10-2018

Perceptions, expectations, and challenges in defect prediction

Zhiyuan WAN

Xin XIA

Ahmed E. HASSAN

David LO
Singapore Management University, davidlo@smu.edu.sg

Jianwei. YIN

See next page for additional authors

DOI: https://doi.org/10.1109/TSE.2018.2877678

Perceptions, Expectations, and Challenges in Defect Prediction

Zhiyuan Wan, Xin Xia, Ahmed E. Hassan, David Lo, Jianwei Yin, and Xiaohu Yang

Abstract—Defect prediction has been an active research area for over four decades. Despite numerous studies on defect prediction, the potential value of defect prediction in practice remains unclear. To address this issue, we performed a mixed qualitative and quantitative study to investigate what practitioners think, behave and expect in contrast to research findings when it comes to defect prediction. We collected hypotheses from open-ended interviews and a literature review of defect prediction papers that were published at ICSE, ESEC/FSE, ASE, TSE and TOSEM in the last 6 years (2012-2017). We then conducted a validation survey where the hypotheses became statements or options of our survey questions. We received 395 responses from practitioners from over 33 countries across five continents. Some of our key findings include: 1) Over 90% of respondents are willing to adopt defect prediction techniques. 2) There exists a disconnect between practitioners' perceptions and well supported research evidence regarding defect density distribution and the relationship between file size and defectiveness. 3) 7.2% of the respondents reveal an inconsistency between their behavior and perception regarding defect prediction. 4) Defect prediction at the feature level is the most preferred level of granularity by practitioners. 5) During bug fixing, more than 40% of the respondents acknowledged that they would make a "work-around" fix rather than correct the actual error-causing code. Through a qualitative analysis of free-form text responses, we identified reasons why practitioners are reluctant to adopt defect prediction tools. We also noted features that practitioners expect defect prediction tools to deliver. Based on our findings, we highlight future research directions and provide recommendations for practitioners.

Index Terms—Defect Prediction, Empirical Study, Practitioner, Survey

1 Introduction

Software quality assurance is a resource-constrained activity due to its conflict with time-to-market requirements [48]. With the growth of software scale and complexity, the cost and duration of quality assurance activities have dramatically increased. Previous studies observed that the majority of defects in a software system are in a small number of software modules [7]. Thus a significant number of prior studies have focused on the prioritization of software quality efforts.

A key focus, defect prediction, has emerged as an active research area for over four decades [2]. Defect prediction techniques build models based on various types of metrics (e.g., code, process, and developer [27], [83], [90]) and predict defects at different granularity levels, e.g., change, file, or module levels. These techniques can be used to effectively allocate quality assurance resources [79].

Despite numerous defect prediction studies, the potential value of defect prediction in practice remains unclear. Addressing this issue can provide insight for both practitioners and researchers. Practitioners can use empirical evidence on defect prediction to make informed decisions about when to use defect prediction and how it would best fit into their development process. Researchers can enhance defect prediction techniques based on the expectations of practitioners and adoption challenges that they face.

To gain insights into the practical value of defect prediction, we performed a mixed qualitative and quantitative study to investigate what practitioners think, behave and expect in contrast to research findings when it comes to defect prediction. The process is as follows:

1) We started with open-ended interviews with 16 senior software practitioners. Through the interviews, we qualitatively investigated the factors that might affect practitioners' willingness to adopt defect prediction techniques.
2) To explore the hypotheses behind existing defect prediction research, we performed a literature review. We first collected the titles of papers that were published at ICSE, ESEC/FSE, ASE, TSE and TOSEM during a time period of 6 years (2012 - 2017) from the DBLP computer science bibliography[1]. We then used the keywords "defect", "fault", "bug" and "predict" to search for papers related to software defect prediction, and downloaded relevant papers from the ACM or IEEE digital libraries. We read the abstract of each paper to validate its relevance. Finally, we identified 41 defect prediction papers. We further went through these identified papers and extracted the hypotheses behind

- *Zhiyuan Wan, Jianwei Yin, and Xiaohu Yang are with the College of Computer Science and Technology, Zhejiang University, Hangzhou, China.*
  E-mail: wanzhiyuan@zju.edu.cn, zjuyjw@cs.zju.edu.cn, yangxh@zju.edu.cn
- *Xin Xia is with the Faculty of Information Technology, Monash University, Melbourne, Australia.*
  E-mail: xin.xia@monash.edu
- *Ahmed E. Hassan is with School of Computing, Queen's University, Canada.*
  E-mail: ahmed@cs.queensu.ca
- *David Lo is with the School of Information Systems, Singapore Management University, Singapore.*
  E-mail: davidlo@smu.edu.sg
- *Xin Xia is the corresponding author.*

Manuscript received ; revised

1. http://dblp.uni-trier.de

the defect prediction techniques as proposed in these papers.

3) We performed a survey to validate the hypotheses that were uncovered in our interviews and literature review. The hypotheses became the statements that we asked our survey respondents to rate and the options for multiple-choice questions in our survey. We invited thousands of practitioners from various backgrounds through emails, including those who work in corporations and those who contribute to open source projects hosted on GitHub. We received 395 responses from 33 countries across five continents.

Our survey revolves around the following research questions:

- **RQ1.** Will practitioners adopt defect prediction techinques? Does practitioners' willingness to adopt vary across different demographics, e.g., job roles, experience levels, project sizes and major programming language?
- **RQ2.** How do practitioners perceive a number of factors related to defect prediction? We consider the following factors: strategies of prioritizing code inspection/testing effort, defect density distribution, relationship between file size and defect proneness, as well as metrics that are associated with defect proneness.
- **RQ3.** What are practitioners' expectations regarding defect prediction, i.e., granularity level on which a defect prediction technique works and the expected features of a defect prediction tool?
- **RQ4.** Which factors do practitioners consider as the main adoption challenges? The challenges can influence the performance of a defect prediction technique and hinder practitioners from adopting it in practice.

We consider **RQ1** to understand the willingness of practitioners to adopt a defect prediction technique and understand how various demographic factors influence their willingness of adoption. **RQ2** investigates practitioners' perceptions regarding defect prediction. Answers to **RQ2** enable us to discover disconnects between empirical evidence in prior studies and practitioners' perceptions. We check if empirical evidence goes against practitioners' intuition. Connection between intuition and empirical evidence highlights the importance and value of such research. In contrast, disconnect indicates a problem of adoption since people might be more apprehensive of things that go against their prior beliefs and intuitions. **RQ3** explores what practitioners expect from defect prediction. Answers to **RQ3** help better understand the disconnect between research and practitioners' expectation. In **RQ4**, we consider the practitioners' behavior during bug fixing that may affect the performance of a defect prediction technique. We also consider the preferred measures of performance evaluation and adoption barriers. Answers to **RQ4** can help researchers find areas for improvement for existing defect prediction techniques so they can increase the likelihood of them being adopted by practitioners.

Some of our key findings include: 1) Over 90% of respondents are willing to adopt defect prediction techniques. 2) There exists a disconnect between practitioners' perceptions and research evidence regarding defect density distribution as well as the relationship between file size and defectiveness. 3) 7.2% of the respondents reveal an inconsistency between their behavior and perception regarding defect prediction. 4) Defect prediction at the feature level (i.e., identifying defective features[2] in a software system) is the most preferred level of granularity by practitioners. 5) During bug fixing, more than 40% of respondents acknowledged that they would make a "work-around" fix rather than correct the actual error-causing code.

This paper makes the following contributions:

1) We perform a mixed qualitative and quantitative study to investigate practitioners' willingness of adoption, perceptions, expectations and adoption challenges regarding defect prediction.
2) We present our findings from a validation survey of 395 practitioners from 33 countries across five continents.
3) We explore the factors that affect practitioners' willingness of adoption of defect prediction tools.
4) We recognize disconnect between research and practitioners' perceptions regarding defect prediction.
5) We provide implications for researchers and outline future avenues of research considering practitioners' expectations and their perceived challenges regarding defect prediction.

The remainder of the paper is structured as follows. In Section 2, we describe the methodology of our study in detail. In Section 3, we present the results of our study. In Section 4, we discuss the implications of our results as well as any threats to validity of our findings. In Section 5, we briefly review related work. Section 6 draws conclusions and outlines avenues for future work.

2 Methodology

2.1 Overview

Our research methodology followed a mixed qualitative and quantitative approach as depicted in Fig. 1. We collected data from different sources: (1) We interviewed 16 senior software practitioners; (2) We conducted a literature review of recent defect prediction studies; (3) We surveyed 395 respondents. The procedure and transcripts of our interviews, as well as questionnaire and responses of our survey are publicly available at https://github.com/shengying/defect_prediction_survey.

2.2 Open-Ended Interviews

We present the protocol, participants selection, and data analysis of our open-ended interviews as follows.

2.2.1 Protocol

The first author conducted 16 face-to-face interviews with 16 senior software practitioners, each interview taking 30-45 minutes. The interviews were semi-structured and made use of *an interview guide*. The guide contained general groupings of topics and questions, rather than a pre-determined exact set and order of questions. We iteratively refined the guide after each interview by clarifying unclear questions and making them easy to understand in subsequent interviews.

2. A feature is a unit of functionality of a software system that satisfies a requirement and represents a design decision [4].

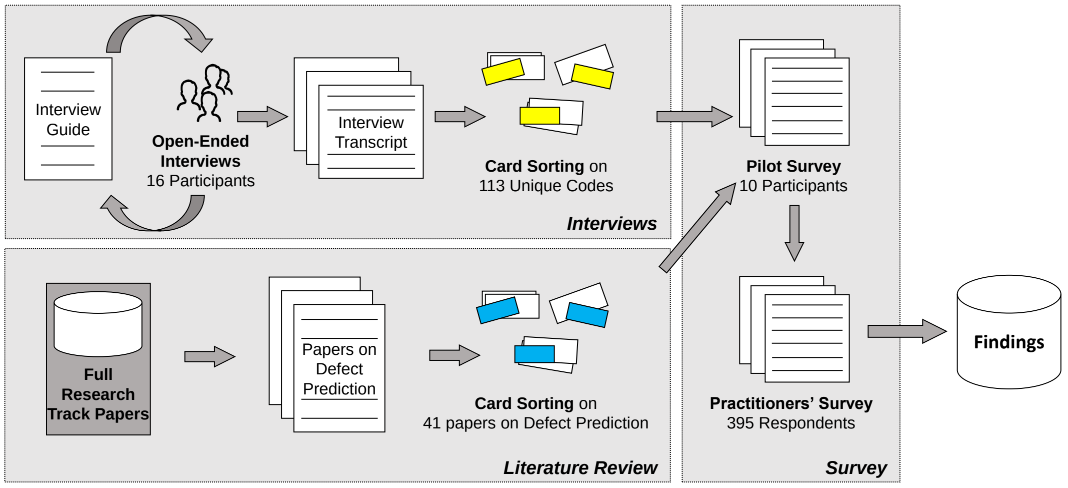


Fig. 1: Research methodology.

The interview was comprised of four parts. In the first part, we introduced the concepts of defect prediction, and ensured that the interviewees understood how defect prediction techniques work and how they can help software practitioners. In the second part, we asked some demographic questions about the experience of the interviewees in software development/testing/project management, and about projects in which they were recently involved.

In the third part, we asked open-ended questions to understand interviewees' experience and expectations regarding defect prediction techniques. The purpose of this part was to allow the interviewees to speak freely about defect prediction without any bias. Specifically, we asked questions as follows:

1) Do you know about research regarding defect prediction? Have you ever used any defect prediction technique?
2) How do you decide which file needs more code inspection/testing effort? Which factors do you consider?
3) Which granularity levels (e.g., method, file, component) do you work on to prioritize your code inspection/testing effort?
4) Which metrics do you consider when looking for the most defect-prone entities in your code base?
5) Which factors affect your adoption of a defect prediction technique?

In the fourth part, we picked a list of factors related to tool adoption. We asked the interviewees to discuss and rank the factors that they have not explicitly mentioned. To comprehensively investigate factors of adoption, we follow *diffusion of innovation* (DOI) research [74]. DOI is a field that investigate how innovations spread, how individuals make adoption decisions, and what factors promote or prevent adoption. Since 1960s, DOI has been applied to the adoption of a number of technologies, as diverse as personal workstations, corn breeds, and dance moves. Our work draws upon parts of the theory that name factors influencing adoption as a part in our interview. The factors include:

- Complexity: how complex is the tool?
- Compatibility: how compatible is the tool with the working environment of a practitioner?
- Trialability: how easy can a practitioner try the tool out?
- Relative advantage: what advantages does the tool offer a practitioner over other tools?
- Reinvention: can a practitioner configure or customize the tool?

At the end of each interview, we thanked the interviewee and briefly informed him/her of our next plans.

2.2.2 Participant Selection

We selected full-time employees from two IT companies in China, namely Insigma Global Service (IGS)[3] and Hengtian[4], as interview participants. We noticed that a recent work also ran a case study and surveyed software engineers within a single company to understand their perceptions about working in monolithic repos [31]. IGS and Hengtian are two outsourcing companies which have more than 500 and 2,000 employees, respectively. IGS mainly works on outsourcing projects for Chinese vendors (e.g., Chinese commercial banks, Alibaba, and Baidu). Hengtian focuses on outsourcing projects from US and European corporations (e.g., State Street Bank, Cisco, and Reuters). Note that in these two companies, around 60% of projects are developed onsite, i.e, the project teams work in the site of client. We cannot perform a face-to-face interview with employees that are currently working in onsite projects. Thus, we removed the onsite employees from the interviewee pool. Interviewees were recruited by emailing each project leader, who was then responsible for disseminating news of our study to senior technical employees working on the projects. Volunteers would inform us if they were willing to participate in the study with no compensation. We ended up having 19 volunteers that contacted us with varied job roles, including development, testing, project management, and business analyst.

We recruited 16 interviewees with varied job roles, including development, testing, and project management.

3. http://www.insigmaservice.com
4. http://www.hengtiansoft.com/en

TABLE 1: Professional experience (in years) and role of the 16 interviewees.

ID	Experience	Role
P1	8	Development
P2	6	Testing
P3	7	Project Management
P4	3	Development
P5	10	Development
P6	11	Development
P7	10	Development
P8	7	Project Management
P9	2	Development
P10	5	Testing
P11	4	Testing
P12	6	Testing
P13	3	Testing
P14	10	Development
P15	12	Development
P16	10	Development

The interviewees participated in various projects located in United States, Canada, Japan, Ireland and Germany. The interviewees need to follow the development practices and leverage the development tools and techniques requested by the project clients. To ensure each interviewee had enough knowledge to answer our questions, we restrict the interviewees to employees with technical roles (no sales or marketing employees). In the remainder of the paper, we denote these 16 interviewees as P1 to P16.

Table 1 presents the number of years of professional experience that each of the 16 interviewees had, along with their job roles. These 16 interviewees have a varying number of years of professional experience, ranging from 3 to 12 years. The average number of years that an interviewee worked as a professional software practitioner is 6.5 years. The interviewees had diverse background and experience on different types of projects. The diversity of our interviewees helps improve the generalizability of our findings.

