Context- and Fairness-Aware In-Process Crowdworker Recommendation

JUNJIE WANG, Laboratory for Internet Software Technologies, State Key Laboratory of Computer Sciences, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences
YE YANG, School of Systems and Enterprises, Stevens Institute of Technology
SONG WANG, Lassonde School of Engineering, York University
JUN HU, Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences
QING WANG, Laboratory for Internet Software Technologies, State Key Laboratory of Computer Sciences, Science & Technology on Integrated Information System Laboratory, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences

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 improve bug detection efficiency, i.e., detect more bugs with fewer workers. However, there are a couple of limitations in existing work. First, these studies mainly focus on one-time recommendations based on expertise matching at the beginning of a new task. Second, the recommendation results suffer from severe popularity bias, i.e., highly experienced workers are recommended in almost all the tasks, while less experienced workers rarely get recommended. This article argues the need for context- and fairness-aware in-process crowdworker recommendation in order to address these limitations. 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. Besides, motivated by the popularity bias in existing crowdworker recommendation approach, this study also aims at alleviating the unfairness in recommendations.

Driven by these observations, this article proposes a context- and fairness-aware in-process crowdworker recommendation approach, iRec2.0, to detect more bugs earlier, shorten the non-yielding windows, and alleviate the unfairness in recommendations. It consists of three main components: (1) the modeling of dynamic

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. 62072442, No. 62002348, and Youth Innovation Promotion Association Chinese Academy of Sciences.

Authors' addresses: J. Wang, Laboratory for Internet Software Technologies, State Key Laboratory of Computer Sciences, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing 101408, China; email: junjie@iscas.ac.cn; Y. Yang, School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ 07030; email: ye.yang@stevens.edu; S. Wang, Lassonde School of Engineering, York University, ON M3J 1P3, Canada; email: wangsong@eecs.yorku.ca; J. Hu, Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing 101408, China; email: hujun@iscas.ac.cn; Q. Wang (corresponding author), Laboratory for Internet Software Technologies, State Key Laboratory of Computer Sciences, Science & Technology on Integrated Information System Laboratory, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing 101408, China; email: wq@iscas.ac.cn.

1049-331X/2022/03-ART35 $15.00
https://doi.org/10.1145/3487571

testing context, (2) the learning-based ranking component, and (3) the multi-objective optimization-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 iRec2.0 in improving the cost-effectiveness of crowdtesting by saving the cost, shortening the testing process, and alleviating the unfairness among workers. In detail, iRec2.0 could shorten the non-yielding window by a median of 50%–66% in different application scenarios, and consequently have potential of saving testing cost by a median of 8%–12%. Meanwhile, the recommendation frequency of the crowdworker drop from 34%–60% to 5%–26% under different scenarios, indicating its potential in alleviating the unfairness among crowdworkers.

CCS Concepts: • **Software and its engineering → Software creation and management;**

Additional Key Words and Phrases: Crowdsourced testing, worker recommendation, multi-objective optimization, fair recommendation

ACM Reference format:
Junjie Wang, Ye Yang, Song Wang, Jun Hu, and Qing Wang. 2022. Context- and Fairness-Aware In-Process Crowdworker Recommendation. *ACM Trans. Softw. Eng. Methodol.* 31, 3, Article 35 (March 2022), 31 pages.
https://doi.org/10.1145/3487571

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 [2–4]. 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 [3].

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 [32, 40, 77, 95]. One of the most essential functions is to identify appropriate workers for a particular testing task [22, 23, 75, 88]. 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 [75, 88], experience [22, 88], capability [75], or expertise with the task [22, 23, 75], and so on. 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 the testing process by recommending appropriate crowdworkers in a dynamic manner, so that the non-yielding windows could be shortened.

Our previous work led to the development of a context-aware in-process crowdworker recommendation approach (named iRec) [78]. iRec can dynamically recommend a diverse set

of capable crowdworkers based on various contextual information of the crowdtesting process, aiming at shortening the non-yielding window, and improving bug detection efficiency. It designs a learning-based ranking component to learn the probability of crowdworkers being able to detect bugs within specific context, and a diversity-based re-ranking component to adjust the ranked list of recommended workers based on the diversity measurement to potentially reduce duplicate bugs. The evaluation result shows that iRec is able to shorten the non-yielding window by a median of 50%–58% in different scenarios.

Nonetheless, iRec has one major limitation associated with most **recommender systems** (**RS**), i.e., popularity bias in recommendation results. Many RS suffer from popularity bias in their output, which refers that popular items are recommended frequently and less popular ones rarely, if at all [5, 7, 15, 62]. Specifically, our pilot study with iRec (see Section 2.4) shows that some highly experienced crowdworkers could be recommended for almost all the tasks, while some less experienced workers rarely get recommended. Such popularity bias not only leads to recommendation results biased towards experienced workers, but lacks of support for less experienced workers. Existing work shows that in software crowdsourcing market, the majority long-tail workers are less-experienced, learning-oriented workers [47, 92]. In this study, we argue that such less-experienced workers are often desirable recommendations, not only because they are thirsty for recommendation, but the lack of consideration or accommodation in RS would lead to unfair recommendations, potential discouraging worker motivation, and hindering the prosperous of the platform.

To address this limitation, this article proposes an extension to iRec called iRec2.0, to alleviate the unfairness. This extension employs a multi-objective optimization-based re-ranking component, which can jointly maximize the crowdworkers' bug detection probability, the expertise diversity and device diversity of crowdworkers so as to produce less duplicate bugs, and meanwhile, minimize the recommendation frequency difference among crowdworkers to alleviate the unfairness. iRec2.0 extends and reinforces iRec, thus offers better crowdworker recommendations regarding both bug detection performance and recommendation fairness.

The rest of the article is structured as follows. We first present the background and motivation of this study. This material is driven by the preliminary empirical analysis and observations on an industry crowdtesting dataset. We then introduce iRec2.0 which consists of three main components: testing context modeling, learning-based ranking, and multi-objective optimization-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 features, the learning-based ranking component learns the probability of crowdworkers being able to detect bugs within a specific context. Third, the multi-objective optimization-based re-ranking component generates the re-ranking list of recommended workers by jointly maximizing the bug detection probability of workers, the expertise and device diversity of workers, and minimizing the recommendation frequency difference of crowdworkers.

iRec2.0 is evaluated on 636 crowdtesting tasks (involving 2,404 crowdworkers and 80,200 reports) from one of the largest crowdtesting platforms. Results show that iRec2.0 could shorten the non-yielding window by a median of 50%–66% in different application scenarios, and consequently have potential of saving testing cost by a median of 8%–12%. Meanwhile, the recommendation frequency of crowdworker drop from 34%–60% to 5%–26% under different scenarios, indicating its potential in alleviating the unfairness among crowdworkers. It significantly outperforms four commonly-used and state-of-the-art baseline approaches, and outperforms the original iRec in bug detection performance and the recommendation fairness.

This article 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 first empirical investigation of the popularity bias and unfairness in crowdworker recommendation to motivate this work and our approach.
— 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 multi-objective optimization-based re-ranking method to generate the re-ranked list to reduce duplicate bugs and alleviate the unfairness among workers.
— The evaluation of the proposed approach on 636 crowdtesting tasks (involving 2,404 crowdworkers and 80,200 reports) from one of the largest crowdsourced testing platforms, with affirmative results.[1]

The article extends a prior publication (presented at ICSE 2020 [78][2]) as follows:

— The empirical investigation of the popularity bias and unfairness in crowdworker recommendation to illustrate the limitation of prior work and motivate this study (Section 2.4).
— The development of multi-objective optimization-based re-ranking component, which generates the re-ranked list of crowdworkers to reduce duplicate bugs and alleviate the unfairness (Section 3.4).
— The experimental evaluation of the newly-proposed iRec2.0 to prove its effectiveness in terms of bug detection performance and recommendation fairness (Section 5).
— The discussion of objectivity and fairness in crowdworker recommendation to motivate future research in this field (Section 6.3).

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 [77, 95], e.g., pay by report. As discussed in previous work [75, 77], 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;[3] and duplicate label indicates with which the report is duplicate. Note that, in the following article, we refer to "bug report" (also

[1] https://github.com/wangjunjieISCAS/InProcessRecommendation.
[2] This article received an ACM SIGSOFT Distinguished Paper award at ICSE 2020.
[3] In our experimental platform, a report corresponds to either 0 or 1 bug, and there is no report containing more than 1 bug.

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

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 article assumes each crowdworker corresponds to a stable device variable.

