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, Beijing, China
YE YANG, School of Systems and Enterprises, Stevens Institute of Technology, Hoboken, NJ, USA
SONG WANG, Lassonde School of Engineering, York University, Canada
JUN HU, Laboratory for Internet Software Technologies, Institute of Software Chinese Academy of Sciences; University of Chinese Academy of Sciences, Beijing, China
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, Beijing, China

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 paper 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 paper 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 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:

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], etc. Unfortunately, these approaches have limited applicability for the highly dynamic and volatile crowdtesting processes. They merely provide one-time recommendation at the beginning of a new task, without considering constantly changing context information of ongoing testing processes.

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

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 paper 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 paper 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 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 paper makes the following contributions:

- The formation of the in-process crowdworker recommendation problem based on the empirical investigation on real-world crowdtesting data. This is the first study to explore the in-process worker recommendation problem to the best of our knowledge.
• The 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.

The paper 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 bug3; and duplicate label indicates with which the report is duplicate. Note that, in the following paper, we refer to "bug report" (also short for "bug") as the report whose bug label is bug, refer to "test report" (also short for "report") as any submitted report, and refer to "unique bug" as the report whose bug label is bug and duplicate label is null.

Crowdworker is a registered worker in a crowdtesting platform, and is denoted by worker ID, and his/her device. It is associated with the historical reports he/she submitted. Note that, in our

1https://github.com/wangjunjieISCAS/InProcessRecommendation
2This paper received an ACM SIGSOFT Distinguished Paper award at ICSE 2020.
3In 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

<table>
<thead>
<tr>
<th>Task ID</th>
<th>T000012</th>
</tr>
</thead>
<tbody>
<tr>
<td>Requirement 1</td>
<td>Browse the videos through list mode IQIYI, rank the videos using different conditions, check whether the rank is reasonable.</td>
</tr>
<tr>
<td>Requirement 2</td>
<td>Cache the video, check whether the caching list is right.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Report ID</th>
<th>R1002948308</th>
</tr>
</thead>
<tbody>
<tr>
<td>Task ID</td>
<td>T000012</td>
</tr>
<tr>
<td>Worker ID</td>
<td>W5124983210</td>
</tr>
<tr>
<td>Description</td>
<td>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.</td>
</tr>
<tr>
<td>Bug label</td>
<td>bug</td>
</tr>
<tr>
<td>Duplicate label</td>
<td>R1002948315, R1002948324</td>
</tr>
<tr>
<td>Submission time</td>
<td>Jan 30, 2016 15:32</td>
</tr>
</tbody>
</table>

| Worker Id       | W5124983210 |
| Device          | Phone type: Samsung SN9009 |
|                 | Operating system: Android 4.4.2 |
|                 | ROM type: KOT49hN9009 |
|                 | Network environment: WIFI |

| Historical Reports | R1002948308, R1037948352 |

experimental dataset which spans across six months, we did not observe the crowdworkers’ device change; thus this paper assumes each crowdworker corresponds to a stable device variable.

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 1a 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 1b. The x-axis shows the length of the non-yielding window, while the y-axis shows the relative position of the non-yielding window expressed using the task’s progress. We can observe that the long-sized non-yielding window is quite common during crowdtesting process. There are 84.5% (538/636) tasks with at least one 10-sized non-yielding window, 67.8% (431/636) tasks with at least one 15-sized window. Furthermore, these long-sized non-yielding windows mainly take place in the second half of crowdtesting processes. For example, 90.7% (488/538) 10-sized non-yielding windows happened at the latter half of the process.

We 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

\(^4\)Following previous work [75, 77], we treat the number of reports as the amount of cost.
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 2a 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

\(^5\)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.
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 2b shows the distribution of crowdworkers’ activity at a finer granularity. The x-axis is the same as Figure 2a, and the y-axis is the random-selected 20 terms (which capture the content under testing) from the top-50 most popular descriptive terms (see Section 3.1 for details). The block in the heat map demonstrates the number of reports which are submitted by the specific worker and contain the specific term. We color-code the blocks, using a darker color to denote a worker submitting reports with corresponding terms more frequently, i.e., worker’s preference in different aspects. The differences across columns in the heat map further reveal the diversified preference across workers. Considering there are usually dozens of crowdtesting tasks open in the platform, even if a crowdworker is active, he/she cannot take all tasks. Intuitively, if a crowdworker has a preference on the specific aspects of a task, he/she would show greater willingness in taking the task and further detecting bugs.

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

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

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 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 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 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 \textit{recPoint}, the following data are obtained for further processing (refer to Section 2.1 for more details of these concepts): 1) \textit{test task}: the specific task currently under testing and recommendation; 2) \textit{test reports}: the set of already received reports for this specific task up till the \textit{recPoint}; 3) all registered \textit{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.