2.2.3 Data Analysis

To collect interviewees' perceptions, expectations and perceived adoption challenges regarding defect prediction, we processed the recorded interviews by following the steps below:

Transcribing and Coding. After the last interview was completed, we transcribed the recordings of the interviews. The first author read the transcripts and coded the interviews using the QDA Miner Lite qualitative analysis software [1]. To ensure the quality of codes, the second author verified initial codes created by the first author and provided suggestions for improvement. After incorporating these suggestions, we generated a total of 338 cards that contain the codes - 18 to 25 cards for each coded interview. After merging the codes with same words or meanings, we have a total of 133 unique codes. We noticed that when our interviews were drawing to a close, the collected codes from interview transcripts reached a saturation. New codes did not appear anymore; the list of codes was considered stable.

Open Card Sorting. The first and second authors then separately categorized the generated cards for thematic similarity (as illustrated in LaToza et al.'s study [47]). The themes that emerged during the sorting were not chosen beforehand. We then use the Fleiss Kappa measure [19] to examine the agreement between the two labelers. The interpretation of the Kappa measure is shown in Table 2. The overall Kappa value between the two labelers is 0.63, which indicates substantial agreement between the labelers. After completing the labeling process, the two labelers discussed their disagreements to reach a common decision. To reduce bias from the first and second authors sorting the cards to form initial themes, they both reviewed and agreed on the final set of categories.

TABLE 2: Interpretation of Kappa values.

Kappa Value	Interpretation
< 0.00	Poor Agreement
$[0.00, 0.20]$	Slight Agreement
$[0.21, 0.40]$	Fair Agreement
$[0.41, 0.60]$	Moderate Agreement
$[0.61, 0.80]$	Substantial Agreement
$[0.81, 1.00]$	Almost perfect Agreement

2.3 Literature Review

To explore the hypotheses behind prior defect prediction research, we performed a literature review by following the steps below.

Paper Selection. First, we collected the titles of full research track papers that were published in six venues from 2012 to 2017 - the ACM/IEEE International Conference on Software Engineering, the ACM SIGSOFT Symposium on Foundations of Software Engineering, the Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on Foundations of Software Engineering, the IEEE/ACM International Conference on Automated Software Engineering, the IEEE Transactions on Software Engineering, and the ACM Transactions on Software Engineering Methodology - from the DBLP computer science bibliography. These 4 conferences and 2 journals are premier publication venues in the software engineering research community. We believe that state-of-the-art findings are likely to be published in these conferences and journals. Second, we used the "defect", "fault", "bug" and "predict" keywords to search for papers regarding software defect prediction, and downloaded relevant papers from the ACM or IEEE digital libraries. Third, we read the titles and abstracts of these papers and judged whether each paper is related to defect prediction. Finally, we identified 14, 6, 4, 15, and 2 papers from ICSE, ESEC/FSE, ASE conferences, and TSE and TOSEM journals, respectively, for a total of 41 papers.

Categorization and Summarization. We grouped the identified 41 papers with similar topics and identified 7 categories (followed by identified papers):

- Prediction models and learning approaches ([14], [20], [21], [32], [35], [53], [75], [76], [84], [86])
- Metrics in defect prediction models ([8], [13], [25], [48], [78], [90], [96], [98], [99], [102])
- Cross-project/cross-company defect prediction ([33], [60]–[62], [65], [71], [94], [100])
- Imbalance, mislabeling, bias and noise ([30], [34], [72], [82], [83])
- Granularity of defect prediction ([27], [37], [97])
- Privacy ([64], [66])
- Performance evaluation ([49], [57])
- Literature review ([24])

For each defect prediction paper, we summarized the main hypotheses behind the proposed technique, e.g., proposed metrics in the prediction models, employed measures to evaluate performance, and defect characteristics. Later, we validated these hypothesis using an online survey – turning these hypotheses into statements for our survey respondents to rate or options for multiple-choice questions in our survey.

2.4 Validation Survey

The goal of the final step is to validate the hypotheses that emerged from the previous steps. Towards this goal, we created an online survey to gather opinions from a wide range of respondents. We followed Kitchenham and Pfleeger's guidelines for personal opinion surveys [39] and used an anonymous survey to increase response rates [88]. A respondent has the option to specify that he/she prefers not to answer or does not understand the description of a particular question. We include this option to reduce the possibility of respondents providing arbitrary answers.

Before sending our survey to a large number of potential respondents, we piloted the survey with a small set of practitioners as pilot respondents. The pilot respondents gave feedback on (1) whether the length of the survey is appropriate, and (2) the clarity and understandability of the terms. We made minor modifications to the preliminary survey based on the received feedback and produced a final version. Note that the collected responses from the pilot survey are excluded from the presented results in this paper.

To support respondents from China, we translated our survey to Chinese before publishing the survey. We chose to make our survey available both in Chinese and English because Chinese is the most spoken language and English is an international lingua franca. We expect that a large number of our survey recipients are fluent in one of these two languages. We carefully translated our survey to make sure there exists no ambiguity between English and Chinese terms in our survey. In addition, we polished the translation by improving clarity and understandability according to the feedbacks from our pilot survey.

To prevent the respondents from being confused about defect prediction tools and static analysis tools, at the beginning of the survey we gave a definition of defect prediction and explained the differences between defect prediction and static analysis. Much of our results show overall strong trends and our analysis is focused on such trends instead of specific answers protecting us from some possible misunderstanding by a small number of respondents (something that is possible with any type of survey).

2.4.1 Survey Design

We capture the following pieces of information:

Demographics:

- *Professional software engineer*: Yes / No
- *Involvement in open source development*: Yes / No
- *Role*: Development / Testing / Project Management / Other (Pick all that apply)
- *Experience in years* (decimal value)
- *Current country of residence*

Collecting demographic information about the respondents allows us to: (1) filter respondents who may not understand our survey (i.e., respondents with less relevant job roles), (2) breakdown the results by groups (e.g., developers, testers, etc).

Most recent project:

- *People involved*: 1-5 / 6-10/ 11-20 / 20-40 / Other (Pick one)
- *Primary programming language*
- *Usage of static analysis tools*: Yes / No

Project size and primary programming language may play a role in how people decide to perform quality assurance activities. The usage of static analysis tools may also shed light on the willingness of a practitioner to adopt defect prediction tools.

Static analysis and defect prediction have emerged from parallel and disparate traditions of intellectual thought [70]: one driven by algorithm and abstraction over code, and other by statistical methods over large defect datasets. In practice, the two approaches tackle the same problem: improving inspection efficiency, finding minimal, potentially defective regions in source code.

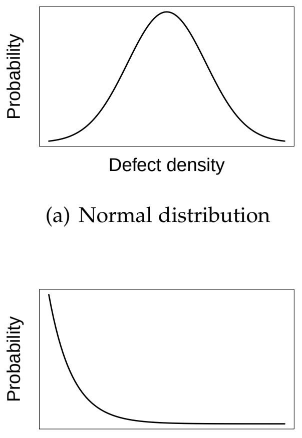
Compared with static analysis tools, defect prediction tools are independent of language and build procedures, and easy to implement: once adequate history is available, programming language, build environment, platform evolution are immaterial. In contract, static analysis tools are language and platform specific, and can be very difficult to deploy. Prior work found that static analysis and defect prediction tools do find different defects in some cases [70]. Thus it might be worthwhile to use both in practice.

Willingness of adoption:

To understand practitioners' willingness of adoption, we asked respondents if they would adopt a defect prediction tool. To prevent bias due to respondents' unfamiliarity with defect prediction, we provided a description about defect prediction, including typical usage scenarios, input and output of defect prediction. To make the description easy to understand, we polished the description according to the feedbacks during the pilot survey. In addition, to reduce the impact of a poor understanding of defect prediction from respondents, we provided an "I don't know" option. For the "unwilling" responses, we asked a follow-up question: *why are you unwilling to adopt a defect prediction tool?* To identify common observations, we took the responses and performed an open card sort to cluster the reasons into groups.

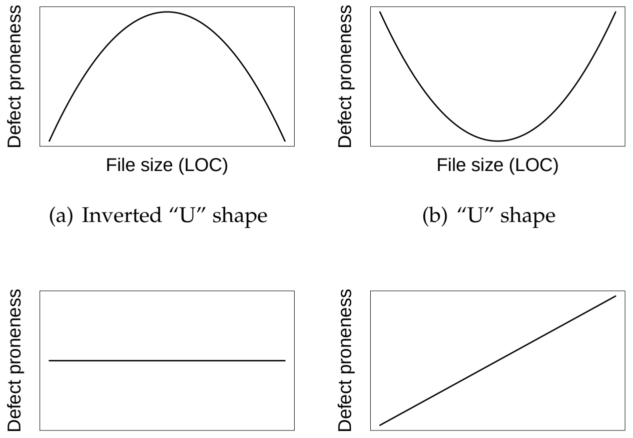
Practitioners' perceptions:

Prioritizing resources for code inspection/testing. To understand how practitioners prioritize resources for code inspection/testing, we provided respondents with 10 strategies of how they could prioritize resources for code inspection/testing. The 10 strategies were derived from file related metrics in our literature review. Note that similar types of metrics can be extracted at different granularity levels, e.g., component, method and commit. We only provided file related strategies in order to focus on respondents' perceptions about metrics in defect prediction. We explored



(a) Normal distribution (b) Uniform distribution (c) Long tail distribution

Fig. 2: Probability density function curves of defect density distributions (defect density is the number of defects per file).



(a) Inverted "U" shape (b) "U" shape (c) Constant (d) Linear growth

Fig. 3: Relations between file size (LOC) and defect proneness.

practitioners' expected granularity levels of defect prediction in subsequent "practitioners' expectations" piece.

We asked the respondents to rank the frequency with which they follow these strategies using the following ratings: *very often*, *often*, *sometimes*, *rarely*, and *very rarely*. The strategies include:

- Most recent buggy file
- Most recently changed file
- File changed by more developers
- File changed by fewer developers
- File with more LOCs
- File with fewer LOCs
- File created/changed by junior developers or newcomers
- File created/changed by senior developers
- File called by many other files
- File that calls many other files

To capture the origins behind the use of such strategies, we asked respondents to rank possible factors that influence their opinion formation:

> *What factors played a role in your previous answer? Please choose the relevant factors from the list below, and rank them from most to least important.*

Respondents were given a choice of "*personal experience*", "*what I hear from my peers*", "*what I hear from my mentors/managers*", "*articles in industry magazines*", "*research papers*", and "*other*". We then gathered the ranks given for each of the above factors. Thus, we had a collection of factor ranks for the prioritization strategies – the lower the ranks, the more important a strategy is rated.

Defect density distribution. Defect density is an important indicator of software quality. A good understanding of defect density allows software practitioners to invest resources proactively and efficiently to improve software quality before delivery. To capture practitioners' perceptions of defect distribution in a project, we asked respondents to select a curve of probability density function that matched their perception. The options of distributions include *normal distribution*, *uniform distribution*, *long tail distribution*, and *other* as shown in Fig. 2. If the *other* option is selected, we asked the respondent to provide a specific distribution. To capture the reasons behind a selection, we included a follow-up question and asked respondents to elaborate the reasons behind their choice. In addition, we checked if respondents' perceptions on defect density connect with research and explored how respondents' experience affects their perceptions on defect density.

File size vs. defect proneness. A good understanding of the size-defect correlation is essential for software practitioners [81]. We asked respondents to select a curve that represents the relationship between file size (measured in terms of lines of code) and defect proneness. We used the relationships between file size and defect proneness as described in Syer et al.'s work [81] as options in our survey, i.e., *inverted "U" shape* and *"U" shape*. We also used the relationships mentioned in our interviews as options, i.e., *constant* and *linear growth*. The curves of the relationships are shown in Fig. 3. In addition, we provided an *other* option. If the *other* option is selected, we asked the respondent to provide a specific relationship. To capture the reason behind the selections, we included a follow-up question and asked respondents to elaborate the reasons behind their choice. In addition, we checked if respondents' perceptions on size-defect correlation connect with research and explored how respondents' experience affects their perceptions on size-defect correlation.

Defect prediction metrics. To understand the metrics that practitioners deemed effective to identify defect proneness, we provided respondents some statements about defect proneness metrics that are extracted from our literature review. The metrics are divided into 3 groups, code metrics [90], process metrics [83] and ownership metrics [27], [83], respectively. For each of these statements, we asked practitioners to respond on a 5-point Likert scale (*strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree*). We also asked respondents to explain the origins of their view as we do for the first question. Furthermore, we asked respondents to rank the groups of metrics according to their perceived impact on defect proneness.

Practitioners' expectations:

Granularity levels. To understand the preferred granularity levels of a defect prediction tool, we asked respondents to select one or more from a list of granularity levels. The list of granularity levels is extracted from our interviews and literature review. We ranked the list of granularity levels from most coarse-grained to most fine-grained, i.e., *feature level*, *component level*, *file level*, *method level*, *commit level* and *session level*[5]. Accordingly, the options include *identify buggy feature*, *identify buggy component*, *identify buggy file*, *identify buggy method*, *identify buggy commit* and *identify buggy session*. An *other* option was provided if respondents preferred other granularity level not included in our list. If the *other* option is selected, we asked the respondent to provide a specific granularity level.

Expected features. We asked respondents to enumerate the expected features of a defect prediction tool. Respondents could enter free-form text to express their thoughts; the question was optional.

Challenges:

Bug fixing. Defect prediction techniques usually leverage previous bug reports and related code changes and fixes. Their performance is validated not on the actual defects but on bug fixes [46]. Therefore, models of how developers fixed bugs may affect the performance of defect prediction techniques. To learn practitioners' bug fixing models, we provided respondents with some statements about bug fixing as included in Murphy-Hill et al.'s work [58]. These include the following statements:

1) *Sometimes the real error lies too deep. So risk of introducing new errors is too high to solve the real error;*
2) *Some defects are not fixed by correcting the "real" error-causing component, but rather by a "work-around" somewhere else.*

For each of these statements, we asked practitioners to respond by providing a rating on a 5-point Likert scale (*strongly disagree*, *disagree*, *neutral*, *agree*, *strongly agree*).

Performance evaluation. To understand practitioners' preferred evaluation measures, we provided respondents with some options that were extracted from our previous interviews and literature review. We asked respondents to pick all the applicable options, including *false alarm rate*, *recall*, *combination of false alarm rate and recall*, *top k% LOC precision*, *initial false alarm count*, *other* and *I don't know*.

Adoption barrier. We asked respondents what they believe are the barriers for wide adoption of defect prediction tools. The options include the barriers that were identified in our previous interviews, for instance: *cost of collecting historic data*, *lack of IDE integration*, *lack of code review tool integration*, and *lack of continuous integration support*. For integration into code review tool, interviewees mentioned that it is necessary to allow the outputs of defect prediction techniques to be used to prioritize code inspection effort, which is a valuable use case of defect prediction. For support of continuous integration, interviewees mentioned that it is necessary to enable defect prediction tools to run automatically and provide continuous results of defect proneness. An *other* option is provided for respondents to specify other barriers.

