IEEE Trans. Software Eng.* 43, 3 (2017), 272–297.

[60] Miao Xie, Qing Wang, Guowei Yang, and Mingshu Li. 2017. COCOON: Crowdsourced Testing Quality Maximization Under Context Coverage Constraint. In *ISSRE'17*. 316–327.

[61] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. Developer prioritization in bug repositories. In *ICSE'12*. 25–35.

[62] Hui Yang, Xiaobing Sun, Bin Li, and Yucong Duan. 2016. DR_PSF: Enhancing developer recommendation by leveraging personalized source-code files. In *COMPSAC'16*, Vol. 1. 239–244.

[63] Ye Yang, Muhammad Rezaul Karim, Razieh Saremi, and Guenther Ruhe. 2016. Who Should Take This Task?: Dynamic Decision Support for Crowd Workers. In *ESEM'16*. 8.

[64] M. B. Zanjani, H. Kagdi, and C. Bird. 2016. Automatically Recommending Peer Reviewers in Modern Code Review. *IEEE Transactions on Software Engineering* 42, 6 (2016), 530–543.

[65] Wen Zhang, Song Wang, Ye Yang, and Qing Wang. 2013. Heterogeneous network analysis of developer contribution in bug repositories. In *CSC'13*. 98–105.

[66] Xiaofang Zhang, Yang Feng, Di Liu, Zhenyu Chen, and Baowen Xu. 2018. Research Progress of Crowdsourced Software Testing. *Journal of Software* 29(1) (2018), 69–88.

[67] Xiaofang Zhang, Yang Feng, Di Liu, Zhenyu Chen, and Baowen Xu. 2018. Research Progress of Crowdsourced Software Testing. *Journal of Software* 29(1) (2018), 69–88.

[68] Guoliang Zhao, Daniel Alencar da Costa, and Ying Zou. 2019. Improving the pull requests review process using learning-to-rank algorithms. *Empirical Software Engineering* 24, 4 (2019), 2140–2170.

[69] M. Zhou and A. Mockus. 2012. What make long term contributors: Willingness and opportunity in OSS community. In *ICSE'12*. 518–528.