\textbf{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 \textit{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 \textit{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 \textit{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 \textit{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:

\[
\text{TestAdeq}(t_j) = \frac{\text{number of bug reports with } t_j}{\text{number of received bug reports in a task}}
\]  (1)

where \( t_j \in \text{task terms vector} \), 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:

- \( \text{LastBug} \): Duration (in hours) between \( \text{recPoint} \) and the time when worker \( w \)’s last bug is submitted.
- \( \text{LastReport} \): Duration (in hours) between \( \text{recPoint} \) and the time when worker \( w \)’s last report is submitted.
- \( \text{NumBugs-X} \): Number of bugs submitted by worker \( w \) in past \( X \) time, e.g., past 2 weeks.
- \( \text{NumReports-X} \): 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:

\[
\text{ProbPref}(w, t_j) = P(w|t_j) = \frac{tf(w, t_j)}{\sum_{w_k} tf(w_k, t_j)} \cdot \frac{\sum_{w_k} df(w_k)}{df(w)}
\]  (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:

\[
\text{ProbExp}(w, t_j) = P(w|t_j) = \frac{tf(w, t_j)}{\sum_{w_k} tf(w_k, t_j)} \cdot \frac{\sum_{w_k} df(w_k)}{df(w)}
\]  

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

#### Table 2. Features for learning to rank

<table>
<thead>
<tr>
<th>Category</th>
<th>ID</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Activeness indexing</td>
<td>1</td>
<td>LastBug</td>
</tr>
<tr>
<td></td>
<td>2</td>
<td>LastReport</td>
</tr>
<tr>
<td></td>
<td>3-7</td>
<td>NumBugs-8 hours, NumBugs-24 hours, NumBugs-1 week, NumBugs-2 week, NumBugs-all (i.e., in the past)</td>
</tr>
<tr>
<td></td>
<td>8-12</td>
<td>NumReports-8 hours, NumReports-24 hours, NumReports-1 week, NumReports-2 week, NumReports-all (i.e., in the past)</td>
</tr>
<tr>
<td>Preference matching</td>
<td>13-14</td>
<td>Partial-ordered cosine similarity, partial-ordered euclidean similarity between worker’s preference and test adequacy</td>
</tr>
<tr>
<td></td>
<td>15-19</td>
<td>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</td>
</tr>
<tr>
<td>Expertise matching</td>
<td>20-21</td>
<td>Partial-ordered cosine similarity, partial-ordered euclidean similarity between worker’s expertise and test adequacy</td>
</tr>
<tr>
<td></td>
<td>22-26</td>
<td>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</td>
</tr>
</tbody>
</table>

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.

\[
\text{POCosSim} = \frac{\sum x_i * y_i}{\sqrt{\sum x_i^2} \sqrt{\sum y_i^2}}
\]

,where \(x_i\) is \(1.0 - \text{TestAdeq}(t_i)\), \(y_i\) is \(\text{ProbExp}(w, t_i)\), and \(t_i\) is the \(i\)th 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.

\[
\text{POEucSim} = \begin{cases} 
\sqrt{\sum (x_i - y_i)^2}, & \text{if } x_i \geq y_i \\
0, & \text{if } x_i < y_i,
\end{cases}
\]

,where \(x_i\) and \(y_i\) are the same as in POCosSim.

**Partial-ordered jaccard similarity with the cutoff threshold of \(\theta\) (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 \(\theta\).

\[
\text{POJacSim} = \frac{A \cap B}{A}
\]

,where \(A\) is a set of descriptive terms whose \((1.0 - \text{TestAdeq}(t_i))\) is larger than \(\theta\), and \(B\) is a set of descriptive terms whose \(\text{ProbExp}(w, t_i)\) is larger than \(\theta\).

### 3.3.2 Ranking

**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 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.
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 \( R \).

\[
ExpDivDlt(w_i, R) = \sum_{t_j} \text{ProfExp}(w_i, t_j) \times \prod_{w_k \in R} (1.0 - \text{ProfExp}(w_k, t_j))
\]

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.

\[
ExpDiv_{s_j} = \frac{\sum_{i=1}^{n} \text{ExpDivDlt}(w_i, R) \times \frac{n+i+1}{n}}{\sum_{i=1}^{n} \text{ExpDivDlt}(w_i, R)}
\]

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, operating system, etc.) to those of the workers on current re-rank list \( R \), so as to facilitate the exploration in new testing environment.

\[
DevDivDlt(w_i, R) = (w_i’s \ attributes) - \cup_{w_k \in R} (w_k’s \ attributes)
\]

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 \( \text{NumBugs-1 week} \) and \( \text{NumReports-1 week} \) (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

\[
\text{RecFreq}(w_i) = \frac{\# \text{tasks where } w_i \text{ is recommended}}{\# \text{tasks under recommendation in past week}}.
\]

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.

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

(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\textsuperscript{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, etc. The minimum, average, and
maximum number of reports (and unique bugs) per task are 20 (3), 126 (24), and 876 (98) respectively.