5. A session is a period of time between two successive "*save*"s by a developer.

2.4.2 Recruitment of Respondents

In order to get a sufficient number of respondents from diverse backgrounds, we followed a multi-pronged strategy to recruit respondents:

- We contacted professionals from various countries and IT companies and asked their help to disseminate our survey within their organizations. We sent emails to our contacts in Microsoft, Amazon, Google, Baidu, IBM, Morgan Stanley, Hengtian, IGS and other companies from various locations around the world, encouraging them to complete the survey and disseminate it to some of their colleagues. By following this strategy, we hope to recruit respondents working in industry from diverse organizations.
- We sent an email with a link to the survey to 4,850 practitioners that contribute to open source projects on GitHub and solicited their participation. Out of these emails, 17 emails received automatic replies notifying us of the absence of the receiver. By sending to GitHub developers, we hope to recruit respondents who are open source practitioners in addition to professionals working in industry.

No identifying information was required or gathered from respondents. Separate from their survey responses, respondents could enter their email addresses into a raffle to win two \$50 Amazon gift cards.

2.4.3 Data Analysis

We computed several statistics to answer our research questions. For use of prioritization strategies, we first converted the ratings into scores. Specifically, we converted *very rarely*, *rarely*, *sometimes*, *often* and *very often* to 1, 2, 3, 4 and 5 respectively. We then computed the mean and standard deviation for all the scores. For ratings of statements, we calculated the percentage of respondents who strongly agree or agree with each statement ($\%\ strongly\ agree + \%\ agree$). We also computed a conflict factor for each statement. The conflict factor is a measure of disagreement between respondents, which is calculated for each statement by the following equation: $(\%\ strongly\ agree + \%\ agree)/(\%\ strongly\ disagree + \%\ disagree)$.

To group free-form answers of questions, we performed an open card sort [80]. Our card sort consisted of two phases: in the preparation phase, we created one card for each response to a question. In the execution phase, cards were sorted into meaningful groups with a descriptive title. We had no predefined groups; instead, we let the groups emerge and evolve throughout the sorting process.

3 Results

In this section, we present the results of our study.

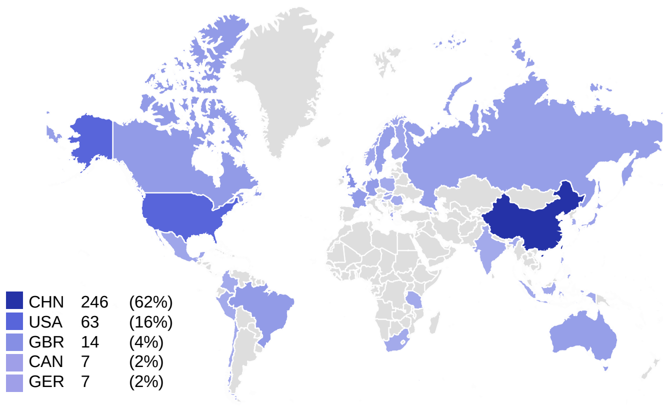


Fig. 4: Countries in which survey respondents reside. The darker the color is, the more respondents that reside in that country. The legend presents the top 5 countries with most respondents.

3.1 Overall Impressions

We received a total of 402 responses, and the respondents reside in 33 countries across five continents as shown in Fig. 4. The top two countries in which the respondents reside are China and the United States.

We excluded 7 responses made by respondents whose job roles are neither development, testing nor project management. Those respondents described their job roles as: vulnerability analyst[6] (1), business analyst (2), data scientist (1), network infrastructure (1), and legal affairs (1). At the end, we had a set of 395 responses.

The number of years of professional experience of the respondents varied from 1 year to 29 years, with an average of 6.8 years. Among the respondents, 320 (81.0%), 162 (41.0%) and 85 (21.5%) described their job roles as development, testing, and project management respectively. Note that the percentages do not add up to 100% since some respondents might perform multiple roles, especially for respondents in small to medium sized corporations, or respondents from open-source projects. The job role combinations of the respondents include “development + project management”, “development + testing”, “development + testing + project management”, “testing + project management”. Thus, we have seven groups of job roles for our respondents.

3.2 Willingness to Adopt

Fig. 5 shows the percentage of practitioners who are willing/unwilling to adopt defect prediction tools across different *demographics*. We consider the following demographic groups:

- All respondents (All),
- Respondents who are developers (Dev),
- Respondents who are testers (Test),
- Respondents who are project managers (PM),
- Respondents who are developers and testers (Dev+Test),
- Respondents who are developers and project managers (Dev+PM),
- Respondents who are testers and project managers (Test+PM),
- Respondents who are developers, testers and project managers (Dev+Test+PM),
- Respondents with low experience, which we define as the bottom 25% with the least experience in years and worked for 1-2 years (ExpLow),
- Respondents with medium experience and worked for 2-10 years (ExpMed),
- Respondents with high experience, which we define as the top 25% with the most experience in years and worked for 10-29 years (ExpHigh),
- Respondents who recently worked in a project with 1-5 members (SizeT),
- Respondents who recently worked in a project with 6-10 members (SizeS),
- Respondents who recently worked in a project with 11-20 members (SizeM),
- Respondents who recently worked in a project with 21-40 members (SizeL),
- Respondents who recently worked in a project with more than 40 members (SizeH),
- Respondents who use static analysis tools (StaticY),
- Respondents who do not use static analysis tools (StaticN).

From Fig. 5, we observe that across all demographics more respondents are willing to adopt a defect prediction tool as compared to those who are not willing to. Across all respondents, only 7.8% are unwilling to adopt defect prediction tools. Furthermore, we observe several differences among the demographic groups:

- There is little difference across various job roles of *development*, *development + testing*, *development + project management*, and *development + testing + project management*. Testers are slightly more willing to adopt a defect prediction tool than other job roles. Respondents with job roles of *project management* and *testing + project management* are more unwilling to adopt a defect prediction tool than those with other job roles. To check whether the differences of willingness across the seven job roles are statistically significant, we performed a Fisher’s exact test [18]. The null hypothesis is that respondents with different roles are equally willing to adopt defect prediction tools. We found no statistically significant difference (p-values max: 1, min: 0.07, median: 0.18, mean: 0.37).
- Respondents with the most experience are least willing to adopt a defect prediction tool. We can especially observe a sharp increase in the percentage of unwilling respondents between ExpMed and ExpHigh groups. Again, we performed a Fisher’s exact test. The null hypothesis is that the ExpMed and ExpHigh groups are equally willing to adopt defect prediction tools. We found that the difference between ExpHigh and ExpMed is significant (p-value = 0.0001).
- There is little difference across various project sizes. As project size increases, the unwillingness of adoption among respondents slightly decreases. The null hypoth-

6. The main duties of a vulnerability analyst include writing vulnerability reports and analyzing software vulnerabilities using black box testing, source code examination, and attack reproduction. Thus, we do not consider a vulnerability analyst to be a developer. (https://www.appone.com/maininforeq.asp?R_ID=945477)

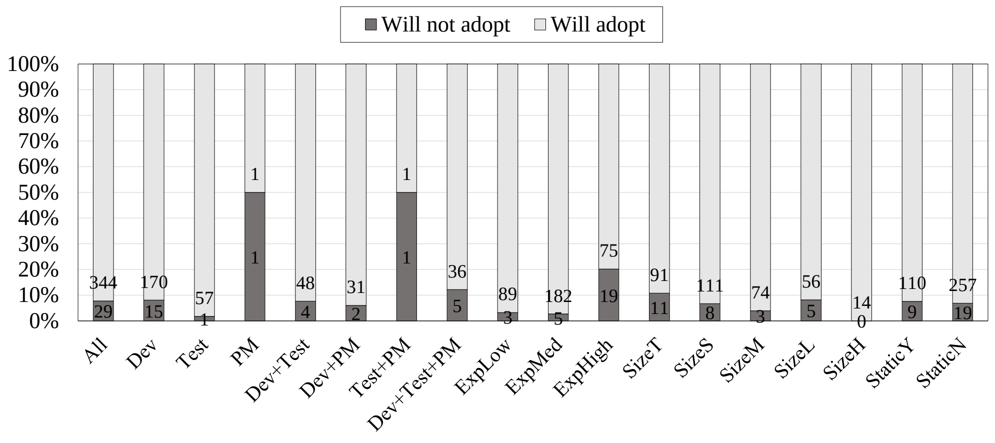


Fig. 5: Percentage of respondents who are willing/unwilling to adopt defect prediction tools across various demographics.

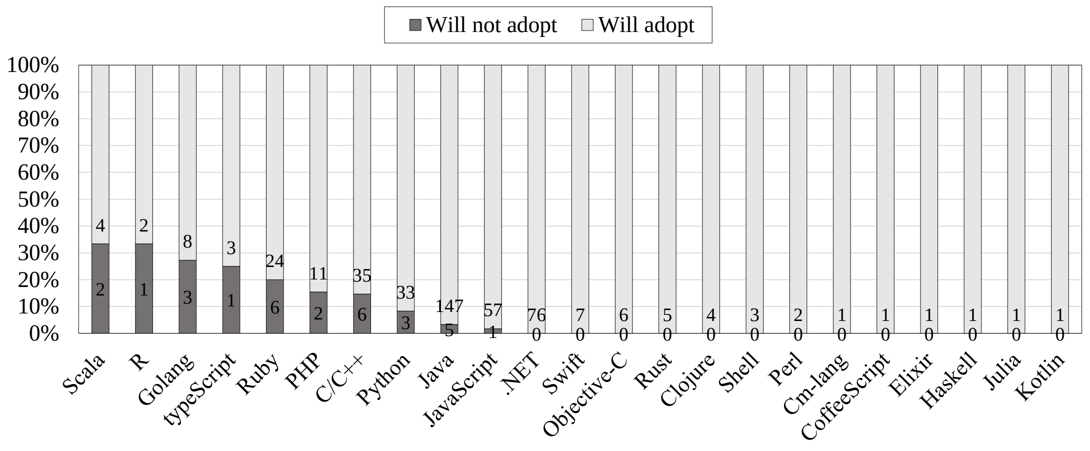


Fig. 6: Percentage of respondents who are willing/unwilling to adopt defect prediction tools across various programming languages.

esis is that the respondents across various project sizes are equally willing to adopt defect prediction tools. A Fisher's exact test indicates no statistically significant difference (p-values max: 1, min: 0.10, median: 0.53, mean: 0.57).

- There is little difference between respondents who use or do not use static analysis tools. The null hypothesis is that the respondents who use or do not use static analysis tools are equally willing to adopt defect prediction tools. A Fisher's exact test indicates no statistically significant difference (p-value = 0.83).

Fig. 6 shows the percentage of practitioners who are willing/unwilling to adopt defect prediction tools across various programming languages. We observe that practitioners who use Scala, R and golang as their primary programming languages are the top-3 groups who are unwilling to adopt a defect prediction tool. Scala is a Java-like programming language which unifies object-oriented and functional programming. R is an interpreted language and widely used for statistics and data analysis. Golang is a statically typed language with garbage collection and memory safety. The three programming languages are quite new since they are released in the early 21st century. We have a few respondents who use these languages as their primary programming languages. We performed a Fisher's exact test [18] with the null hypothesis - the respondents who use Scala, R and golang as primary programming languages are equally willing to adopt defect prediction tools. The result shows no statistically significant difference due to using Scala, R and golang as primary programming languages in willingness to adopt defect prediction tools (p-value = 0.1258, 0.4777, 0.0675).

To investigate the impact of various demographic factors to adoption willingness, we built a decision tree classifier by using the function `ctree` in the `party` R package. The constructed decision tree is shown in Fig. 7. Note that the respondents do not add up to 395 since we dropped respondents who do not provide specific programming languages. From the decision tree, we notice that high-experience respondents who use C/C++ as their primary programming languages account for the majority of the unwilling population. High-experience respondents are likely to have a deep understanding of their projects and may tend to trust their intuition rather than a defect prediction tool. C/C++ provides facilities for low-level memory manipulation, which is usually used for system programming with performance as a design highlight. Some respondents mentioned that "*projects written in C/C++ tend to be too complicated for a defect prediction tool to achieve high performance*". Thus respondents who use C/C++ are less willing to adopt a defect prediction tool because they are unsure that defect prediction techniques can deal with the complexities involved in the

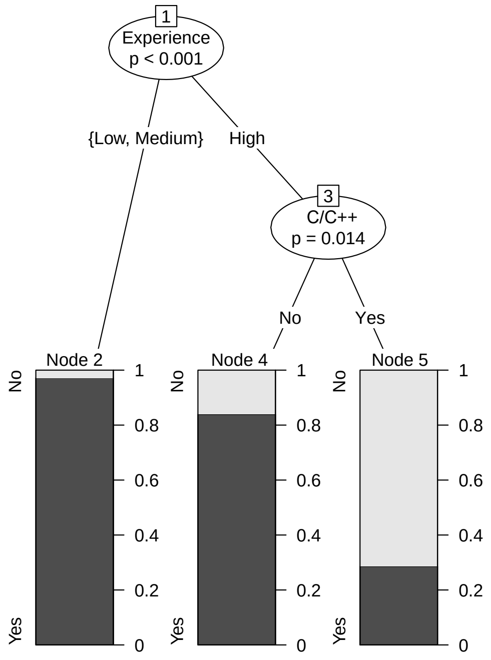


Fig. 7: Decision tree of factors that influence practitioners' willingness of adoption. "Yes" labels represent the responses that will adopt a defect prediction tool; "No" labels represent the responses that will not adopt a defect prediction tool.

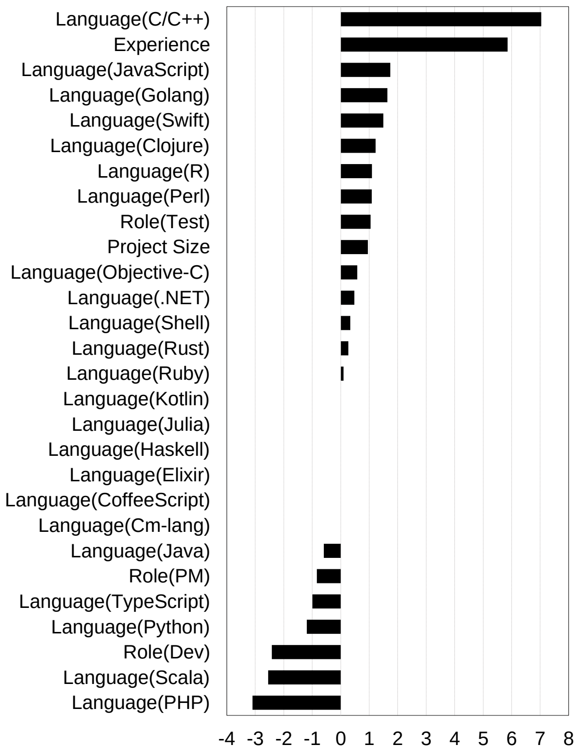


Fig. 8: Mean decrease in accuracy of each factor that affects willingness of adoption in random forest models.

programs that they write.