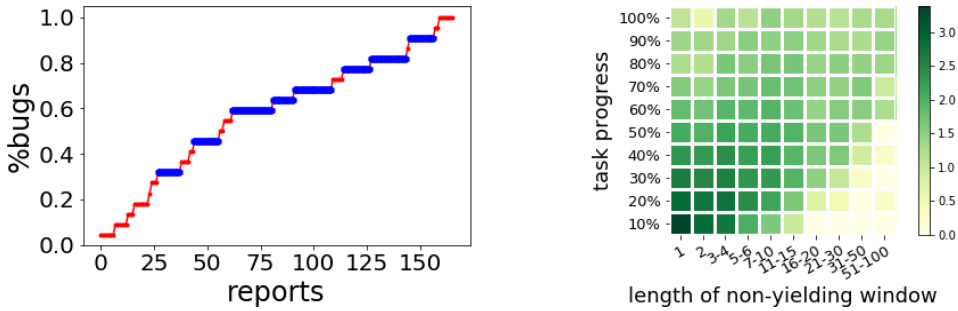
2.2 Non-yielding Windows in Crowdtesting Processes

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 306 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(a) 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.

We further investigate this phenomenon and present a summarized view in Figure 1(b). 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



(a) Bug arrival curve (b) Non-yielding windows

Fig. 1. Observations based on Baidu dataset.

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 then explore the cost waste of these non-yielding windows. Specifically, an average of 39% cost[4] 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[5] 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.

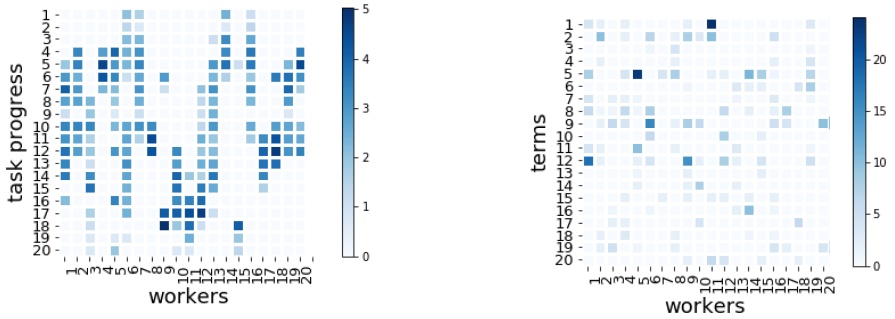
2.3 Characterizing Crowdworker's Bug Detection Capability

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 2(a) 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 2(b) shows the distribution of crowdworkers' activity at a finer granularity. The x-axis is the same as Figure 2(a), 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

[4]Following previous work [75, 77], we treat the number of reports as the amount of cost.
[5]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.



(a) Crowdworkers' activeness

(b) Crowdworkers' preference

Fig. 2. Characterizing crowdworker's bug detection capability.

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., a 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.

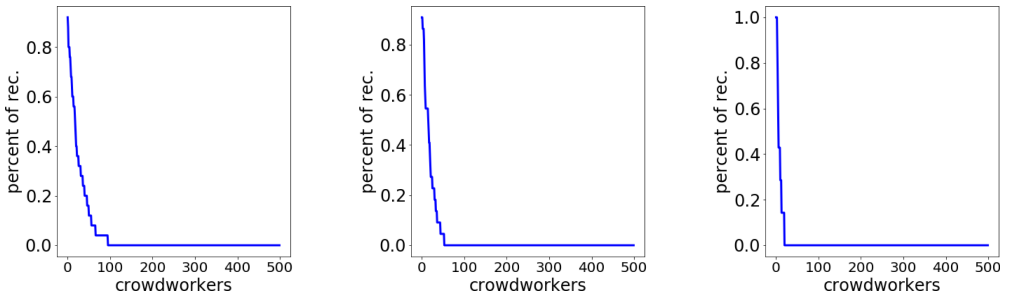
Expertise. Similarly, we explore the heat map with the terms from the crowdworkers' *bug reports* (rather than *reports*), we observe a similar trend. Due to space limit, we leave the detailed figure in our website. This indicates the crowdworkers' diversified expertise over different crowdtesting tasks. 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 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.

2.4 Observations on Popularity Bias in Existing Crowdworker Recommendation Approach

The popularity bias in recommendation systems has been noticed and investigated in product recommendation, i.e., the recommendation typically emphasizes popular items (those with more ratings) much more than other "long-tail" items [5, 7, 15, 62]. Researchers have pointed out that the recommendation should seek a balance between popular and less-popular items, so as to alleviate the item display difference and potential unfairness of the items.

In crowdworker recommendation, similarly, the crowdworkers hope to be recommended in a relatively fair manner, i.e., with no big difference in the recommended number of times. This



(a) Time slice -1 (b) Time slice -2 (c) Time slice -3

Fig. 3. Popularity bias in current crowdworker recommendation.

section seeks to explore the current status of popularity bias in crowdworker recommendation with the state-of-the-art approach iRec [78].

In detail, we run iRec and obtain the recommendation results for each crowdtesting task. We then count the number of tasks each crowdworker is recommended within one week, and the number of open tasks during same duration, then derive the percentage of tasks a crowdworker is recommended. We assume the distribution of this percentage among crowdworkers reflects the popularity bias. The reason we investigate the recommendation results by each week is to consider the time-series crowdworker activities as demonstrated in Section 2.3, and the duration one week is set empirically only for demonstration purpose.

Figure 3 presents the distribution of the percentage of recommendations in terms of three random-chosen time slices. We show the top 500 crowdworkers with the highest value to improve the readability of the plots (all other crowdworkers having zero values).

We can see that the recommended number of times are highly unevenly distributed, in which some highly experienced crowdworkers would be recommended in terms of almost all the tasks, yet some less experienced crowdworkers can only be recommended in a tiny fraction of tasks (or never be recommended). This implies the significant popularity bias and unfairness in current crowdworker recommendation approach. This work targets at alleviating the unfairness by introducing the fairness-aware aspect in the recommendation approach.

3 APPROACH

Figure 4 shows the overview of the proposed iRec2.0. 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 in the bottom-left corner of Figure 4.

iRec2.0 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 the 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 multi-objective optimization-based re-ranking component generates the re-ranking list of recommended workers in order to potentially reduce duplicate bugs and alleviate the unfairness among crowdworkers.

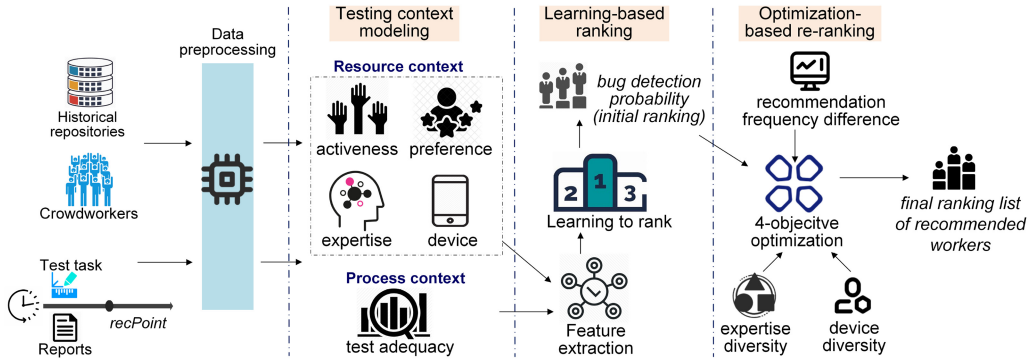


Fig. 4. Overview of iRec2.0.

Note that, the optimization-based re-ranking component is the new part which differs iRec2.0 from its pioneer iRec. There is a diversity-based re-ranking component after the learning-based ranking in iRec. The new re-ranking component in iRec2.0 employs the multi-objective optimization-based algorithm to optimize the re-ranking list, which can help adjust the whole re-ranking and would come out with improved results. By comparison, the diversity-based re-ranking in iRec uses a greedy-based algorithm to adjust the ranking list so that it is less effective.

iRec2.0 first develops a learning-based ranking which can find the workers with the greatest potential to detect bugs abstracted from historical tasks. Then it develops a multi-objective optimization-based re-ranking which jointly optimize the bug detection effectiveness (i.e., the bug detection probability in objective (1), the recommendation fairness (i.e., recommendation frequency difference among workers in objective 4), and others. We adopt the NSGA-II algorithm (i.e., Non-dominated Sorting Genetic Algorithm-II) for optimization. The NSGA-II algorithm is a widely used multi-objective optimizer in and out of the software engineering area. Taken in this sense, our proposed iRec2.0 can come out with improved bug detection results and recommendation fairness.