4.3 Experimental Setup

To simulate the usage of 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 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 $N$th 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

\textsuperscript{6}test.baidu.com
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 HyperVolume ($HV$), Inverted Generational Distance ($IGD$), and Generalized Spread ($GS$) 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. **Generalized Spread ($GS$)** 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 5a 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., \textit{recThres} is 3, 8, and 10), iRec2.0 can shorten the non-yielding window by 50% to 66%.

Figure 5b to 5e demonstrate the BDR@k of worker recommendation under four representative \textit{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 \textit{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 \textit{recThres}, the advantage of iRec2.0 over current practice is larger. In detail, when \textit{recThres} is 3, iRec2.0 can improve the current practice by 150% (100% vs. 40%) for BDR@20, and when \textit{recThres} is 8, the improvement is 600% (100% vs. 14%). This holds true for other metrics. A larger \textit{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 \textit{recPoint} with larger First Hit of Ground Truth, our proposed approach can shorten the non-yielding window in a larger extent. For example, for the \textit{recPoint} whose First Hit of Ground Truth is 6 (\textit{resThres} is 3), iRec2.0 can shorten the non-yielding window by 50% on median (3 vs. 6), while when \textit{First Hit of Ground Truth} is 12 (\textit{resThres} is 10), the improvement is 66% (4 vs. 12). This further indicates the effectiveness of our approach since for \textit{recPoint} with a larger First Hit of Ground Truth, it is in higher demand for an efficient worker recommendation so that the “right” worker can come soon.
In the following paper, we use the experimental setting when recThres is 5 for further analysis and comparison due to space limit.

**Comparison with Baselines.** Figure 6 demonstrates the comparison results with 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 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 crowdsworkers 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 crowdsworkers, 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 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 crowdsworkers’ 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 be because many crowdsworkers 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 crowdsworkers, 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 crowdsworkers from historical tasks, and has great indicative effect on their bug detection performance on this new task. Without such information, the optimization-based reranking 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, etc.

**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.
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 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.

<table>
<thead>
<tr>
<th>Table 3. Role of re-ranking in bug detection</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>FirstHit</td>
</tr>
<tr>
<td>Average performance</td>
</tr>
<tr>
<td>iRec2.0</td>
</tr>
<tr>
<td>iRec2.0 without re-ranking</td>
</tr>
<tr>
<td>iRec</td>
</tr>
<tr>
<td>Improvement</td>
</tr>
<tr>
<td>iRec2.0 vs. iRec2.0 without re-ranking</td>
</tr>
<tr>
<td>iRec2.0 vs. iRec</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Table 4. Role of re-ranking in alleviating unfairness</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>fairRate@3</td>
</tr>
<tr>
<td>Average performance</td>
</tr>
<tr>
<td>iRec2.0</td>
</tr>
<tr>
<td>iRec2.0 without re-ranking</td>
</tr>
<tr>
<td>iRec</td>
</tr>
<tr>
<td>Improvement</td>
</tr>
<tr>
<td>iRec2.0 vs. iRec2.0 without re-ranking</td>
</tr>
<tr>
<td>iRec2.0 vs. iRec</td>
</tr>
</tbody>
</table>

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., HyperVolume ($HV$), Inverted Generational Distance ($IGD$), and Generalized Spread ($GS$) \cite{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 \cite{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 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 1a. 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 9a, 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 9b, although the bug detection curve can not always dominate the current practice, the first “right” worker can be found earlier than current practice. Similarly, with the dynamic recommendation, the current practice of bug detection can be improved.

<table>
<thead>
<tr>
<th>recThres</th>
<th>1st-quarter</th>
<th>median</th>
<th>3rd-quarter</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4.8%</td>
<td>12.1%</td>
<td>21.3%</td>
</tr>
<tr>
<td>5</td>
<td>4.2%</td>
<td>9.8%</td>
<td>18.6%</td>
</tr>
<tr>
<td>8</td>
<td>2.7%</td>
<td>8.6%</td>
<td>16.7%</td>
</tr>
<tr>
<td>10</td>
<td>2.8%</td>
<td>8.1%</td>
<td>16.4%</td>
</tr>
</tbody>
</table>

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 recommender systems, 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], etc.

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], etc. The aforementioned studies either recommended one worker or assumed the recommended set of workers are independent of each other, which is not applicable for testing activity.

Several studies explored worker recommendation for crowdtesting tasks by modeling the workers’ testing environment [75, 88], experience [22, 88], capability [75], expertise with the task [22, 23, 75], etc. However, these existing worker recommendation solutions only apply at the beginning of the task, and do not consider the dynamic nature of crowdtesting process.

The need for context in software engineering is officially proposed by Prof. Gail Murphy in 2018 [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], etc. 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, etc., 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], etc. This paper 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 paper 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.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under grant No.2018YFB1403400, the National Natural Science Foundation of China under grant No.62072442, No. 62002348, and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] [n.d.]. ([n. d.]).
Wang et al.


[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 *COMPSAC’17*. 75–84.