Furthermore, we adopted classification techniques to examine the importance of demographics features on adoption willingness. We adopted random forest since it is one of the most commonly used classification techniques and it is quite robust for noisy data [87]. We used the `randomForest` R package to implement random forest models. We bootstrap the data set by sampling the same size of our data set (with replacement) 1,000 times. The mean AUC of the random forest models is 0.7270 with a standard deviation 0.0182 and a standard error 0.0006 (min: 0.6718, median: 0.7273, max: 0.7803). To extract the demographics features that are important for the adoption willingness, we used the `importance` function in the `randomForest` R package to calculate variable importance measures as suggested in [22]. The `importance` function [51] implements two algorithms for calculating variable importance measures in the randomForest R package. The first algorithm calculates the Gini variable importance measure. The second algorithm calculates the mean decrease in accuracy using the out-of-bag observations. We leveraged the second algorithm as suggested in [22]. The mean variable importance measure of each factor is shown in Fig. 8. We found that C/C++ as primary programming languages (*Language(C/C++)*) and experience in years (*Experience*) are the top two important factors that influence the random forest models.

To further analyze those factors, we apply Scott-Knott Effect Size Difference (ESD) test [86] to group the 28 factors into statistically distinct groups according to their variable importance. Notice that each random forest model generates a vector of variable importance. We input the variable importance of those models into the Scott-Knott ESD test. The Scott-Knott ESD test evaluates the treatment means of variable importance scores for each factor. The null hypothesis is that the treatment means between variable importance scores of two factors are similar. The Scott-Knott ESD test then partitions the set of treatment means into statistically distinct groups (α = 0.05), and merge any two statistically distinct groups that have a negligible effect size into one group. Table 3 presents the 28 factors as ranked according to the Scott-Knott ESD test in terms of variable importance.

We identified 5 categories of reasons in unwilling cases as discussed below:

R1. A defect prediction tool is not needed. This group consists of comments where the practitioner perceived a defect prediction tool as not useful to their daily work. The respondents deemed that current state-of-practice is good enough and believed that no additional support is necessary.

- *... my software projects are not extremely large or complex.*
- *We don't have a bug problem. HTTP semantics + microservice architecture + strong testing = simple and few bugs.*
- *I don't see how most institutions I've worked at would be able to use this kind of thing in a sensible way ... I have no interest in introducing any more process.*
- *I have not seen any reason to do so. We automatically test all of our code and manually test/inspect practically none of our code ...*

R2. Incompatibility. Practitioners mentioned that a defect prediction tool may be incompatible with their personal or organizational development environments.

- *I prefer a simple development environment ...*

TABLE 3: Groups of factors that affect willingness of adoption created by Scott-Knott ESD test.

Factor	Mean of Variable Importance
Group 1	
Language (C/C++)	7.04
Group 2	
Experience	5.86
Group 3	
Language (JavaScript)	1.74
Language (Golang)	1.64
Group 4	
Language (Swift)	1.50
Group 5	
Language (Clojure)	1.22
Group 6	
Language (R)	1.09
Language (Perl)	1.09
Role (Test)	1.05
Project Size	0.95
Group 7	
Language (Objective-C)	0.58
Language (.NET)	0.48
Group 8	
Language (Shell)	0.33
Language (Rust)	0.27
Group 9	
Language (Ruby)	0.10
Language (Cm-lang)	0.00
Language (CoffeeScript)	0.00
Language (Elixir)	0.00
Language (Haskell)	0.00
Language (Julia)	0.00
Language (Kotlin)	0.00
Group 10	
Language (Java)	-0.59
Group 11	
Role (PM)	-0.84
Group 12	
Language (TypeScript)	-0.99
Group 13	
Language (Python)	-1.19
Group 14	
Role (Dev)	-2.42
Language (Scala)	-2.55
Group 15	
Language (PHP)	-3.09

- *... useful for me personally but a horrible nightmare when integrated into an institutional workflow ...*
- *Not allowed to prioritize fixing code that is not broken.*

R3. Cost outweighs benefits. Practitioners deemed the cost of using a defect prediction tool to be higher than the benefits gained from using the tool.

- *It's cheaper (on my time) to let users find and report the defects.*

R4. Disbelief in defect prediction. Some practitioners had strong disbelief in defect prediction techniques, including the heuristics, benefits, value and actionability.

- *At this point, I'm quite suspicious this kind of heuristic can ever be useful ...*
- *... instead of being a tool to help developers be more confident in their approach, it becomes a tool for oppressing technical people with "objective" metrics that can clearly be gamified.*
- *Would need to prove value.*
- *They don't provide useful information; they don't tell me anything I don't already know.*

R5. Another solution seems better. Some practitioners believed that it is better to work on alternative solutions that are likely to resolve the problem that is addressed by defect prediction.

- *It would be easier, cheaper, and more effective to eradicate error-prone programming languages like C and Java, but instead we are trying to fix software engineering from the outside with more tools and more heuristics ...*
- *I would rather invest in raising a team to code in Haskell then invest a team in tooling over a PHP, Ruby, Python or the mess of JS.*
- *The kinds of static analysis tools we use are more narrowly domain focused than a general "defect prediction" tool.*

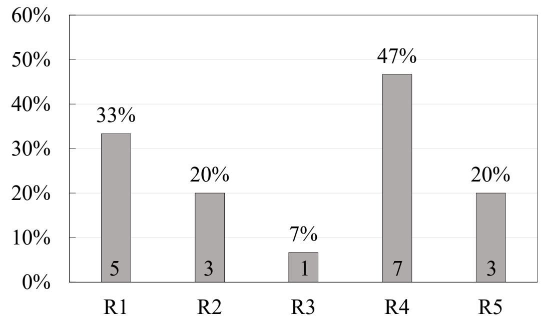


Fig. 9: Percentages of respondents specifying various reasons of adoption unwillingness.

Fig. 9 shows the percentage of respondents who provided reasons of unwillingness across various concerns. R4 *disbelief in defect prediction* is the biggest reason for unwillingness.

3.3 Perceptions of Practitioners

We present practitioners' perceptions regarding defect prediction in our study as follows.

3.3.1 Resource Prioritization

Table 4 presents the average score and variance for all the prioritization strategies. The strategies are sorted in terms of their average score (descending), followed by their score variance (ascending). The higher the average score of a strategy is, the more frequently the strategy is used; the higher the score variance is, the more controversial the strategy is. We observe that respondents believed that the most recently changed file and the most recent buggy file have the highest chance to have defects. Meanwhile, the most controversial strategy of all is related to files that are called by many other files.

Furthermore, we analyzed the score distribution of each strategy across experience levels of respondents. To check whether the difference of distribution between experience levels are statistically significant, we performed the Kolmogorov-Smirnov test. The null hypothesis is that the respondents with different experience levels have same score distribution of each strategy. We found no significant difference in score distribution across experience levels (p-values max: 1, min: 0.82, median: 1, mean: 0.93).

We further broke down the frequency ratings in Fig. 10, and observed the following:

- Almost 70% of the respondents indicated that they would *very often* or *often* select most recently changed file and most recent buggy file for code inspection/testing to find potential buggy files.

TABLE 4: Frequency of strategies that practitioners use to prioritize resources for code inspection/testing. Ratings have been converted to numeric scores (*very rarely* = 1, *rarely* = 2, *sometimes* = 3, *often* = 4, *very often* = 5). An answer score of 3 indicates neutrality. Variance is a measure of disagreement between respondents. "Total" is the number of respondents who rate a strategy and did not select *I don't know*.

	Prioritization Strategy	Total	Score	Variance
T1	Most recently changed file	391	3.80	0.71
T2	Most recent buggy file	386	3.80	0.72
T3	File created/changed by junior developers or newcomers	389	3.69	0.87
T4	File changed by a large number of developers	389	3.58	0.81
T5	File called by many other files	385	3.55	1.08
T6	File that calls many other files	386	3.48	0.90
T7	File with more LOC	387	3.44	0.75
T8	File changed by few developers	387	2.93	0.75
T9	File created/changed by senior developers	386	2.76	0.79
T10	File with fewer LOC	386	2.50	0.69

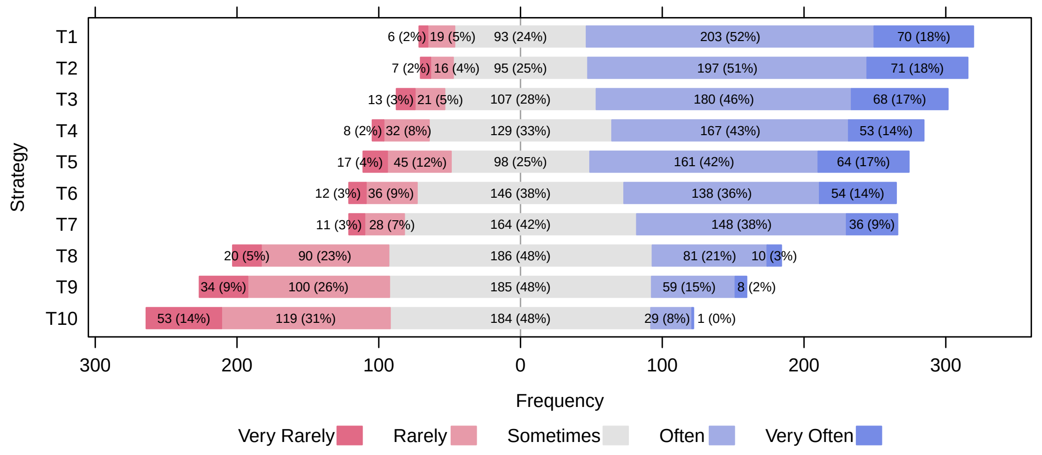


Fig. 10: Frequency of the strategies that practitioners use to prioritize resources for code insepction/testing.

- More than 50% of the respondents mentioned that they would *very often* or *often* select files changed by junior developers, changed by more developers, and called by many other files for code inspection/testing.
- Around 50% of the respondents were on the fence and chose *sometimes* for prioritizing a file that is changed by fewer developers, created/changed by senior developers, and with fewer LOCs.

In addition, we did solicit (as described earlier) practitioners' statements on how they formed their opinions. For the practitioners who expressed the strongest opinion (*very often* and *very rarely*), we gathered a collection of ranks of possible factors in their opinion formation. The results are shown in Fig. 11.

The highest ranked factor influencing respondents' opinion on a given strategy is *personal experience* which was selected 229 times. A look at the plot in Fig. 11 shows that *personal experience* was almost always ranked at the top, and a handful of times at other positions. Next is *what I hear from my peers*, which was selected 161 times, ranked second in terms of median rank. Next is *what I hear from my mentors/managers*, selected 128 times, ranked third in terms of median rank. The lowest ranked is *other*, selected 9 times. The factors affecting opinion are consistent with earlier work of Devanbu et al. [15]. In their work, the respondents gave the strongest weight to personal experience, then to peers, and then to mentor and managers. Furthermore, their respondents appeared to be influenced by trade journals rather than research papers.

To figure out if respondents who chose *personal experience* were senior enough to form a reliable opinion, we plotted the experience in years of those respondents in Fig. 12. Among those respondents, we found the median experience in years is 5 years; 31.7% of them are software practitioners with more than 10 years of experience. Additionally, a total of 215 respondents expressed strong opinions (those selected *very often* or *very rarely*) based on *personal experience*, accounting for 69%, 58% and 51% of respondents with high, medium and low experience respectively. Thus the more senior a respondent is, the more likely he/she formed a strong opinion based on *personal experience*.

Furthermore, we investigated how high-experience respondents who formed a strong opinion based on *personal experience* use prioritization strategies. We plotted the frequency of used strategies by high-experience respondents who formed a strong opinion based on *personal experience* as shown in Fig. 13. The prioritization strategies that are concerned with the number of developers (T4 and T8) [91], code (T7 and T10) [104], process (T1 and T2) [38] and complexity (T5 and T6) [104] are preferred by high-experience respondents. These prioritization strategies are consistent with research results in prior work. However, for strategies T3 and T9 that are concerned with the experience of developers, there exists a disconnect between the practice of high-experience respondents and research. As shown in Fig. 13, the high-experience respondents would prioritize files created or changed by low-experience developers over files created or changed by senior developers. However, an

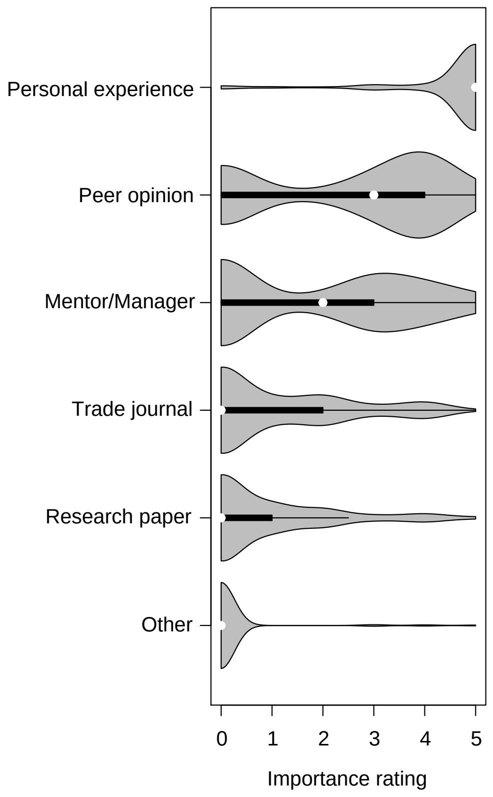


Fig. 11: Violin plots of factors stated as most influential on forming strong opinions about resource prioritization strategies.

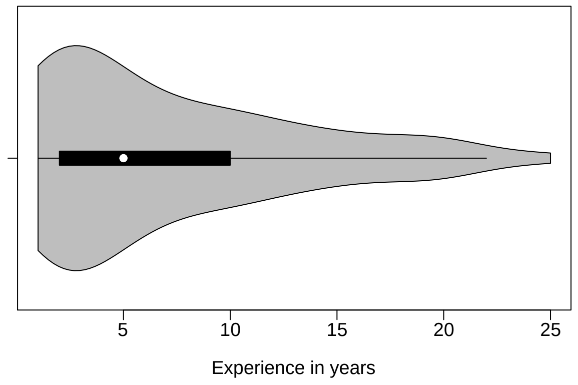


Fig. 12: Violin plot of experience in years of respondents that selected "*Personal experience*" factor in their opinion formation.

experienced developer might be overconfident in his/her approach to unfamiliar code, and thus lower the code quality. In prior work, Rahman and Devanbu found that the lack of general experience is not consistently associated with defect proneness [68].

3.3.2 Defect Density Distribution

Defect density is an important indicator of software quality. A good understanding of defect density allows project managers to invest resources proactively and efficiently to improve software quality before delivery [40]. Fig. 14 shows the percentage of respondents' different belief regarding defect density of files in a project. We observe that close

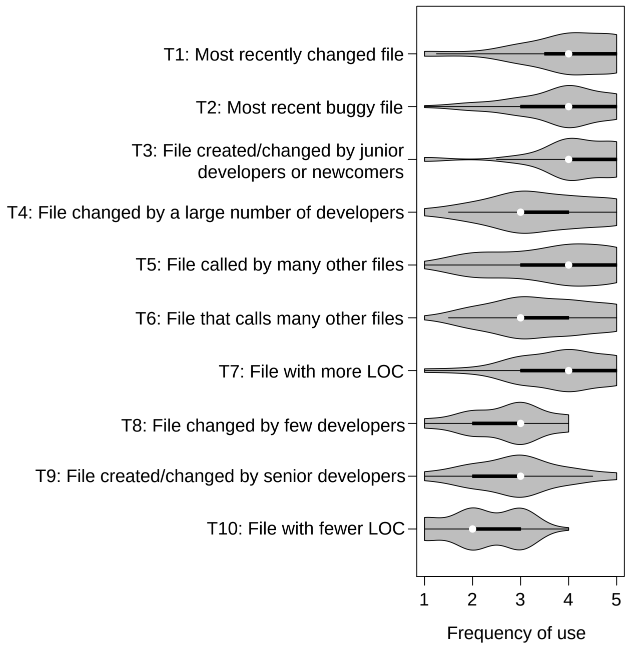


Fig. 13: Violin plots of frequency of strategies of senior respondents who formed a strong opinion based on *personal experience* (*very rarely* = 1, *rarely* = 2, *sometimes* = 3, *often* = 4, *very often* = 5).