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 [72, 76], 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 [22, 75]. 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 [66], 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:

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

where $t_j \in$ *task terms vector*, i.e., it is one descriptive term in the description of task's requirements. 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.

In other words, $TestAdeq(t_j)$ is measured in terms of each descriptive term t_j in the task's requirements, i.e., the extent to which a descriptive term t_j has been covered by already submitted reports. Guided by $TestAdeq(t_j)$, iRec2.0 would try to find the workers who can cover the terms which are inadequate-tested. This is realized through the characterization of worker's preference and expertise which are also measured in terms of descriptive terms, which will be shown as follows.

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 [75, 88].

(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 [61] 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 *ProbPref* 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 [83]. 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 [32, 95].

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 [75, 97], 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

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*

examining the similarities. For brevity, we only present the details to produce the third group of features, i.e., #20–#26.

Previous work has proven extracting features from different perspectives can help improve the learning performance [17, 44, 60], 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 studies [29, 30, 72, 76], 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 are 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 [86, 96].

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 [69, 70], 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 [19, 60], 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 this ranked list of crowdworkers together with each worker's predicted bug detection probability, as the output of the learning-based ranking component, i.e., *initial ranking* in Figure 4.

3.4 Multi-Objective Optimization-based Re-ranking

To improve the bug detection performance and reduce potential duplicate reports, as discussed in Section 2.3, we should optimize the diversity among crowdworkers. Meanwhile, as indicated in Section 2.4, this study also hopes to balance the number of times a crowdworker get recommended in order to alleviate the issue of popularity bias. More than that, Section 3.3 has derived a ranked list of crowdworkers based on their probability in detecting bugs. Overall, we have the above several objectives to consider. Taken in this sense, we design a multi-objective optimization method to jointly optimize these objectives and attain the re-ranking list of recommended crowdworkers. The designed multi-objective optimization-based re-ranking method has the following four objectives to optimize.

First, we aim at ensuring the crowdworkers with higher probability in detecting bugs being ranked higher, so that more bugs can be revealed earlier. Second, we aim at ensuring the crowdworkers with larger degree of diverse expertise being ranked higher, in order to help produce less duplicate reports. Third, similar with expertise diversity, we also want to have the crowdworkers with larger degree of diverse device being ranked higher, so as to facilitate the exploration in new testing environment and revealing new bugs. Fourth, we hope the crowdworkers with the lower frequency of recommendation in previous tasks being ranked higher, so as to balance the recommendation frequency and alleviate the unfairness among crowdworkers. In the following subsections, we first illustrate the multi-objective optimization framework, followed by the details of these four objectives.

3.4.1 Multi-Objective Optimization Framework. iRec2.0 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 maintain the original ranking list of crowdworkers, thus, potentially sacrifice the other three objectives. Our proposed iRec2.0 seeks a *Pareto front* (or set of solutions). Solutions outside Pareto front cannot dominate (better than, under all objectives) any solutions within the front.

iRec2.0 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 [41], 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 [24].

In our recommendation scenario, a Pareto front represents the optimal trade-off between the four objectives determined by NSGA-II. The manager can then inspect a Pareto front to find the best compromise between having a crowdworker re-ranking list that balances bug detection probability, expertise diversity, device diversity, and recommendation frequency difference or alternatively having a re-ranking list that maximizes one/two/three objective/s penalizing the remain one/s.

iRec2.0 has the the following four steps:

(1) Solution encoding. Like other prioritization problems [25, 63], we encode each solution as a list of n integer numbers which are arranged as a permutation of size n. Each value of the permutation is stored in a solution variable. The solution space for the re-ranking problem is the set of all possible permutations about how the crowdworkers are ranked.

(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 [43].

(3) Genetic operators. For the evolution of permutation encoding for the solutions, we exploit standard operators as described in [79]. We use partially matched crossover and swap 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 3.4.2 to 3.4.5. For bug detection probability, expertise diversity, and device diversity, the larger these values are, the faster the convergence of a solution is. The recommendation frequency difference objective benefits from the smaller values.

3.4.2 Objective 1: Maximize Bug Detection Probability. The bug detection probability of a crowdworker is obtained based on the trained ranking model in Section 3.3. It denotes the success probability of a crowdworker in detecting bugs with respect to the particular testing context.

We refer to the multi-objective test case prioritization studies [25, 63] to measure this objective for each solution. Bug detection probability for a solution s_j (i.e., a candidate re-ranked list of crowdworkers) can be calculated as follows.

$$BDPrb_{s_j} = \frac{\sum_{i=1}^{n} BDP(w_i) \times \frac{n-i+1}{n}}{\sum_{i=1}^{n} BDP(w_i)}, \tag{7}$$

where n is the total number of workers in the solution, w_i is the worker being ranked in the i_{th} place in the solution, and $BDP(w_i)$ represents the bug detection probability of worker w_i.

A higher value of bug detection probability implies a crowdworker is more capable in finding bugs. The goal is to maximize the bug detection probability of a solution since we aim at finding a re-ranking list of crowdworkers that detect bugs as early as possible, i.e., having the workers with higher bug detection probability being ranked higher.

3.4.3 Objective 2: Maximize Expertise Diversity. We define expertise diversity delta, which measures the newly-added expertise diversity of a worker with respect to current re-ranked list of workers (i.e., the workers ahead of the considered worker in the re-ranked list).

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

$$ExpDivDlt(w_i, R) = \sum_{t_j} ProfExp(w_i, t_j) \times \prod_{w_k \in R} (1.0 - ProfExp(w_k, t_j)), \tag{8}$$

where the first part is the expertise of crowdworker w_i towards the descriptive term t_j, and the later part (i.e., $\prod$) estimates the extent to which term t_j is tested by the workers on the current re-ranked list.

Similar with Section 3.4.2, the expertise diversity for a solution s_j can be calculated as follows.

$$ExpDiv_{s_j} = \frac{\sum_{i=1}^{n} ExpDivDlt(w_i, R) \times \frac{n-i+1}{n}}{\sum_{i=1}^{n} ExpDivDlt(w_i, R)}, \tag{9}$$

where n is the total number of workers in the solution, R is the current re-ranked list of $i-1$ workers, and w_i is the worker being ranked in the i_th place in the solution.

A higher value of expertise diversity delta implies a crowdworker can contribute more different expertise with respect to the current re-ranked list. The goal is to maximize the expertise diversity of a solution since we aim at finding a re-rank list of crowdworkers that demonstrates diversified expertise as early as possible.

3.4.4 Objective 3: Maximize Device Diversity. Similar with Section 3.4.3, we define device diversity delta, which measures the newly-added device diversity of a worker with respect to current re-ranked list of workers.

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

$$DevDivDlt(w_i, R) = (w_i's\ attributes) - \cup_{w_k \in R}(w_k's\ attributes), \tag{10}$$

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

The device diversity for a solution s_j is calculated similar as $ExpDiv_{s_j}$. And the goal is to maximize the device diversity of a solution since we aim at finding a re-rank list of crowdworkers that contribute various device attributes as early as possible.

3.4.5 Objective 4: Minimize Recommendation Frequency Difference. The recommendation frequency denotes the frequency of each worker being recommended in the short past. It is obtained based on the recommendation results on the open crowdtesting tasks during previous week. The reason why we measure it by one week is to consider the time-series crowdworker activities as demonstrated in Section 2.3, and we find the features *NumBugs-1 week* and *NumReports-1 week* (in Table 2) play relatively large role than other activeness-related features. It is measured as the percentage of tasks where a worker is being recommended among all the tasks under recommendation in the past week, and is represented as $RecFrq(w_i) = \frac{\#tasks\ where\ w_i\ is\ recommended}{\#tasks\ under\ recommendation\ in\ past\ week}$.

For the recommendation frequency difference among the crowdworkers, we hope to have the crowdworkers with the smaller recommendation frequency being ranked higher, so that the recommendation frequency difference of the list of crowdworkers can be balanced. Similar as Section 3.4.2, recommendation frequency difference for a solution s_j can be calculated as follows.

$$RecFrqDif\, s_j = \frac{\sum_{i=1}^{n} RecFrq(w_i) \times \frac{n-i+1}{n}}{\sum_{i=1}^{n} RecFrq(w_i)} \tag{11}$$

Note that, a smaller value of recommendation frequency difference implies a better solution. The goal is to minimize the recommendation frequency difference of a solution since we aim at

having the crowdworkers with smaller recommendation frequency being ranked higher, so as to balance the recommendation number of times among workers and alleviate the unfairness among crowdworkers.