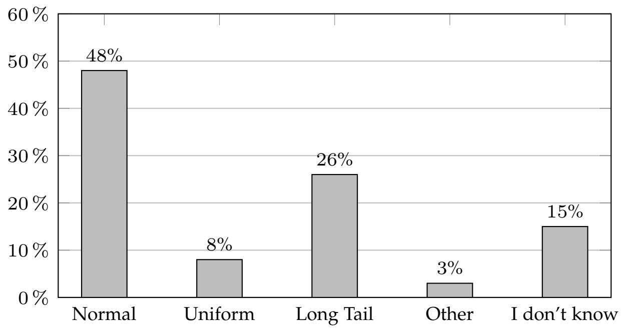


Fig. 14: Percentages of respondents specifying various defect density distributions.

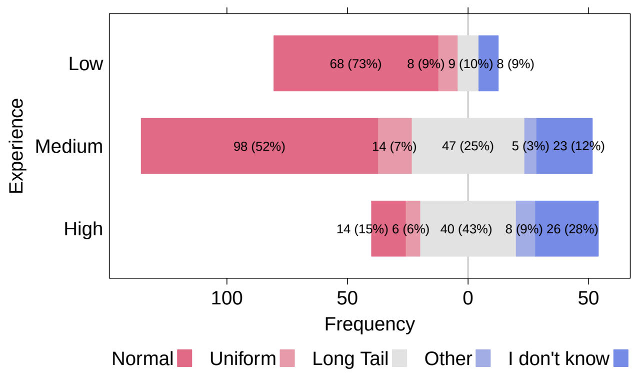


Fig. 15: Percentages of respondents specifying various defect density distributions relative to experience levels.

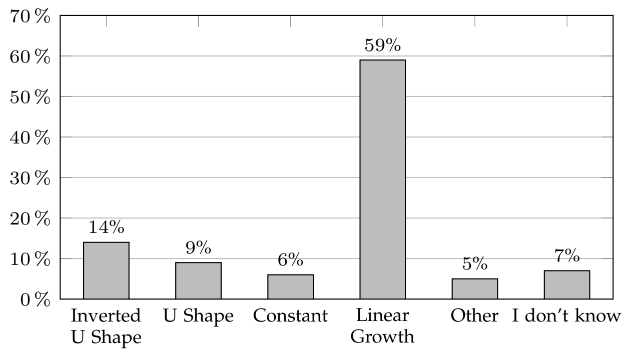


Fig. 16: Percentages of respondents specifying various file size vs. defect proneness correlations.

to 50% of the respondents considered a normal distribution, i.e., defect densities are randomly distributed across files in a project; 26% of respondents considered a long tail distribution, i.e., most files have a few defects and most defects appear in a small number of files in a project; less than 10% of the respondents considered a uniform distribution, i.e., bugs are distributed equally across files in a project.

Prior studies show that defects are distributed within a project according to the Pareto principle [3], [17], which is a long tail distribution. However, in total, nearly half of the respondents did not consider a long tail distribution; 15% of the respondents had no opinions about defect density distribution. There exists an obvious disconnect between research and practice.

To further analyze the impact of experience on the opinion of respondents, we grouped the distribution selections relative to the experience of respondents. Fig. 15 shows how the distribution selection varies as a function of the experience level of respondents. By observing the variation of defect prediction selection percentage across different experience levels, we found a 15% increase from low experience to medium experience and a 17% increase from medium experience to high experience in the population which selected long tail distribution. Specifically, nearly half of the high-experience respondents selected long tail distribution, accounting for around 42% of respondents who selected long tail distribution. In contrast, only 9.7% of the low-experience respondents selected long tail distribution. Practitioners' experience has a significant effect on the perception of defect density distribution. Surprisingly, 28% of high-experience respondents chose *I don't know*. Some of them mentioned that "defect density distribution is too complicated to be represented by the curves". As shown in Fig. 15, respondents are more willing to say *I don't know* as they become more senior.

Future studies are needed to better understand the disconnect between research and practitioners' perception. In addition, our findings suggest a large demand of defect prediction solutions for inexperienced software practitioners and novices. Defect prediction solutions could help inexperienced software practitioners and novices learn effective ways to identify defect-prone files in turn improving the effectiveness of their code inspection/testing efforts.

3.3.3 File Size vs. Defect Proneness

A good understanding of the size-defect correlation is essential for software practitioners [81]. Fig. 16 shows the percentage of different belief of respondents regarding relationship between file size and defect proneness. We observed that 59% of the respondents considered a linear growth correlation between file size and defect proneness, i.e., probability of defect proneness would increase proportionally as file size increases. 14% of the respondents considered an *inverted "U" shape* correlation between file size and defect proneness, i.e., smallest files and largest files are less defect-prone. 9% of the respondents selected *"U" shape* (i.e., smallest files and largest files are more defect-prone) and 6% selected *constant* (i.e., file size does not affect defect proneness) correlations. In addition, 7% of the respondents had no idea about the relationship between file size and defect proneness.

We further analyzed the effect of respondents' experience on their perception of relationship between size and defect proneness. Fig. 17 shows how the perceptions of the relationship between size and defect proneness varies as a function of the experience levels of respondents. For *inverted "U" shape* relationship, we found a 12% decrement from low experience to medium experience and a 5% decrement from medium experience to high experience. Over half of the respondents in each experience level perceived a *linear* size-defect relationship.

The work of Koru et al. has been instrumental in the recent understanding of size-defect relationship [43]–[45]. In contrast to the perception of most practitioners, findings of Koru et al. suggested a nonlinear size-defect relationship where defect proneness increases with size *at a slower rate*. Thus smaller instances are proportionally more defect prone; that is a "U" shaped relationship between size and defect proneness. Syer et al. [81] replicated the work of Koru et al. and reached contradicting findings: defect proneness has an *inverted* "U" shaped pattern (i.e., defect density *increases* in smaller files, peaks in the small/medium-sized files, then *decreases* in medium-sized and larger files).

As shown in Fig. 16, a total of 14% of the respondents perceived an *inverted* "U" shaped relationship between size and defect proneness. Fig. 17 shows that the percentage is even lower (7%) for the high-experience respondents. There exists an obvious disconnect between research and practitioners' perceptions. Our findings suggest that more efforts should be taken to expand prior studies [43]–[45], [81] and provide more evidence regarding the need to disseminate empirical findings to software practitioners.

3.3.4 Defect Prediction Metrics

Metrics in defect prediction research are used as independent variables of defect prediction models. Table 5 presents the percentages of agreement and conflict factors for all the statements regarding the studied metric groups. In each metric group, statements are sorted in terms of their percentages of agreement (descending), followed by conflict factors (descending). The higher the percentage of agreement is, the more the agreement is with the statement; the lower the conflict factor is, the more the conflict of opinion is within respondents for a particular statement. Fig. 18 shows respondents' agreement levels for each of the statements. We made the following observations:

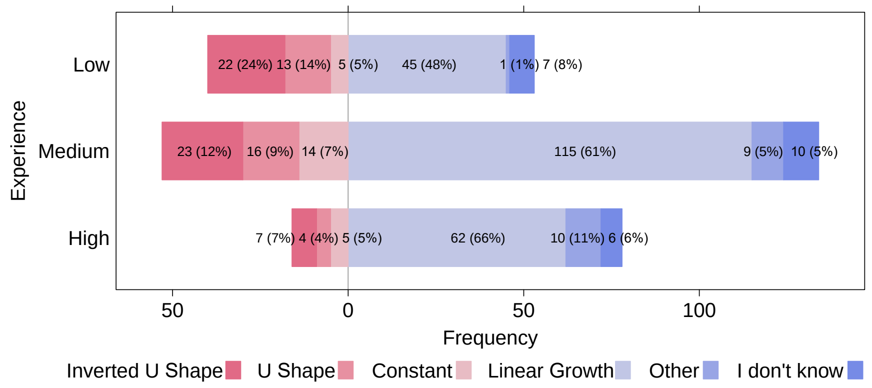


Fig. 17: Percentages of respondents specifying the different file size vs. defect proneness correlations relative to experience levels.

TABLE 5: Respondents' ratings on statements related to defect prediction metrics. "Total" is the number of respondents who rated a statement and did not select *I don't know*. "% Agreement" is the percentage of respondents who strongly agree or agree with each statement ($\%\ strongly\ agree + \%\ agree$). "Conflict Factor" is a measure of disagreement between respondents, which is calculated for each statement by the following equation: $(\%\ strongly\ agree + \%\ agree)/(\%\ strongly\ disagree + \%\ disagree)$.

	Metric Statement	Total	% Agreement	Conflict Factor
	Code Metrics			
S1	Semantic information is more capable of distinguishing one code region from another than syntax information.	365	**62.46**	9.49
	Process Metrics			
S2	A file with a complex code change process tends to be buggy.	386	**76.17**	22.60
S3	A file with frequent changes tends to be bug-prone.	389	62.21	8.64
S4	A file with more added lines is more bug-prone.	389	61.95	7.30
S5	A file with more bug-fix changes tends to be more bug-prone.	390	61.80	5.74
S6	A file recently co-changed with bug-introduced files tends to be buggy.	382	61.26	9.35
S7	Recently changed files tend to be buggy.	390	58.72	5.72
S8	A commit that involves more added and removed lines is more bug-prone.	388	57.98	5.11
S9	Recently created files tend to be buggy.	389	52.70	4.18
S10	Recently bug-fixed files tend to be buggy.	389	49.62	3.78
S11	A file with more fixed bugs tends to be more bug-prone.	387	48.06	2.95
S12	A file with more commits is more bug-prone.	389	46.27	**2.34**
S13	A file with more removed lines is more bug-prone.	389	35.73	**1.30**
	Ownership Metrics			
S14	A file that is changed by more developers is more bug-prone.	386	**64.51**	8.89
S15	Files with fewer lines contributed by their owners (who contribute most changes) are more bug-prone.	376	30.59	**1.47**

- The top 3 statements that the respondents agreed with are S2, S14 and S1, whose percentages of agreement are 76.17, 64.51 and 62.46 respectively.
  The statement S2 is concerned with the complexity of code change process. The result is consistent with prior work by Hassan who designed metrics to measure the complexity of change processes using Shannon's Entropy [26]. His work reported that a complex code change process negatively affects the quality of software: the more complex changes to a file, the more likely that the file will be defect-prone. Their results further indicated that change process complexity metrics are better to predict proneness than other process metrics.
  The statement S14 is concerned with the number of developers for a software system. The result is consistent with Weyuker et al.'s findings [91]. The metrics related to the number of developers had a positive, statistically significant relationship with defect proneness.
  The statement S1 is related to the semantic information of code region. The result is consistent with prior work. In a recent study, Wang et al. leveraged deep learning algorithm to learn semantic information of programs from source code [90]. Their results on ten open source projects showed that their learned semantic information significantly improve defect prediction compared to traditional features.
- The top 3 statements that have conflict in the opinions of respondents are S13, S15 and S12, whose conflict

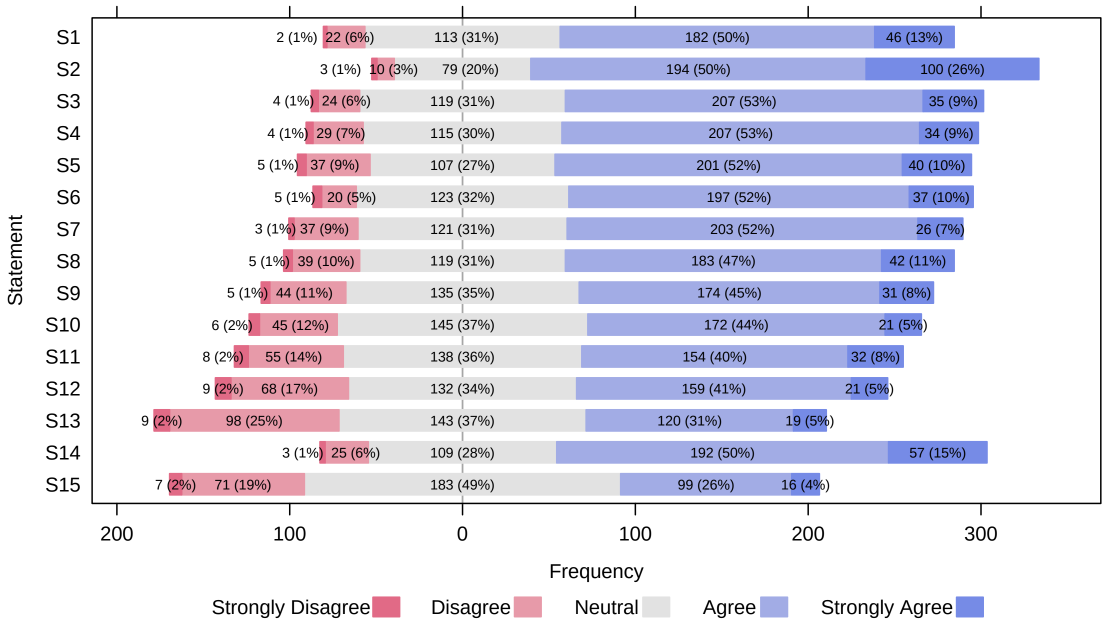


Fig. 18: Practitioners' agreement levels for statements related to defect prediction metrics.

factors are 1.30, 1.47 and 2.34 respectively.
The statements S13 and S12 are concerned with process metrics. The statement S13 is concerned with the number of deleted lines in a file. The statement S12 is regarding the number of commits made to a file. However, Rahman and Devanbu found that metrics related to commit count and deleted line count had good predictive power of defect proneness [69].
The statement S15 is related to code ownership. The conflict about code ownership is consistent with findings in prior studies. On the one hand, Raymond [73] claimed that increasing the number of collaborators would accelerate defect diagnosis. On the other hand, Martin et al. [67] observed that "too many cooks" working on the software leads to unfocused, defect-prone contributions. Researchers also found that these overheads can slow down development [28] and increase defects [11].

- The statements S3, S4, S5, S6, S7, S8 show similar agreement levels.

The strategies that developers adopt to prioritize code inspection/testing efforts reflect practitioners' behavior regarding defect prediction. To investigate if practitioners' behavior always corresponds with their perceptions, we built mappings between prioritization strategies and defectiveness metrics (as shown in Table 4 and Table 5): Statement *S7* and Strategy *T1* are both concerned with "*recently changed file*"; Statement *S10* and Strategy *T2* are both concerned with "*recent buggy file*"; Statement *S14* and Strategy *T4* are both concerned with "*file that is changed by more developers*". We then separately ranked the 3 strategies (i.e., *T1*, *T2* and *T4*) according to their mean scores and 3 statements (i.e., *S7*, *S10* and *S14*) according to their percentages of agreement. Among the 3 statements, statement *S14* ranked at the top with the greatest percentage of agreement. However, its corresponding strategy *T4* ranked the third. The agreement rank of a statement is not exactly in line with the frequency of our respondents use of the corresponding strategies. In addition, we found that 13 respondents out of 249 who strongly agreed or agreed with statement *S14* very rarely or rarely used strategy *T4*; 7 out of 28 who strongly disagreed or disagreed with statement *S14* very often or often used strategy *T4*. In total, 7.2% of the respondents reveal an inconsistency between their behavior and their perception.

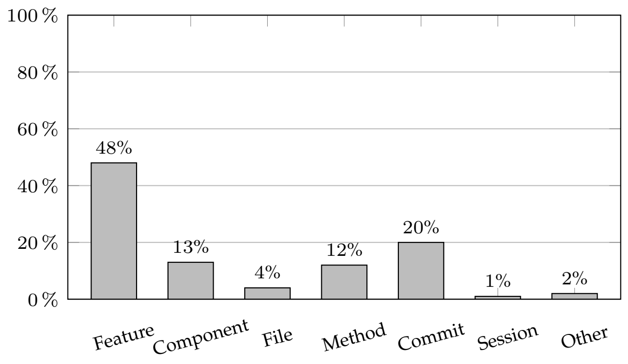


Fig. 19: Percentage of respondents who selected various granularity levels as their topmost preferred granularity levels.

3.4 Expectations of Practitioners

We present practitioners' expectations regarding defect prediction in our study as follows.

3.4.1 *Preferred Granularity Levels*

Defect prediction techniques can predict defects at different granularity levels, e.g., feature, component, file (class), method, commit and session (every time one saves a file). Fig. 19 shows practitioners' topmost *preferred* granularity levels when prioritizing code inspection/test efforts. We observe that the top-3 preferred granularity levels are feature (48%), commit (20%) and component (13%). Feature is a clear winner among the preferred granularity levels. Some respondents picked "*other*" as their topmost preferred

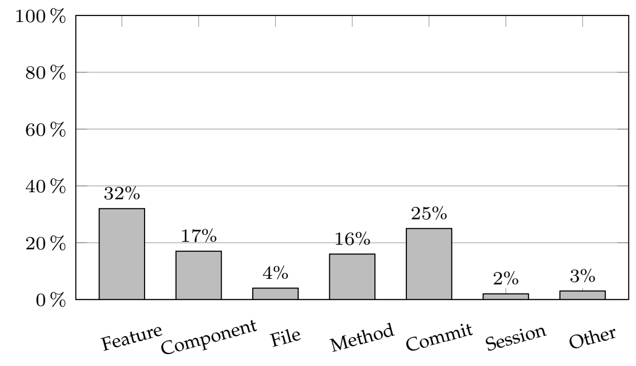


Fig. 20: Percentage of respondents with high experience who selected various granularity levels as their topmost preferred granularity levels.

granularity level. Among these respondents, some of them specified that they preferred a tool that can predict defective pull requests and the exact defective logic branch.

We further analyzed the topmost preferred granularity levels of the high-experience respondents. The top-3 preferred granularity levels are feature (32%), commit (25%) and component (17%) as shown in Fig. 20. The ranking is consistent with the ranking of preferred granularity levels of all respondents. Although feature level is still the most preferred granularity level, the percentage of respondents who preferred feature level defect prediction decreased by 16%.

3.4.2 Expected Features

A total of 67 respondents provided free-form text comments regarding expected features. We found groups of features that the respondents expected (followed by their corresponding frequency) as follows:

- **Integration with IDE/editor (7)**
  - *IDE/editor-plugin with inline visual marking of errors and explanations of why an error is considered an error.*
  - *Nice IDE with nice highlighted LOC on possible buggy code ...*
  - *IDE integration that shows warnings whenever the dangerous place in a file is modified, tooltips and notes by the IDE to help navigate through the possible defective places in the code.*
- **Integration into CI (7)**
  - *Needs to be very, very easy to configure in standard continuous integration setups ...*
  - *Integration with Jenkins and other CI tools would be awesome.*
  - *I would try them out if I could integrate them into a CI tool like Jenkins.*
  - *It would be ideal for the functionality of the tool to run ... also on a CI server as a fallback.*
- **Integration with code review tools (7)**
  - *... Code review helper, typically show most risky files/part of code, first with color codes.*
  - *While reviewing code in a pull request, I can visually see in the changes that are prone to being buggy based on this analysis.*
- **Ability to provide rationale (6)**
  - *... ideally with the reasoning (i.e., see these lines in this pair of commits) for why it's a likely defect.*
  - *... and explanations of why an error is considered an error.*
  - *... Tell me why certain lines are potentially wrong.*
  - *... The tool would have to make a very compelling case for spending extra review effort.*
  - *... Anything flagged would have a concrete, context relevant example ... help developers of all skill levels understand why the code is suspect, so they can learn to avoid making the same mistake in the future.*
- **Full programming language support (6)**
  - *language specific "known bugs" and language independent attack vectors should be checked.*
  - *Should be fully integrated into the language.*
- **Simplicity (5)**
  - *Keep it simple, keep it a command-line tool with a good command-line interface ...*
  - *I'd prefer a simple tool that can analyse history and metrics that arise from software engineering process ...*
- **Integration with software configuration management systems (4)**
  - *Easy to plug in with github so it analyzes every commit in every branch.*
  - *Github widget which annotates the review web page for me with notes on what to also note. The history between me the reviewer and the code author - how have I reviewed their code before?*
  - *... Integration with SCM via github and/or bitbucket.*
- **Ability to provide actionable guidance (4)**
  - *... recommendation for refactoring, recommendation for usage of more suitable data structure ...*
  - *... Suggested solutions for easy fixes (off by one, typos, etc.)*
- **Cooperation with coverage analysis (4)**
  - *Another thing I miss on defect prediction tools is integration with code coverage that shows more than an obvious percentage.*
  - *I would expect something that I have now with test coverage tools ...*
- **Online prediction (4)**
  - *... flags changes that worsen metrics.*
  - *... give a risk grade for the change, making the engineer aware of what he is doing.*
  - *continuously analyzing code to detect the error asap while you develop.*
  - *Defect prediction must be ONLINE. It must react to change.*
- **Low false positives (3)**
  - *I want it to provide as little noise as possible (I'd rather learn less than have false positives.*
  - *... if a framework provides some mitigation, it's effectiveness must be taken into account. This should lower critical warnings.*
  - *... False positive rate must be low.*
- **Efficiency (2)**
  - *I would like it if it were fast enough to run over modified files for every save of file, on the order of hundreds of milliseconds or less.*

- **Scalability (1)**

  ✍ *Something that can process a whole github.com.*

- **Others (4)**

  ✍ *Must integrate with and respect existing process.*

  ✍ *Track developer proficiency in languages (number of introduced bugs in a given language by a developer in the past per LOC), and use that to prioritize what part of the code to check more for potential defects.*

  ✍ *... I would like a defect prediction tool that could recognize semantic constructs that are frequently the source of bugs in *other* people's code. In this way, I could benefit from the knowledge of the entire community, rather than just my or my team's past projects.*

  ✍ *... you can also perhaps predict per developer, what faults they might be more prone to introduce. That way, you can focus in a more targeted way.*

3.5 Adoption Challenges

We present practitioners' adoption challenges regarding defect prediction in our study as follows.

3.5.1 Bug Fixing

Table 6 presents the percentages of agreement and conflict factors for all the statements related to bug fixing. Statements are sorted in terms of their percentages of agreement (descending), followed by the conflict factors (descending). The higher the percentage of agreement is, the more the agreement is with the statement; the lower the conflict factor is, the more the conflict of opinion is within respondents for a particular statement. Fig. 21 shows respondents' agreement levels of the statements. We made the following observations:

- The statement with which respondents disagreed the most was "*S6: Sometimes a try-catch block could be placed anywhere between where the exception was originally thrown and the user interface.*" (% Agreement = 35.38); this statement also incited the most disparity in respondents' agreement (Conflict Factor = 1.33). The statement S6 corresponds to one design dimension of bug fixes - "data propagation (across components)" [58]. "Data propagation" describes how far information is allowed to propagate across a piece of software, where developers have the option of fixing a bug by intercepting the data in any of the components of the software. The results are inconsistent with the findings of Murphy-Hill et al. [58]. In their interviews, interviewees mentioned that placement of try-catch block at any locations between where the exception was originally thrown and the user interface would have fixed the bug from the end-user's perspective.
- The statement with which most respondents agreed was "*S1: The same bug can be fixed in multiple ways*" (% Agreement = 79.70); which was also the least controversial statement related to bug fixing (Conflict Factor = 20.92). Note that the statement S1 is the assumption that was explored by Murphy-Hill et al. [58]. Based on this assumption, they explored the design dimensions of bug fixes.
- 52% of our respondents claimed that "*Location in the software at which I fix a bug is the location at which an error was made*" (S3); whereas 43% of the respondents mentioned that "*Some defects are not fixed by correcting the 'real' error-causing component, but rather by a 'work-around' somewhere else.*"(S5). This result gives evidence to support an assumption in Murphy-Hill et al.'s study [58]: fixed bugs in the past may not capture the true cause of failures, instead they may capture work-arounds.
- 66% of our respondents strongly agreed or agreed that "*Sometimes the real error lies too deep. So risk of introducing new errors is too high to solve the real error*" (S2). The statement may help to understand under what circumstances S5 come into existence.

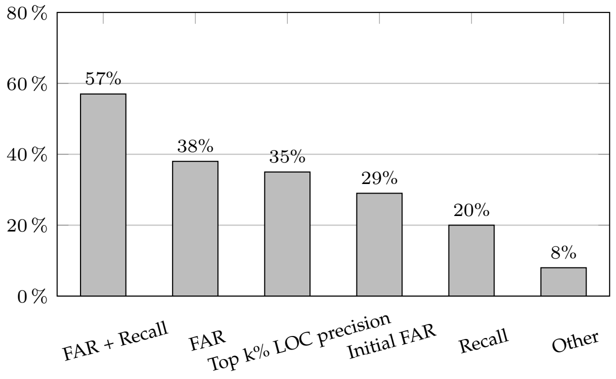


Fig. 22: Percentage of respondents who selected various measures as their preferred measure for evaluating defect prediction tools.

3.5.2 Performance Evaluation

Fig. 22 shows practitioners' preferred measures to evaluate performance of defect prediction tools. Note that the percentages do not add up to 100% since a respondent can select 1-5 options as their preferred measurements. We can observe that the top-3 preferred measurements are a combination of false alarm rate and recall (57%, "FAR + Recall" in Fig. 22) , false alarm rate (38%, "FAR" in Fig. 22) and Top k% LOC precision (35% in Fig. 22). False alarm rate represents the proportion of non-defective files among all the inspected files. A higher false alarm rate indicates that users would encounter more false alarms. Recall is the proportion of flagged defective files among all the actual defective files. A lower recall indicates that less defective files could be detected. Top k% LOC precision is the proportion of defective files inspected when k% LOC are inspected; a lower value indicates that, after inspecting the same number of LOC (k%), developers need to inspect more files.

3.5.3 Adoption Barriers

Fig. 23 presents the adoption barriers of defect prediction tools from practitioners' perspective. Note that the percentages do not add up to 100% since a respondent can select 1-4 options as their adoption barriers. We observe that the top-3 adoption barriers are "*lack of continuous integration support*" (64%, "CI" in Fig. 23), "*lack of code review tool integration*" (61%, "CR" in Fig. 23) and "*lack of IDE integration*" (53%, "IDE" in Fig. 23). There is no clear winner among these three barriers, with respondents slightly more concerned about support of continuous integration.

TABLE 6: Respondents' ratings on statements related to bug fixing. "Total" is the number of respondents who rated a statement and did not select *I don't know*. "% Agreement" is the percentage of respondents who strongly agree or agree with each statement ($\%\ strongly\ agree + \%\ agree$). "Conflict Factor" is a measure of disagreement between respondents, which is calculated for each statement by the following equation: $(\%\ strongly\ agree + \%\ agree)/(\%\ strongly\ disagree + \%\ disagree)$.

	Bug Fixing Statement	Total	% Agreement	Conflict Factor
S1	The same bug can be fixed in multiple ways.	394	**79.70**	20.92
S2	Sometimes the real error lies too deep. So the risk of introducing new errors is too high to solve the real error.	392	66.07	3.41
S3	Location in the software at which I fix a bug is the location at which an error was made.	391	52.17	3.19
S4	To fix a bug, I prefer to change the code I am familiar with rather than the code for which I have no ownership.	391	43.47	1.47
S5	Some defects are not fixed by correcting the "real" error-causing component, but rather by a "work-around" somewhere else.	389	43.45	1.59
S6	Sometimes a try-catch block could be placed anywhere between where the exception was originally thrown and the user interface (Placement of the block at any of these locations would have fixed the bug from the perspective of end-users by eliminating the exception being thrown).	376	35.38	**1.33**

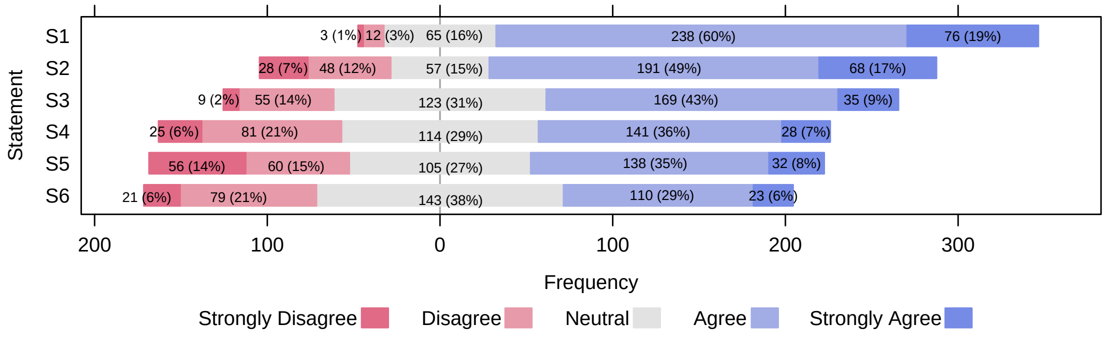


Fig. 21: Practitioners agreement levels of statements related to bug fixing.

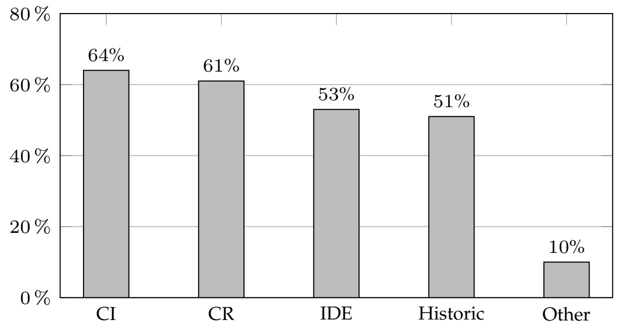


Fig. 23: Percentage of respondents who selected various factors as barriers to adopt defect prediction tools.