4 EXPERIMENT DESIGN

4.1 Research Questions

— **RQ1:** (Performance Evaluation) How effective is iRec2.0 for crowdworker recommendation?

For RQ1, we first present some general views of iRec2.0 for worker recommendation. To further demonstrate its advantages, we then compare its performance with five state-of-the-art and commonly-used baseline methods (details are in Section 4.5).

— **RQ2:** (Context Sensitivity) To what degree iRec2.0 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 iRec2.0 when removing different sub-category of the context, to understand the context sensitivity of recommendation.

— **RQ3:** (Re-ranking Gain) How much is the re-ranking gain by introducing the multi-objective optimization-based method in recommendation?

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

— **RQ4:** (Optimization Quality) Do the results of iRec2.0 achieve high quality?

RQ4 is to evaluate the quality of Pareto fronts produced by our multi-objective optimization-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 4.4).

— **RQ5:** (Runtime Overhead) What is the runtime cost of iRec2.0?

Since the multi-objective optimization algorithm is commonly-known as time-consuming, RQ5 is to investigate the runtime overhead of iRec2.0 to further demonstrate its practical value.

4.2 Dataset

We collected crowdtesting data from Baidu[6] 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, and so on. 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 iRec2.0 in practice, we employ a commonly-used longitudinal data setup [68, 72, 77]. That is, all the 636 experimental tasks were sorted in the chronological order, and

[6] test.baidu.com.

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 iRec2.0 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 iRec2.0 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), others demonstrate similar trend. The size of the experimental dataset (i.e., number of total *recPoint*) under the four *recThres* are 676, 479, 345, and 278, respectively.

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 [22, 23, 75] 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.

Furthermore, to measure the role of re-ranking in alleviating the unfairness, we additionally obtain **fairRate@k** to measure the frequency of the crowdworkers being recommended. Related studies utilized similar indicator for measuring the fairness and popularity bias. For example, [5] uses the number of long tail items, and [14] counts the amount of popular items in high positions. For *fairRate@k*, we first calculate the percentage of tasks where each crowdworker is recommended in the past one week, and then obtain the average percentage for the top k recommended workers in this recommendation. As *BDR@k*, we set k as 3, 5, 10, and 20 since a smaller subset is preferred in crowdworker recommendation. Take *fairRate@3* of a task being 80% as an example, it denotes, for that task, the top 3 recommended workers are recommended in an average of 80% open tasks in the past week.

To further demonstrate the superiority of our proposed approach, we perform one-tailed Mann Whitney U test [58] between our proposed iRec2.0 and other approaches. We include the Bonferroni correction [84] 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 [21]).

In addition, we apply *HV*, *IGD*, and *GS* to evaluate the quality of Pareto fronts produced by our multi-objective optimization-based re-ranking, which have been widely used in existing Search-Based Software Engineering studies [26, 41, 79]. 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; i.e., *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*. ***GS*** is a diversity indicator. It computes the extent of spread for the non-dominated solutions found by the algorithm. A higher value of GS shows that the results have a better distribution; i.e., *higher* values of GS are *better*. Due to the limited space, for details about the three quality indicators, please refer to [79].

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 iRec2.0, we compare iRec2.0 with five commonly-used and state-of-the-art baselines.

iRec [78]**:** This is the state-of-the-art crowdworker recommendation approach to recommend a diverse set of capable crowdworkers based on the dynamic contextual information. The difference between iRec and the newly-proposed iRec2.0 is that iRec develops a diversity-based re-ranking method to generate the final ranking of recommended workers which aims at improving the diversity among crowdworkers, while iRec2.0 designs a multi-objective optimization-based re-ranking method to optimize both the diversity and the recommendation fairness.

MOCOM [75]**:** 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 [22]**:** This is a weight-based crowdworker recommendation approach that linearly combines experience strategy, relevance strategy, and diversity strategy.

MOOSE [23]**:** 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.

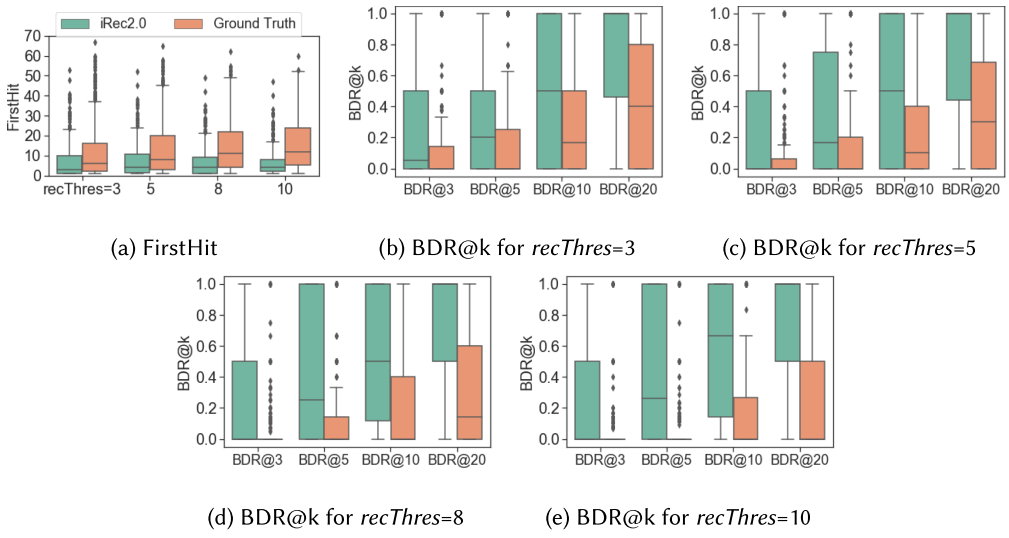
Cocoon [88]**:** This crowdworker recommendation approach is designed to maximize the testing quality (measured in worker's historical submitted bugs) under the test coverage constraint.

For baseline *iRec*, we use the same experimental setup as the newly-proposed *iRec2.0*. For other four baselines, since they are not proposed for the in-process recommendation, 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 RESULTS AND ANALYSIS

5.1 Answering RQ1: Performance Evaluation

Figure 5(a) 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 iRec2.0 is significantly (p-value is 0.00) and substantially (Cliff's delta is 0.23–0.39) better than current practice of crowdtesting. When *recThres* is 5, the median *FirstHit* of iRec2.0 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), iRec2.0 can shorten the non-yielding window by 50% to 66%.



(a) FirstHit (b) BDR@k for *recThres*=3 (c) BDR@k for *recThres*=5

(d) BDR@k for *recThres*=8 (e) BDR@k for *recThres*=10

Fig. 5. Performance of iRec2.0.

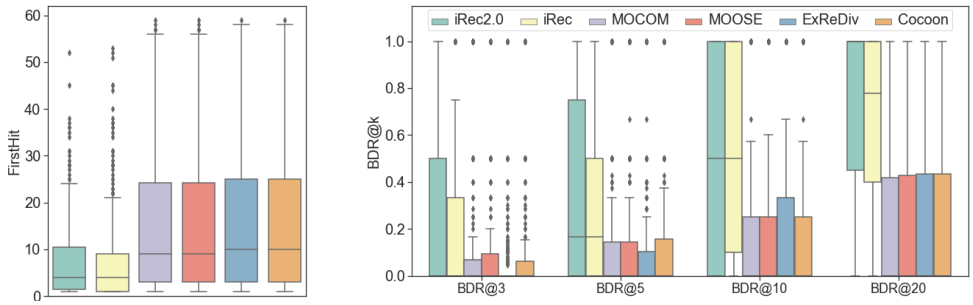
Figure 5(b) to 5(e) demonstrate the *BDR@k* of worker recommendation under four representative *recThres*. iRec2.0 significantly (p-value is 0.00) and substantially (Cliff's delta is 0.24–0.41) outperforms current practice of crowdtesting for BDR@k (k is 3, 5, 10, and 20). When *recThres* is 5, a median of 50% remaining bugs can be detected with the first 10 recommended crowdworkers by our proposed iRec2.0, with 400% improvement compared with current practice of crowdtesting (50% vs. 10%). Besides, a median of 100% remaining bugs can be detected with the first 20 recommended crowdworkers by iRec2.0, with 230% improvement compared with current practice (100% 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 iRec2.0 over current practice is larger. In detail, when *recThres* is 3, iRec2.0 can improve the current practice by 150% (100% vs. 40%) for *BDR@20*, and when *recThres* is 8, the improvement is 600% (100% 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 iRec2.0 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. For example, for the *recPoint* whose *FirstHit of Ground Truth* is 6 (*resThres* is 3), iRec2.0 can shorten the non-yielding window by 50% on median (3 vs. 6), while when *FirstHit of Ground Truth* is 12 (*resThres* is 10), the improvement is 66% (4 vs. 12). 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.