4 Discussion

We summarize the results in our study, and discuss implications and threats to validity in this section.

4.1 Summary of Results

For each of the research questions, we summarized our results in Table 7 and Table 8.

4.2 Implications

We discuss the implications of our results as follows.

4.2.1 Perception vs. Evidence

Defect density distribution. Practitioners' experience has a significant effect on their perception of defect density distribution. Nearly half of the high-experience respondents selected the long tail distribution, accounting for 42% of respondents who selected long tail distribution. In contrast, only 9.7% of the low-experience respondents selected long tail distribution. Future studies are needed to provide more evidence on defect density distribution. In addition, our observations suggest a large demand for defect prediction education and solutions for inexperienced software practitioners and novices. Defect prediction solutions could help them learn effective ways to identify defect-prone files and thus improve the effectiveness of their code inspection/testing efforts.

File size vs. defect proneness. A total of 14% of the respondents perceived an *inverted* "U" shaped relationship between size and defect proneness. The percentage is even lower (7%) for the high-experience respondents. There exists an obvious disconnect between research findings and practitioners' perceptions. Our findings suggest that more efforts is needed to expand prior studies [43]–[45], [81] and provide more evidence regarding the need to disseminate empirical findings to software practitioners.

TABLE 7: Summary of Results: Answers to RQ1 and RQ2.

RQ1. Willingness of adoption

Overall findings:

- Only 7.8% are unwilling to adopt defect prediction tools.

Differences in willingness of adoption across demographic factors:

- Testers are slightly more willing to adopt a defect prediction tool than other job roles.
- More experienced respondents are less willing to adopt a defect prediction tool.
- Practitioners who use Scala, R and golang as their primary programming languages are the top-3 groups who are unwilling to adopt a defect prediction tool.

In unwilling cases:

- High-experience respondents who use C/C++ as their primary programming languages are the majority of the unwilling population.
- The reasons of unwillingness include "a defect prediction tool is not needed", "incompatibility", "incompatibility", "disbelief in defect prediction", and "another solution seems better".

RQ2. Perception

Resource prioritization

Overall findings:

- Most frequently used strategy: near 70% of the respondents select "most recently changed file" and "most recent buggy file" for code inspectation/testing.
- Most controversial strategy: select "files that called by many other files" for code inspectation/testing.

Impact of experience:

- No significant differences across experience levels of respondents.

Senior practitioners:

- The more experienced a respondent is, the more likely he/she formed a strong opinion based on personal experience.
- High-experience respondents who formed a strong opinion based on personal experience frequently use strategies that are concerned with number of developers, code, process and complexity.
- High-experience respondents are inclined to prioritize files created or changed by low-experience developers over files created or changed by high-experience developers.

Defect density distribution

Overall findings:

- close to 50% of the respondents selected a normal distribution.
- 26% of respondents selected a long tail distribution.
- less than 10% of the respondents selected a uniform distribution.

Impact of experience:

- In the population who selected long tail distribution, we observed a 15% increase from low experience to medium experience and a 17% increase from medium experience to high experience.
- Nearly half of the high-experience respondents selected long tail distribution, accounting for around 42% of respondents who selected long tail distribution. In contrast, only 9.7% of the low-experience respondents selected long tail distribution.

File size vs. defect proneness

Overall findings:

- 59% of the respondents considered a linear growth correlation between file size and defect proneness.
- 14% of the respondents considered an inverted "U" shape correlation between file size and defect proneness.
- 9% of the respondents selected "U" shape and 6% selected constant correlations.

Impact of experience:

- Only 7% of the high-experience respondents perceived an inverted "U" shaped relationship between size and defect proneness.
- For inverted "U" shape relationship, we found a 12% decrement from low experience to medium experience and a 5% decrement from medium experience to high experience.
- Over half of the respondents in each experience level perceived a linear size-defect relationship.

Defect prediction metrics

Overall findings:

The top 3 statements that the respondents agreed with are:

- Statement S2 which is concerned with the complexity of code change process.
- Statement S14 which is concerned with the number of developers for a software system.
- Statement S1 which is related to the semantic information of code region.

The top 3 statement that have conflict in the opinions of respondents:

- Statement S13 which is concerned with the number of deleted lines in a file.
- Statement S12 which is regarding the number of commits made to a file.
- Statement S15 which is related to code ownership.

TABLE 8: Summary of Results: Answers to RQ3 and RQ4.

RQ3. Expectation

Preferred granularity levels

Overall findings:

- The top-3 preferred granularity levels are feature (48%), commit (20%) and component (13%).

Senior practitioners:

- The top-3 preferred granularity levels of the high-experience respondents are feature (32%), commit (25%) and component (17%).

Expected features

- Integration with IDE/editor
- Integration into CI
- Integration with code review tools
- Integration with software configuration management systems
- Cooperation with coverage analysis
- Ability to provide rationale
- Ability to provide actionable guidance
- Full programming language support
- Online prediction
- Simplicity
- Low false positives
- Efficiency
- Scalability

RQ4. Challenges

Bug fixing

- The statement that respondents disagreed most with and incited the most disparity in agreement was concerned with "S6: Sometimes a try-catch block could be placed anywhere between where the exception was originally thrown and the user interface".
- The statement that most respondents agreed with and was least controversial was "S1: The same bug can be fixed in multiple ways".
- 52% of our respondents claimed that "Location in the software at which I fix a bug is the location at which an error was made" (S3).
- 43% of the respondents mentioned that "Some defects are not fixed by correcting the real error-causing component, but rather by a work- around somewhere else"(S5).
- 66% of our respondents strongly agreed or agreed that "Sometimes the real error lies too deep. So risk of introducing new errors is too high to solve the real error" (S2).

Performance evaluation

The top-3 preferred measurements are a combination of false alarm rate and recall (57%) , false alarm rate (38%) and Top k% LOC precision (35%).

Adoption barriers

The top-3 adoption barriers are "lack of continuous integration support" (64%), "lack of code review tool integration" (61%) and "lack of IDE integration" (53%).

4.2.2 Behavior vs. Perception

The strategies that developers adopt to prioritize code inspection/testing efforts reflect practitioners' behavior regarding defect prediction. To investigate if practitioners' behavior always corresponds with their perception, we built mappings between prioritization strategies and defectiveness metrics (as shown in Table 4 and Table 5): Statement *S7* and Strategy *T1* are both concerned with "*recently changed file*"; Statement *S10* and Strategy *T2* are both concerned with "*recent buggy file*"; Statement *S14* and Strategy *T4* are both concerned with "*file that is changed by more developers*". We then separately ranked the 3 strategies (i.e., *T1*, *T2* and *T4*) according to their mean scores and 3 statements (i.e., *S7*, *S10* and *S14*) according to their percentages of agreement. Among the 3 statements, statement *S14* ranked at the top with the greatest percentage of agreement. However, its corresponding strategy *T4* ranked the third. The agreement rank of a statement is not exactly in line with the frequency at which our respondents used the corresponding strategy. In addition, we found that 13 respondents out of 249 who strongly agreed or agreed with statement *S14* very rarely or rarely used strategy *T4*; 7 out of 28 who strongly disagreed or disagreed with statement *S14* very often or often used strategy *T4*. In total, 7.2% of the respondents reveal an inconsistency between their behavior and their perception.

Future defect prediction tools should provide rationales that explicitly describe the connection between defectiveness metrics and prioritization strategies. The rationales may help software practitioners to better prioritize code inspection/testing effort.

4.2.3 Expectations

Granularity levels. Surprising, almost half of our respondents preferred feature level defect prediction (requirement or *conceptual concerns* [10] proposed by customers/users). Our finding differs from the conclusion of a prior study

[36] which concluded that the practical value of prediction decreases as the granularity level increases. Although defect prediction of fine granularity facilitates defect localization, coarse granularity would help practitioners gain an insight into overall quality.

Researchers have proposed various metrics based on source code entities (e.g., methods, classes, or components). These metrics largely ignored features, i.e., the *conceptual concerns*, of a software system [10]. Unlike a method or a class, a feature may span components, creating feature interdependencies. The boundary of features could be unclear. Dependency analysis could be used to gain a better understanding of how a feature is implemented. Extraction of feature level metrics must consider such feature interdependencies. In addition, in agile programming practice, team members usually specify, develop, and test each new feature during each release cycle. Future studies are needed to examine the feasibility of feature level defect prediction.

Features. Integration into continuous integration, code review software, popular IDE is practitioners' frequently expected feature. In our survey, lack of integration with CI, CR software and IDE is perceived as the top-3 adoption barriers of defect prediction techniques. Several industrial studies have been conducted to investigate defect prediction techniques at Microsoft [12], [59], [103], Google [50], Avaya [56], BlackBerry [37], [77] and Cisco [82]. There is a need for a community-wide effort to encourage integrating state-of-the-art defect prediction techniques into continuous integration, code review software and IDEs.

We observe there is high preference among practitioners for sensible, interpretable and actionable metrics when applying defect prediction models in a practical setting. Future studies are needed to explore the relationships between metrics and defect proneness in order to provide understandable insights and inform decisions in various phases of software practice.

Some of the respondents in our survey expect online defect prediction. However, a limited number of prior studies apply defect prediction techniques in an online manner [82]. Future studies are needed to investigate online defect prediction models in practice.

4.3 Threats to Validity

Internal Validity. It is possible that some of our survey respondents had a poor understanding of defect prediction or of our questions. Their responses may introduce noise to the data that we collected. To reduce the impact of this issue, we included an "I don't know" option in the survey and ignored responses marked as such. We also dropped responses that were submitted by people whose job roles are none of these: software development, testing and project management. The first and second authors translated our survey to Chinese to ensure that respondents from China could understand our survey well. To reduce the bias of presenting survey bilingually, we carefully translated our survey to make sure there is no ambiguity between English and Chinese terms. We also polished the translation by improving clarity and understandability according to the feedbacks from our pilot survey.

Some findings reported in this work depend on our understanding about respondents' perception from their comments. To minimize this threat, we read the comments several times, and followed a process to help improve the quality of our conclusions.

External Validity. To improve the generalizability of our findings, we interviewed 16 interviewees from two companies, and surveyed 395 respondents from 33 countries across five continents who are working for various companies (e.g., Microsoft, Amazon, Google, Baidu, IBM, Morgan Stanley, Hengtian and IGS) or contributing to open source projects that are hosted on GitHub, in various roles. We wish though to highlight that while we selected employees from two Chinese IT companies for our interviews, the surveyed population is considerably wide. The responses from the interviews were used to bootstrap the options in our survey questions. The survey permitted respondents to add additional comments whenever appropriate via free-form fields; looking at the responses in such fields we do not observe any signs of missing options. Moreover, our surveyed population represents one of the most diverse population of any software engineering study in recent history. In contrast, much of the recent studies focus on impression within a single company (e.g. Google [31], or Microsoft [15]). Moreover, given the consulting positions of these interviewed employees, the interviewed practitioners bring a much wider perspective given that they have worked on projects around the globe, namely in Canada, Australia, Japan, Ireland, Germany and USA. These practitioners continue to work with many companies around the globe (e.g., Microsoft, Honda, Bosch, IBM, Cisco, Alibaba, and DST). Finally, we do also note that as our interviews were drawing to a close, the collected codes from interview transcripts came to saturation. New codes did not appear anymore; the list of codes was considered stable. This is also noted by Guest et al. [23] that conducting 12 to 15 in-depth interviews of a homogeneous group is adequate to reach saturation. We also provide various details about our survey (e.g., the actual survey) to ease future replications. Still, our findings may not generalize to represent the perception of all software practitioners. However, to the best of our knowledge, this work is the largest and widest study to date on this topic.

In our survey, we only considered several barriers that may hinder the adoption of defect prediction tools, including *"cost of collecting historic data"*, *"lack of IDE integration"*, *"lack of code review tool integration"* and *"lack of continuous integration support"*. Through our open card sort, we identified other specific barriers from the respondents' free-form text comments, i.e, answers to *why won't you adopt a defect prediction tool*. The other barriers include *"lack of awareness"*, *"language support"*, *"false alarms"*, *"accuracy"*, *"developer overhead"* and *"learning curve"*. Future studies can reduce this threat by including these barriers, performing a second round of surveys by inviting more respondents.

Our survey can only assess practitioners' perceived willingness to adopt. However, perceived willingness of adoption is one factor that leads to actual adoption [42]. Actual adoption typically depends on various other factors, e.g., social, culture, education and exposure [92], [95]. Many of these factors are often external to the actual technique.

5 Related Work

We briefly review the related work in this section.

5.1 Surveys of Software Practitioners

There have been several prior studies of surveys of software practitioners in several settings. Surveys have explored practitioners' attitudes, such as work habits [47] and motivation [29] as well as behavior [41]. Begel and Zimmermann [5] used a survey methodology to find questions that were of most interest to developers. Our work surveyed the perception of software practitioners and focused on topics related to defect prediction.

There exists a considerable body of work that surveyed the perception of software practitioners. Lo et al. surveyed hundreds of practitioners in Microsoft on how they perceive the relevance of 571 papers that were published in ICSE, ESEC/FSE and FSE from 2009 to 2014 [52]. They asked each respondent to rate 40 randomly selected papers by answering a question: *In your opinion, how important are the following pieces of research?* Different from them, we surveyed a more diverse population of software practitioners, who are from multiple corporations over 33 countries across 5 continents. Carver et al. replicated Lo et al. study to understand how practitioners perceived ESEM research [9]. Devanbu et al. surveyed hundreds of practitioners at Microsoft on their belief of empirical findings from software engineering research [16]. They compared the belief with actual empirical evidence and reported a large set of findings. We also compared global practitioners' belief with findings in previous studies on defect prediction. In addition, we particularly considered more in-depth questions on barriers for adoptions and expectations about defect prediction.

5.2 Adoption Factor Exploration

Tool adoption and spread within organizations and communities is a well-studied problem in software engineering and other research communities. Diffusion of Innovation theory is a popular framework for studying the transmission of ideas through communities [74]. Xiao et al. applies this framework to security tools [95]. Meyerovich and Rabkin investigated the factors related to spread and adoption of programming languages [55]. Wang et al. investigated the usefulness of information retrieval based techniques for real-world fault localization [89]. Kochhar et al. investigated practitioners' expectations of fault localization tools and estimated the adoption threshold [42]. Xia et al. [93] revisited the usefulness of spectra-based fault localization techniques with professionals. Parnin and Orso investigated how developers use and benefit from automated debugging tools through a set of human studies [63]. To our knowledge, we are the first to explore the factors that affect the adoption of defect prediction tools.