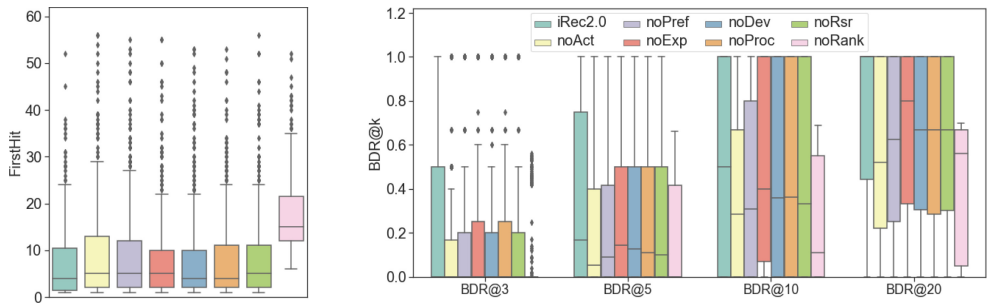
In the following article, we use the experimental setting when *recThres* is 5 for further analysis and comparison due to space limit.

Comparison with Baselines. Figure 6 demonstrates the comparison results with five baselines. We first put our focus on the last four baselines. Overall, our proposed iRec2.0 significantly (p-value is 0.00) and substantially (Cliff's delta is 0.16–0.25) outperforms the last four baselines in terms of *FirstHit* and *BDR@k* (k is 3, 5, 10, and 20). Specifically, iRec2.0 can improve the best baseline *MOCOM* by 60% (4 vs. 10) for median *FirstHit*; and the improvement is infinite for median



(a) FirstHit (b) BDR@k

Fig. 6. Performance comparison with baselines.



(a) FirstHit (b) BDR@k

Fig. 7. Context sensitivity.

BDR@k (e.g., 100% vs. 0 for BDR@20). This is because all these 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.

For our previous proposed iRec, the newly-proposed iRec2.0 has the same median *BDR@k* for *k* is 3, 5, and 10, and has better median *BDR@20* than iRec. Furthermore, iRec2.0 outperforms iRec in the average *BDR@k* (k is 3, 5, 10, and 20). For example, iRec2.0 can improve the average *BDR@3* by 20% (25% vs. 21%), and can improve the average *BDR@10* by 10% (53% vs. 48%). iRec2.0 has the same median *FirstHit*, and is slightly (7%) inferior in the average *FirstHit*, i.e., 7.78 vs. 7.21. Overall, the newly-proposed iRec2.0 is better than iRec in bug detection performance. This is because iRec utilizes the greedy strategy in optimizing the diversity among crowdworkers, while iRec2.0 designs a multi-objective optimization-based method which has better chances in achieving more optimized solution.

5.2 Answering RQ2: Context Sensitivity

Figure 7 shows the comparison results between iRec2.0 and its seven variants. Specifically, *noAct*, *noPref*, *noExp*, and *noDev* are different variants of iRec2.0 without activeness, preference, expertise, and device context, respectively. Because process context cannot be removed, *noProc* denotes

Table 3. Role of Re-ranking in Bug Detection

	FirstHit	BDR@3	BDR@5	BDR@10	BDR@20
Average performance					
iRec2.0	7.78%	25.3%	36.0%	53.0%	68.1%
iRec2.0 without re-ranking	8.35%	18.4%	26.7%	41.6%	59.1%
iRec	7.21%	21.5%	32.1%	48.6%	67.1%
Improvement					
iRec2.0 vs. iRec2.0 without re-ranking	6.8%	38.8%	38.4%	29.2%	15.2%
iRec2.0 vs. iRec	−7.9%	17.6%	12.1%	9.1%	1.5%

using the process context at the beginning of a task. *noRsr* denotes using the resource context at the beginning of the task to further demonstrate the necessity of precise context modeling. We additionally add *noRank* which denotes using the random list of crowdtesting workers as the initial ranking for the re-ranking optimization.

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 iRec2.0 further indicates the necessity of the precise context modeling.

In addition, we can also see that with the randomly generated initial rank, iRec2.0 performs bad. For instance, with iRec2.0, a median of 50% remaining bugs can be detected with the first 10 recommended crowdworkers, while this number declines to 10% when the initial ranking does not exist. This is because the initial ranking learns the successful knowledge about the bug detection potential of crowdworkers from historical tasks, and has great indicative effect on their bug detection performance on this new task. Without such information, the optimization-based re-ranking lacks of the guidance of finding capable workers in bug detection, and would act like searching a diverse set of workers based on expertise, device, and so on.

5.3 Answering RQ3: Re-ranking Gain

Table 3 demonstrates the average bug detection performance of iRec2.0, *iRec2.0 without re-ranking*, and iRec, followed by the improvement of iRec2.0. We can see that with the re-ranking component, the average bug detection performance can be improved by 6.8% to 38.8%. Specifically, the re-ranking can increase the *BDR@3* by 38.8% and increase the *BDR@10* by 29.2%. This is because there are large amount of duplicate bugs, and increasing the expertise diversity and device diversity of recommended workers can help decrease the duplicate bugs so as to increase the unique bugs. Furthermore, for all the investigated k in *BDR@k*, the re-ranking can improve the bug detection rate, indicating no matter how many crowdworkers are recommended, the bug detection performance can be improved. This is because we employ the multi-objective optimization-based re-ranking method to optimize the re-ranking list which can help adjust the whole re-ranking and improve it at each inspected point.

The extended iRec2.0 outperforms its pioneer iRec in most metrics for bug detection performance, which has been illustrated in Section 5.1. Nevertheless, a slight decrease in *FirstHit* of

Table 4. Role of Re-ranking in Alleviating Unfairness

	fairRate@3	fairRate@5	fairRate@10	fairRate@20
Average performance				
iRec2.0	5.6%	7.4%	12.9%	26.1%
iRec2.0 without re-ranking	61.1%	62.2%	57.6%	48.9%
iRec	60.2%	54.3%	41.4%	34.9%
Improvement				
iRec2.0 vs. iRec2.0 without re-ranking	90.8%	88.1%	77.6%	47.8%
iRec2.0 vs. iRec	90.6%	86.3%	68.8%	26.9%

iRec2.0 is observed, i.e., from 7.78 to 7.21, possibly because the fair-oriented re-ranking would occasionally compromise bug detection performance of recommended crowdworkers, especially in hitting the first correct worker.

Table 4 presents the average recommendation fairness *fairRate@k* of iRec2.0, *iRec2.0 without re-ranking*, and iRec, followed by the improvement of iRec2.0. We can see that, before applying re-ranking, *fairRate@5* is 62.2% and *fairRate@20* is 48.9%, indicating the top 5 recommended crowdworkers have been recommended in 62% tasks in the past one week, and when we consider the top 20 recommended crowdworkers, this ratio is 48%. After applying re-ranking, the top 5 recommended crowdworkers are only recommended in 7% tasks in the past one week, and the top 20 recommended crowdworkers are only recommended in 26% tasks in the past one week. The dramatic reduction of recommendation frequency of top workers shows that iRec2.0 is able to mitigate popularity bias and produces more fair recommendations. Also, remember that, it can retain or increase the bug detection efficiency in the meanwhile.

When compared with iRec, the newly-proposed iRec2.0 also outperform it in all the evaluation metrics, with 47% to 90% improvement.

We have also counted the percentage of tasks each crowdworker would take per week in our experimental crowdtesting platform, and the ratio is 25%, which is almost equal with our recommendation frequency for the top 20 crowdworkers. This indicates, when we send the recommendation invitation to the top 20 crowdworkers produced by our approach, the recommendation frequency received by the crowdworker is similar with the frequency he/she takes the tasks. This further implies the potential practicability of our worker recommendation approach in real-world crowdtesting scenario.

5.4 Answering RQ4: Optimization Quality

Since iRec2.0 is a multi-objective optimization-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., *HV*, *IGD*, and *GS* [79], are applied. For each experiment, we present the value of each quality indicator obtained by iRec2.0 in Figure 8.

We can see that most experiments have very high *HV* values, very low *IGD* values, and very high *GS* values. The average *HV* is 0.79, the average *IGD* is 0.01, and the average *GS* is 0.83. This denotes our optimization has achieved high quality. Existing researches on test case selection and worker selection achieve similar results [23, 26]. This further suggests that the results of iRec2.0 have high quality.

5.5 Answering RQ5: Runtime Overhead

The runtime overhead of iRec2.0 is composed of two parts: the training of learning-based ranking model and the crowdworker recommendation. The training of learning-based ranking model consumes 4.35 minutes; yet it can be conducted offline and would not influence the application

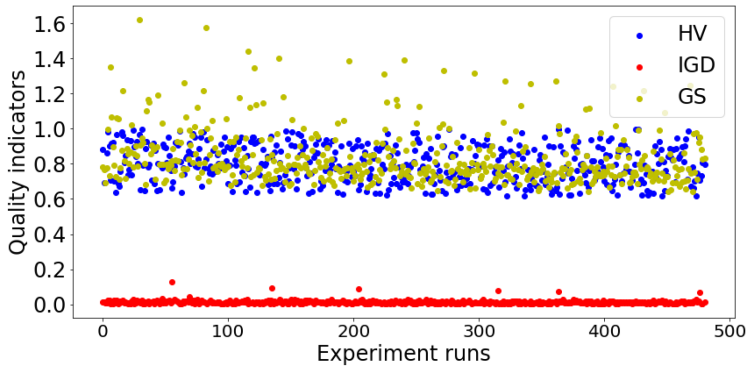
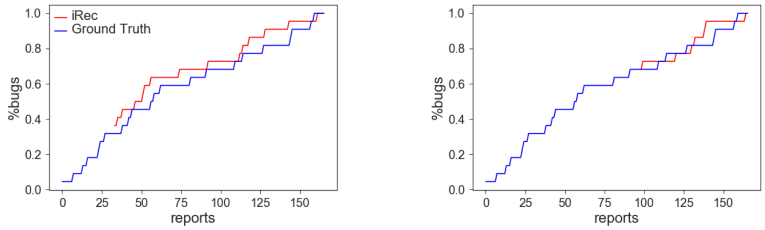


Fig. 8. Quality of optimization.



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

Fig. 9. Illustrative examples of iRec2.0.

of iRec2.0 in real-world practice. The crowdworker recommendation consumes an average of 4.24 seconds (the minimum is 0.92 seconds and the maximum is 6.68 seconds) for all experiment runs. The small runtime overhead of iRec2.0 again implies its practical value in real-world crowdtesting scenario.

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 iRec2.0 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.

We utilize illustrative examples to demonstrate the benefits of the application of iRec2.0. Figure 9 demonstrates two typical bug detection curve using iRec2.0 for two *recPoint* of the task in Figure 1(a). We can easily see that with iRec2.0, 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 9(a), 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 iRec2.0, 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 9(b), although the bug detection curve can not always dominate

Table 5. Reduced Cost with iRec2.0

	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%

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.

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 iRec2.0 for in-process crowdworker recommendation. It is measured based on the number of reduced report, i.e., the difference of *FirstHit* value between iRec2.0 and *Ground Truth*, following previous work [75, 77]. For a crowdtesting task with multiple *recPoint*, we simply add up the reduced cost of each *recPoint*. As shown in Table 5, 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 9, 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 iRec2.0.

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 Objectivity vs. Fairness

The crowdworker recommendation of this study lies in the characterization of workers learned from historical data of the crowdtesting platform, as well as fairness-aware adjustment to

alleviate popularity bias. In another word, the generated recommendation results is a list of crowdworkers reflecting the multi-objective optimization results of balanced objectivity and fairness goals. This advances existing objectivity-only approached such as iRec, which leads to overloaded expert workers and potential bottleneck resources.

However, like other history-based recommendation [34, 82], the proposed approach also suffers from the cold-start problem [89], i.e., unable to provide recommendation for newcomers who do not own any history yet. To accommodate cold-start problems, one can use a calibrated characterization for newcomers, e.g., incorporate such static attributes of the workers as occupation, interest for modeling. By summarizing hot technical aspects from recent open tasks, a decision tree type of preference/expertise questionnaire can be formulated and is presented to the new comers, so that a default worker characterization can be configured for the new comers and used for recommendation systems like iRec2.0.

From another point of view, ranking of people and items are at the heart of selection-making, match-making, and RS, ranging from e-commerce to crowdsourcing platforms. As ranking positions influence the amount of attention the ranked subjects would receive, biases in rankings can lead to unfair distribution of opportunities and resources such as buy decisions [5, 13]. Existing work focused on the equity of attention when talking about the fairness, and they think it is the true fairness [7, 13, 15].

However, in crowdworker recommendation, we also observed that different crowdworkers tend to have different working habit and affordable workload, e.g., some workers constantly take less than three tasks a week while other workers can finish ten tasks a week in our experimental dataset. Considering this difference, we argue the equity of attention is far from the true fairness in worker recommendation scenario, and recommending tasks in accordance of each crowdworker' aptitude might be preferred. Nevertheless, the true fairness is very challenging to define and achieve, and requires future human-centered design research to explore it.

6.4 Threats to Validity

First, following existing work [75, 77], we use the number of crowdtesting reports as the amount of cost when measuring the reduced cost. As discussed in [77], 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 [32, 95]. This can be done manually or with automatic tool support (e.g., [72, 73]). Therefore, we assume our designed methods can be easily adopted in the crowdtesting platform.

Third, we evaluate iRec2.0 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 [16, 39, 45, 68, 90]. In real-world practice, iRec2.0 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 [1] 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,

learning-based or optimization-based ranking (re-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 optimization 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 [20], usability testing [37], software performance analysis [57], software bug detection, and reproduction [36]. There were dozens of approaches focusing on the new encountered problems in crowdtesting, e.g., crowdtesting reports prioritization [29, 30, 46], reports summarization [40], reports classification [72–74, 76], automatic report generation [48], crowdworker recommendation [22, 23, 75, 88], crowdtesting management [77], and so on.

There were many lines of related studies for recommending workers for various software engineering tasks, such as bug triage [10, 12, 45, 53, 59, 68, 80, 81, 87, 90, 94], code reviewer recommendation [27, 39, 93], expert recommendation [16, 51], developer recommendation for crowdsourced software development [47, 52, 91, 92], worker recommendation for general crowdsourcing tasks [9, 49, 65], and so on. 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 [75, 88], experience [22, 88], capability [75], expertise with the task [22, 23, 75], and so on. 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 [55, 56], 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 [33], software documentation [8], static analysis [42], and so on. This work provides new insights about how to model and utilize the context information in open environment.

The growing ubiquity of data-driven learning models in algorithmic decision-making has recently boosted concerns about the issues of fairness and bias. Friedman defined that a computer system is biased "if it systematically and unfairly discriminates against certain individuals or groups of individuals in favor of others" [31]. For example, job recommenders can target women with lower-paying jobs than equally-qualified men [28]. News recommenders can favor particular political ideologies over others [50]. And even ad recommenders can exhibit racial discrimination [67]. The fairness in data-driven decision-making algorithms (e.g., recommendation systems) requires that similar individuals with similar attributes, e.g., gender, age, race, religion, and so on, be treated similarly. For instance, the fairness-aware news recommendation aims at alleviating the unfairness in news recommendation brought by the biases related to sensitive user attributes like genders [85]. Geyik et al. proposed a framework for fairness-aware ranking of job searching results based on desired proportions over the protected attribute such as gender or age [35].

Another type of unfairness in recommendation systems, which is also well studied by researchers [5–7, 15, 54, 62], is the problem of popularity bias, i.e., popular items are being recommended too frequently while the majority of other items do not get the deserved attention. However, less popular, long-tail items are precisely those that are often desirable recommendations. A

market that suffers from popularity bias will lack opportunities to discover more obscure products and will be dominated by a few large brands or well-known artists [18]. Such a market will be more homogeneous and offer fewer opportunities for innovation and creativity [7]. To tackle this, Abdollahpouri et al. proposed a regularization-based framework to enhance the long-tail coverage of recommendation lists and balance the recommendation accuracy and coverage [5]. Borges et al. proposed a method that penalizes scores given to items according to historical popularity for mitigating the bias [14].

In crowdworker scenario as this work, we did not observe the sensitive user attributes towards which the recommendation is biased. And the fairness in this work refers to the popularity bias in the crowdworker recommendation results, which is mentioned in the pilot study.

There were researches exploring the fairness problems and solutions in various areas, e.g., e-commerce product recommendation [5, 62], search engine [13, 15], employment [64], and so on. This article focuses on alleviating the unfairness in worker recommendation, which is another important application scenario. Existing work suggests fairness pipeline [11] for detecting and mitigating algorithmic bias that introduces unfairness or inequality. The fairness pipeline handles different bias through pre-processing to remove data bias, in-processing to address algorithm bias, post-processing to mitigate recommendation bias [38, 71]. In this study, we attempt to address the popularity bias and alleviate the unfairness by formulating the fairness-aware optimization problem, which falls into the in-processing category.

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, as well as the popularity bias among the crowdworkers.

This article proposed a context- and fairness-aware in-process crowdworker recommendation approach, iRec2.0, to bridge this gap. Built on top of a fine-grained context model, iRec2.0 incorporates the learning-based ranking component and multi-objective optimization-based re-ranking component for worker recommendation. The evaluation results demonstrate its potential benefits in shortening the non-yielding window, improving bug detection efficiency, and alleviating the unfairness in the recommendations.

Directions of future work include: (1) design and conduct user study to validate the usage of iRec2.0; (2) further evaluate iRec2.0 on cross-platform datasets; (3) incorporate more context-related information to improve the performance; and (4) explore the true fairness with human-centered design researches.

REFERENCES

[1] 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.
[2] Retrieved on 2021 from https://www.topcoder.com/.
[3] Retrieved on 2021 from https://www.applause.com/.
[4] Retrieved on 2021 from https://www.testbird.com/.
[5] Himan Abdollahpouri, Robin Burke, and Bamshad Mobasher. 2017. Controlling popularity bias in learning-to-rank recommendation. In *Proceedings of the 11th ACM Conference on Recommender Systems*. 42–46.
[6] Himan Abdollahpouri, Masoud Mansoury, Robin Burke, and Bamshad Mobasher. 2019. The unfairness of popularity bias in recommendation. In *Proceedings of the Workshop on Recommendation in Multi-stakeholder Environments co-located with the 13th ACM Conference on Recommender Systems*.
[7] Himan Abdollahpouri, Masoud Mansoury, Robin Burke, and Bamshad Mobasher. 2020. The connection between popularity bias, calibration, and fairness in recommendation. In *Proceedings of the RecSys 2020: 14th ACM Conference on Recommender Systems*. 726–731.

[8] Emad Aghajani. 2018. Context-Aware software documentation. In *Proceedings of the 2018 IEEE International Conference on Software Maintenance and Evolution*. 727–731.
[9] Eiman Aldahari, Vivek Shandilya, and Sajjan G. Shiva. 2018. Crowdsourcing multi-objective recommendation system. In *Proceedings of the Companion of the The Web Conference 2018 on The Web Conference 2018*. 1371–1379.
[10] John Anvik, Lyndon Hiew, and Gail C. Murphy. 2006. Who should fix this bug? In *Proceedings of the 28th International Conference on Software Engineering*. 361–370.
[11] Rachel K. E. Bellamy, Kuntal Dey, Michael Hind, Samuel C. Hoffman, Stephanie Houde, Kalapriya Kannan, Pranay Lohia, Jacquelyn Martino, Sameep Mehta, Aleksandra Mojsilović, Seema Nagar, Karthikeyan Natesan Ramamurthy, John Richards, Diptikalyan Saha, Prasanna Sattigeri, Moninder Singh, Kush Varshney, and Yunfeng Zhang. 2019. AI fairness 360: An extensible toolkit for detecting and mitigating algorithmic bias. *IBM Journal of Research and Development* 63, 4/5 (2019), 4–1.
[12] Pamela Bhattacharya and Iulian Neamtiu. 2010. Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging. In *Proceedings of the 2010 IEEE International Conference on Software Maintenance*. 1–10.
[13] Asia J. Biega, Krishna P. Gummadi, and Gerhard Weikum. 2018. Equity of attention: Amortizing individual fairness in rankings. In *Proceedings of the 41st International ACM SIGIR Conference on Research & Development in Information Retrieval*. 405–414.
[14] Rodrigo Borges and Kostas Stefanidis. 2021. On mitigating popularity bias in recommendations via variational autoencoders. In *Proceedings of the SAC'21: The 36th ACM/SIGAPP Symposium on Applied Computing*. 1383–1389.
[15] Rocío Cañamares and Pablo Castells. 2017. A probabilistic reformulation of memory-based collaborative filtering: Implications on popularity biases. In *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*. 215–224.
[16] Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. 2012. Who is going to mentor newcomers in open source projects? In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 44.
[17] Zherui Cao, Yuan Tian, Tien-Duy B. Le, and David Lo. 2018. Rule-based specification mining leveraging learning to rank. *Automated Software Engineering* 25, 3 (2018), 501–530.
[18] Òscar Celma and Pedro Cano. 2008. From hits to niches? Or how popular artists can bias music recommendation and discovery. In *Proceedings of the 2nd KDD Workshop on Large-Scale Recommender Systems and the Netflix Prize Competition*. Association for Computing Machinery, Article 5, 8.
[19] 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*. 456–467.
[20] Ning Chen and Sunghun Kim. 2012. Puzzle-based automatic testing: Bringing humans into the loop by solving puzzles. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 140–149.
[21] N. Cliff. 2014. *Ordinal Methods for Behavioral Data Analysis*. Psychology Press.
[22] 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 *Proceedings of the 2017 IEEE 41st Annual Computer Software and Applications Conference*. 75–84.
[23] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-Objective crowd worker selection in crowdsourced testing. In *Proceedings of the 29th International Conference on Software Engineering and Knowledge Engineering*. 218–223.
[24] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and Tanaka Meyarivan. 2000. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In *Proceedings of the International Conference on Parallel Problem Solving From Nature*. Springer, 849–858.
[25] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 234–245.
[26] Michael G. Epitropakis, Shin Yoo, Mark Harman, and Edmund K. Burke. 2015. Empirical evaluation of pareto efficient multi-objective regression test case prioritisation. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. 234–245.
[27] 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.
[28] Ayman Farahat and Michael C. Bailey. 2012. How effective is targeted advertising? In *Proceedings of the 21st World Wide Web Conference 2012*. 111–120.
[29] Yang Feng, Zhenyu Chen, James A. Jones, Chunrong Fang, and Baowen Xu. 2015. Test report prioritization to assist crowdsourced testing. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*. 225–236.

[30] Yang Feng, James A. Jones, Zhenyu Chen, and Chunrong Fang. 2016. Multi-objective test report prioritization using image understanding. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 202–213.

[31] Batya Friedman and Helen Nissenbaum. 1996. Bias in computer systems. *ACM Transactions on Information Systems* 14, 3 (1996), 330–347.

[32] 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* 24, 2 (2018), 1–25.

[33] Marko Gasparic, Gail C. Murphy, and Francesco Ricci. 2017. A context model for IDE-based recommendation systems. *Journal of Systems and Software* 128, C (2017), 200–219.

[34] Theodore Georgiou, Amr El Abbadi, and Xifeng Yan. 2017. Extracting topics with focused communities for social content recommendation. In *Proceedings of the 2017 ACM Conference on Computer Supported Cooperative Work and Social Computing*. 1432–1443.

[35] Sahin Cem Geyik, Stuart Ambler, and Krishnaram Kenthapadi. 2019. Fairness-Aware ranking in search & recommendation systems with application to linkedin talent search. In *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 2221–2231.

[36] María Gómez, Romain Rouvoy, Bram Adams, and Lionel Seinturier. 2016. Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring. In *Proceedings of the 2016 IEEE/ACM International Conference on Mobile Software Engineering and Systems*. IEEE, 88–99.

[37] Victor H. M. Gomide, Pedro A. Valle, José O. Ferreira, José R. G. Barbosa, Adson F. Da Rocha, and TMGdA Barbosa. 2014. Affective crowdsourcing applied to usability testing. *International Journal of Computer Science and Information Technologies* 5, 1 (2014), 575–579.

[38] Sara Hajian, Francesco Bonchi, and Carlos Castillo. 2016. Algorithmic bias: From discrimination discovery to fairness-aware data mining. In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. 2125–2126.

[39] 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*. 99–110.

[40] Rui Hao, Yang Feng, James Jones, Yuying Li, and Zhenyu Chen. 2019. CTRAS: Crowdsourced test report aggregation and summarization. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 921–932.

[41] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys* 45, 1 (2012), 11.

[42] 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*. 94–104.

[43] J. H. Holland. 1992. *Genetic Algorithms*. Scientific American.

[44] 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.

[45] Gaeul Jeong, Sunghun Kim, and Thomas Zimmermann. 2009. Improving bug triage with bug tossing graphs. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*. 111–120.

[46] 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.

[47] Muhammad Rezaul Karim, Ye Yang, David Messinger, and Guenther Ruhe. 2018. Learn or earn? - Intelligent task recommendation for competitive crowdsourced software development. In *Proceedings of the 51st Hawaii International Conference on System Sciences*. 1–10.

[48] Di Liu, Xiaofang Zhang, Yang Feng, and James A. Jones. 2018. Generating descriptions for screenshots to assist crowdsourced testing. In *Proceedings of the 25th International Conference on Software Analysis, Evolution and Reengineering*. 492–496.

[49] Zheng Liu and Lei Chen. 2017. Worker recommendation for crowdsourced Q&A services: A triple-factor aware approach. *Proceedings of the VLDB Endowment* 11, 3 (2017), 380–392.

[50] Jacqueline M. Otala, Gillian Kurtic, Isabella Grasso, Yu Liu, Jeanna Matthews, and Golshan Madraki. 2021. Political polarization and platform migration: A study of parler and twitter usage by united states of america congress members. In *Proceedings of the Companion Proceedings of the Web Conference*. 224–231.

[51] David Ma, David Schuler, Thomas Zimmermann, and Jonathan Sillito. 2009. Expert recommendation with usage expertise. In *Proceedings of the 2009 IEEE International Conference on Software Maintenance*. 535–538.

[52] Ke Mao, Ye Yang, Qing Wang, Yue Jia, and Mark Harman. 2015. Developer recommendation for crowdsourced software development tasks. In *Proceedings of the 2015 IEEE Symposium on Service-Oriented System Engineering*. 347–356.

[53] Dominique Matter, Adrian Kuhn, and Oscar Nierstrasz. 2009. Assigning bug reports using a vocabulary-based expertise model of developers. In *Proceedings of the 2009 6th IEEE International Working Conference on Mining Software Repositories*. 131–140.

[54] Marco Morik, Ashudeep Singh, Jessica Hong, and Thorsten Joachims. 2020. Controlling fairness and bias in dynamic learning-to-rank. In *Proceedings of the 43rd International ACM SIGIR Conference on Research and Development in Information Retrieval*. 429–438.

[55] 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*. 5–5.

[56] 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*. 73–76.

[57] 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.

[58] Nadim Nachar. 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.

[59] Hoda Naguib, Nitesh Narayan, Bernd Brügge, and Dina Helal. 2013. Bug report assignee recommendation using activity profiles. In *Proceedings of the 10th Working Conference on Mining Software Repositories*. 22–30.

[60] 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.

[61] 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.

[62] Yoon-Joo Park and Alexander Tuzhilin. 2008. The long tail of recommender systems and how to leverage it. In *Proceedings of the 2008 ACM Conference on Recommender Systems*. 11–18. DOI: https://doi.org/10.1145/1454008.1454012

[63] Dipesh Pradhan, Shuai Wang, Shaukat Ali, Tao Yue, and Marius Liaaen. 2019. Employing rule mining and multi-objective search for dynamic test case prioritization. *Journal of Systems and Software* 153 (2019), 86–104.

[64] Lionel P. Robert, Casey Pierce, Liz Marquis, Sangmi Kim, and Rasha Alahmad. 2020. Designing fair AI for managing employees in organizations: A review, critique, and design agenda. *Human Computer Interaction* 35, 5–6 (2020), 545–575.

[65] 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 Transactions on Information Systems* 37, 1 (2019), 2:1–2:46.

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

[67] Latanya Sweeney. 2013. Discrimination in online ad delivery. *Communication of the ACM* 56, 5 (2013), 44–54. DOI: https://doi.org/10.1145/2447976.2447990

[68] Ahmed Tamrawi, Tung Thanh Nguyen, Jafar M. Al-Kofahi, and Tien N. Nguyen. 2011. Fuzzy set and cache-based approach for bug triaging. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*. 365–375.

[69] Ming Tan, Lin Tan, Sashank Dara, and Caleb Mayeux. 2015. Online defect prediction for imbalanced data. In *Proceedings of the 2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. 99–108.

[70] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed Hassan, and Kenichi Matsumoto. 2016. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering* 43, 1 (2016), 1–18.

[71] Mariya I. Vasileva. 2020. The dark side of machine learning algorithms: How and why they can leverage bias, and what can be done to pursue algorithmic fairness. In *Proceedings of the KDD'20: The 26th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*. 3586–3587.

[72] Junjie Wang, Qiang Cui, Qing Wang, and Song Wang. 2016. Towards effectively test report classification to assist crowdsourced testing. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 6.

[73] Junjie Wang, Qiang Cui, Song Wang, and Qing Wang. 2017. Domain adaptation for test report classification in crowdsourced testing. In *Proceedings of the 2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track*. 83–92.

[74] 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, 1 (2019), 139–155.

[75] Junjie Wang, Song Wang, Jianfeng Chen, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2021. Characterizing crowds to better optimize worker recommendation in crowdsourced testing. *IEEE Transactions on Software Engineering* 47, 6 (2021), 1259–1276.

[76] Junjie Wang, Song Wang, Qiang Cui, and Qing Wang. 2016. Local-based active classification of test report to assist crowdsourced testing. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*. 190–201.

[77] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. iSENSE: Completion-Aware crowdtesting management. In *Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering*. 932–943.

[78] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. 2020. Context-aware in-process crowdworker recommendation. In *Proceedings of the 42nd International Conference on Software Engineering*. 1535–1546.

[79] Shuai Wang, Shaukat Ali, Tao Yue, Yan Li, and Marius Liaaen. 2016. A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In *Proceedings of the 38th International Conference on Software Engineering*. 631–642.

[80] Song Wang, Wen Zhang, and Qing Wang. 2014. FixerCache: Unsupervised caching active developers for diverse bug triage. In *Proceedings of the 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 25.

[81] Song Wang, Wen Zhang, Ye Yang, and Qing Wang. 2013. DevNet: Exploring developer collaboration in heterogeneous networks of bug repositories. In *Proceedings of the 2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 193–202.

[82] Honghao Wei, Cheng-Kang Hsieh, Longqi Yang, and Deborah Estrin. 2016. GroupLink: Group event recommendations using personal digital traces. In *Proceedings of the 19th ACM Conference on Computer Supported Cooperative Work and Social Computing*. 110–113.

[83] 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*. 878–888.

[84] Eric W. Weisstein. 2004. *Bonferroni Correction*. Wolfram Research, Inc.

[85] Chuhan Wu, Fangzhao Wu, Xiting Wang, Yongfeng Huang, and Xing Xie. 2021. Fairness-aware news recommendation with decomposed adversarial learning. In *Proceedings of the 35th AAAI Conference on Artificial Intelligence, AAAI 2021, 33rd Conference on Innovative Applications of Artificial Intelligence, IAAI 2021, The 11th Symposium on Educational Advances in Artificial Intelligence*. 4462–4469.

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

[87] 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 Transactions on Software Engineering* 43, 3 (2017), 272–297.

[88] Miao Xie, Qing Wang, Guowei Yang, and Mingshu Li. 2017. COCOON: Crowdsourced testing quality maximization under context coverage constraint. In *Proceedings of the 2017 IEEE 28th International Symposium on Software Reliability Engineering*. 316–327.

[89] Jingwei Xu, Yuan Yao, Hanghang Tong, XianPing Tao, and Jian Lu. 2015. Ice-Breaking: Mitigating cold-start recommendation problem by rating comparison. In *Proceedings of the 24th International Joint Conference on Artificial Intelligence*. 3981–3987.

[90] Jifeng Xuan, He Jiang, Zhilei Ren, and Weiqin Zou. 2012. Developer prioritization in bug repositories. In *Proceedings of the 34th International Conference on Software Engineering*. 25–35.

[91] Hui Yang, Xiaobing Sun, Bin Li, and Yucong Duan. 2016. DR_PSF: Enhancing developer recommendation by leveraging personalized source-code files. In *Proceedings of the 2016 IEEE 40th Annual Computer Software and Applications Conference*, Vol. 1. 239–244.

[92] Ye Yang, Muhammad Rezaul Karim, Razieh Saremi, and Guenther Ruhe. 2016. Who should take this task?: Dynamic decision support for crowd workers. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*. 8.

[93] 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.

[94] Wen Zhang, Song Wang, Ye Yang, and Qing Wang. 2013. Heterogeneous network analysis of developer contribution in bug repositories. In *Proceedings of the 2013 International Conference on Cloud and Service Computing*. 98–105.

[95] 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.

[96] 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.

[97] M. Zhou and A. Mockus. 2012. What make long term contributors: Willingness and opportunity in OSS community. In *Proceedings of the 2012 34th International Conference on Software Engineering*. 518–528.

Received March 2021; revised July 2021; accepted September 2021