5.3 Empirical Studies on Defect Prediction

Factors affecting performance. Prior research examined the relationship between performance and some factors including data set [21], [30], [53], [72], [83], and input metrics [25], [96], [98], [101], [102]. Rahman et al. conducted an empirical analysis using simulated datasets to explore whether bias or size would affect performance of defect prediction more. Their results suggested size always matters just as much as bias direction, and much more than bias direction when considering information-retrieval measures [72]. Tantithamthavorn et al. studied whether mislabeling in defect prediction is random, and the impact of realistic mislabelling on the performance through a case study [83]. Herzig et al. discovered that misclassification introduces bias in defect prediction models and investigated the impact of misclassification on defect prediction studies [30]. Ghotra et al. used both a clear dataset and a noisy and biased dataset to evaluate the impact of classification techniques on the performance of defect prediction models [21]. Bettenburg et al. [6] and Menzies et al. [53], [54] generated and compared the lessons of defection prediction and effort estimation from local project, clustered projects, and all the projects. Shepperd et al. compared the relative contribution of factors that influence the performance of defect prediction, including learning technique, data set, input metrics, and research group [76]. Later Tantithamthavorn et al. performed an alternative investigation of Shepperd et al.'s data and made further observations and conclusions [85].

Metrics in models. Yang et al. conducted an empirical study to examine the usefulness of slice-based cohesion metrics in defect prediction [98]. Zhou et al. investigated the confounding effect of class size in defect prediction [102]. The work of Koru et al. has been instrumental in the recent understanding of size-defect relationship [43]–[45]. However, in Koru et al.'s work, the use of survival analysis in the context of defect modelling has not been well studied. Thus Syer et al. [81] replicated Koru et al.'s work and revisited the size-defect relationship. Hall et al. investigated the relationship between defects and code smells [25]. Zhao et al. conducted an empirical study to investigate the value of considering client usage context in package cohesion metrics in defect prediction [101]. Yang et al. investigated the relationship between inter-dependent program elements and function-level defect proneness [96].

Industrial studies. Industrial studies evaluated the actual adoption of defect prediction techniques. Nagappan and Ball presented an empirical approach to predict the actual pre-release defect density for Windows Server 2003 [59]. Zimmermann et al. investigated cross-project defect prediction and conducted large scale experiments on commercial systems from Microsoft, namely Direct-X, IIS, Printing, Windows Clustering, Windows File system, SQL Server 2005 and Windows Kernel [103]. Lewis et al. conducted a case study in Google to verify the effectiveness of several defect prediction algorithms, and further investigated whether an experimental defect prediction tool changed developers' behaviors [50]. Monden et al. conducted a case study in a software purchaser-side company to assess the cost effectiveness of defect prediction in test effort allocation [57]. Change-level defect prediction has been successfully adopted by industrial teams at Avaya [56], BlackBerry [37], [77] and Cisco [82].

6 Conclusion and Future Work

This paper explores the potential value of defect prediction in practice. We propose a mixed qualitative and quantitative approach to explore practitioners' perceptions, expectations and adoption challenges of defect prediction. We collected hypotheses regarding defect prediction from our open-ended interviews and papers that were published in the top-tier conferences and journals over a five year period. We conducted a survey to investigate those hypotheses and received 395 responses from practitioners from over 33 countries across five continents. Our results suggest that over 90% respondents are willing to adopt defect prediction techniques; but there is room for future research including fulfilling practitioners' expectations and resolving adoption challenges that are faced by practitioners.

Future studies should carefully compare our observations with state-of-the-art defect prediction studies and answer questions such as *how many studies are doing feature level defect prediction* and *how are defect prediction "models" evaluated nowadays*. Future work should consider developing defect prediction techniques that can bring current state-of-research closer to the expectations of practitioners highlighted in this work. Taking into account the response quality (i.e., respondents are less likely to give response of high quality for a very long survey), we tried to ensure an appropriate length for the survey done in this work. Future studies could answer other research questions, e.g., *how the primary programming language of practitioner affects his/her willingness of adoption*, *why practitioners prefer coarse granularity of defect prediction*, and *why more experienced practitioners are less enthused on defect prediction techniques*.

Acknowledgments

The authors would like to thank all survey participants for responding our survey.

References

[1] QDA Miner. https://provalisresearch.com/products/qualitative-data-analysis-software/freeware/. [Online; accessed 2018-04-25].

[2] F. Akiyama. An example of software system debugging. In *IFIP Congress (1)*, volume 71, pages 353–359, 1971.

[3] C. Andersson and P. Runeson. A replicated quantitative analysis of fault distributions in complex software systems. *IEEE Transactions on Software Engineering*, 33(5):273–286, May 2007.

[4] S. Apel and C. Kästner. An overview of feature-oriented software development. *Journal of Object Technology*, 8(5):49–84, 2009.

[5] A. Begel and T. Zimmermann. Analyze this! 145 questions for data scientists in software engineering. In *Proceedings of the 36th International Conference on Software Engineering*, pages 12–23. ACM, 2014.

[6] N. Bettenburg, M. Nagappan, and A. E. Hassan. Think locally, act globally: Improving defect and effort prediction models. In *Mining Software Repositories (MSR), 2012 9th IEEE Working Conference on*, pages 60–69. IEEE, 2012.

[7] B. Boehm and V. R. Basili. Software defect reduction top 10 list. *Computer*, 34(1):135–137, Jan. 2001.

[8] B. Caglayan, B. Turhan, A. Bener, M. Habayeb, A. Miransky, and E. Cialini. Merits of organizational metrics in defect prediction: an industrial replication. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 2, pages 89–98. IEEE, 2015.

[9] J. C. Carver, O. Dieste, N. A. Kraft, D. Lo, and T. Zimmermann. How practitioners perceive the relevance of esem research. In *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ESEM '16, pages 56:1–56:10, New York, NY, USA, 2016. ACM.

[10] T.-H. Chen, W. Shang, M. Nagappan, A. E. Hassan, and S. W. Thomas. Topic-based software defect explanation. *Journal of Systems and Software*, 129:79–106, 2017.

[11] B. Curtis, E. M. Soloway, R. E. Brooks, J. B. Black, K. Ehrlich, and H. R. Ramsey. Software psychology: The need for an interdisciplinary program. *Proceedings of the IEEE*, 74(8):1092–1106, 1986.

[12] J. Czerwonka, R. Das, N. Nagappan, A. Tarvo, and A. Teterev. Crane: Failure prediction, change analysis and test prioritization in practice–experiences from windows. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 357–366. IEEE, 2011.

[13] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan. A framework for evaluating the results of the szz approach for identifying bug-introducing changes. *IEEE Transactions on Software Engineering*, 43(7):641–657, 2017.

[14] K. Dejaeger, T. Verbraken, and B. Baesens. Toward comprehensible software fault prediction models using bayesian network classifiers. *IEEE Transactions on Software Engineering*, 39(2):237–257, 2013.

[15] P. Devanbu, T. Zimmermann, and C. Bird. Belief & evidence in empirical software engineering. In *Proceedings of the 38th international conference on software engineering*, pages 108–119. ACM, 2016.

[16] P. Devanbu, T. Zimmermann, and C. Bird. Belief and evidence in empirical software engineering. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 108–119, New York, NY, USA, 2016. ACM.

[17] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Trans. Softw. Eng.*, 26(8):797–814, Aug. 2000.

[18] Fisher and Ra. On the interpretation of $\chi 2$ from contingency tables, and the calculation of p. *Journal of the Royal Statistical Society*, 85:87–94, 1922.

[19] J. L. Fleiss. Measuring nominal scale agreement among many raters. *Psychological bulletin*, 76(5):378, 1971.

[20] W. Fu and T. Menzies. Revisiting unsupervised learning for defect prediction. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 72–83. ACM, 2017.

[21] B. Ghotra, S. McIntosh, and A. E. Hassan. Revisiting the impact of classification techniques on the performance of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 789–800, Piscataway, NJ, USA, 2015. IEEE Press.

[22] G. Gousios, M. Pinzger, and A. v. Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the 36th International Conference on Software Engineering*, pages 345–355. ACM, 2014.

[23] G. Guest, A. Bunce, and L. Johnson. How many interviews are enough? an experiment with data saturation and variability. *Field methods*, 18(1):59–82, 2006.

[24] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell. A systematic literature review on fault prediction performance in software engineering. *IEEE Trans. Softw. Eng.*, 38(6):1276–1304, Nov. 2012.

[25] T. Hall, M. Zhang, D. Bowes, and Y. Sun. Some code smells have a significant but small effect on faults. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 23(4):33:1–33:39, Sept. 2014.

[26] A. E. Hassan. Predicting faults using the complexity of code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 78–88. IEEE Computer Society, 2009.

[27] H. Hata, O. Mizuno, and T. Kikuno. Bug prediction based on fine-grained module histories. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 200–210, Piscataway, NJ, USA, 2012. IEEE Press.

[28] J. D. Herbsleb and A. Mockus. An empirical study of speed and communication in globally distributed software development. *IEEE Transactions on software engineering*, 29(6):481–494, 2003.

[29] G. Hertel, S. Niedner, and S. Herrmann. Motivation of software developers in open source projects: an internet-based survey of contributors to the linux kernel. *Research policy*, 32(7):1159–1177, 2003.

[30] K. Herzig, S. Just, and A. Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[31] C. Jaspan, M. Jorde, A. Knight, C. Sadowski, E. Smith, C. Winter, and E. Murphy-Hill. Advantages and disadvantages of a monolithic repository - a case study at google. In *Proceedings of the International Conference of Software Engineering: Software Engineering in Practice Track*, 2018.

[32] T. Jiang, L. Tan, and S. Kim. Personalized defect prediction. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, ASE'13, pages 279–289, Piscataway, NJ, USA, 2013. IEEE Press.

[33] X. Jing, F. Wu, X. Dong, F. Qi, and B. Xu. Heterogeneous cross-company defect prediction by unified metric representation and cca-based transfer learning. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 496–507, New York, NY, USA, 2015. ACM.

[34] X.-Y. Jing, F. Wu, X. Dong, and B. Xu. An improved sda based defect prediction framework for both within-project and cross-project class-imbalance problems. *IEEE Transactions on Software Engineering*, 43(4):321–339, 2017.

[35] X.-Y. Jing, S. Ying, Z.-W. Zhang, S.-S. Wu, and J. Liu. Dictionary learning based software defect prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 414–423. ACM, 2014.

[36] Y. Kamei and E. Shihab. Defect prediction: Accomplishments and future challenges. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 33–45, March 2016.

[37] Y. Kamei, E. Shihab, B. Adams, A. E. Hassan, A. Mockus, A. Sinha, and N. Ubayashi. A large-scale empirical study of just-in-time quality assurance. *IEEE Transactions on Software Engineering*, 39(6):757–773, 2013.

[38] S. Kim, T. Zimmermann, E. J. Whitehead Jr, and A. Zeller. Predicting faults from cached history. In *Proceedings of the 29th international conference on Software Engineering*, pages 489–498. IEEE Computer Society, 2007.

[39] B. A. Kitchenham and S. L. Pfleeger. Personal opinion surveys. In F. Shull, J. Singer, and D. I. K. Sjøberg, editors, *Guide to Advanced Empirical Software Engineering*, pages 63–92. Springer London, London, 2004.

[40] P. Knab, M. Pinzger, and A. Bernstein. Predicting defect densities in source code files with decision tree learners. In *Proceedings of the 2006 International Workshop on Mining Software Repositories*, MSR '06, pages 119–125, New York, NY, USA, 2006. ACM.

[41] A. J. Ko, R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, pages 344–353. IEEE, 2007.

[42] P. S. Kochhar, X. Xia, D. Lo, and S. Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, pages 165–176, New York, NY, USA, 2016. ACM.

[43] A. G. Koru, K. E. Emam, D. Zhang, H. Liu, and D. Mathew. Theory of relative defect proneness. *Empirical Software Engineering*, 13(5):473, Sep 2008.

[44] A. G. Koru, D. Zhang, K. E. Emam, and H. Liu. An investigation into the functional form of the size-defect relationship for software modules. *IEEE Transactions on Software Engineering*, 35(2):293–304, March 2009.

[45] A. G. Koru, D. Zhang, and H. Liu. Modeling the effect of size on defect proneness for open-source software. In *Proceedings of the Third International Workshop on Predictor Models in Software Engineering*, PROMISE '07, pages 10–, Washington, DC, USA, 2007. IEEE Computer Society.

[46] M. Lanza, A. Mocci, and L. Ponzanelli. The tragedy of defect prediction, prince of empirical software engineering research. *IEEE Software*, 33(6):102–105, 2016.

[47] T. D. LaToza, G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 492–501, New York, NY, USA, 2006. ACM.

[48] T. Lee, J. Nam, D. Han, S. Kim, and H. P. In. Developer micro interaction metrics for software defect prediction. *IEEE Transactions on Software Engineering*, 42(11):1015–1035, Nov 2016.

[49] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 372–381. IEEE Press, 2013.

[50] C. Lewis, Z. Lin, C. Sadowski, X. Zhu, R. Ou, and E. J. Whitehead Jr. Does bug prediction support human developers? findings from a google case study. In *Proceedings of the 2013 International Conference on Software Engineering*, ICSE '13, pages 372–381, Piscataway, NJ, USA, 2013. IEEE Press.

[51] A. Liaw, M. Wiener, et al. Classification and regression by randomforest. *R news*, 2(3):18–22, 2002.

[52] D. Lo, N. Nagappan, and T. Zimmermann. How practitioners perceive the relevance of software engineering research. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 415–425, New York, NY, USA, 2015. ACM.

[53] T. Menzies, A. Butcher, D. Cok, A. Marcus, L. Layman, F. Shull, B. Turhan, and T. Zimmermann. Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on Software Engineering*, 39(6):822–834, June 2013.

[54] T. Menzies, A. Butcher, A. Marcus, T. Zimmermann, and D. Cok. Local vs. global models for effort estimation and defect prediction. In *Automated Software Engineering (ASE), 2011 26th IEEE/ACM International Conference on*, pages 343–351. IEEE, 2011.

[55] L. A. Meyerovich and A. S. Rabkin. Empirical analysis of programming language adoption. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '13, pages 1–18, New York, NY, USA, 2013. ACM.

[56] A. Mockus and D. M. Weiss. Predicting risk of software changes. *Bell Labs Technical Journal*, 5(2):169–180, 2000.

[57] A. Monden, T. Hayashi, S. Shinoda, K. Shirai, J. Yoshida, M. Barker, and K. Matsumoto. Assessing the cost effectiveness of fault prediction in acceptance testing. *IEEE Transactions on Software Engineering*, 39(10):1345–1357, Oct 2013.

[58] E. Murphy-Hill, T. Zimmermann, C. Bird, and N. Nagappan. The design space of bug fixes and how developers navigate it. *IEEE Transactions on Software Engineering*, 41(1):65–81, Jan 2015.

[59] N. Nagappan and T. Ball. Static analysis tools as early indicators of pre-release defect density. In *Proceedings of the 27th international conference on Software engineering*, pages 580–586. ACM, 2005.

[60] J. Nam and S. Kim. Clami: Defect prediction on unlabeled datasets (t). In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 452–463. IEEE, 2015.

[61] J. Nam and S. Kim. Heterogeneous defect prediction. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 508–519, New York, NY, USA, 2015. ACM.

[62] J. Nam, S. J. Pan, and S. Kim. Transfer defect learning. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 382–391. IEEE Press, 2013.

[63] C. Parnin and A. Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 international symposium on software testing and analysis*, pages 199–209. ACM, 2011.

[64] F. Peters and T. Menzies. Privacy and utility for defect prediction: Experiments with morph. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 189–199, Piscataway, NJ, USA, 2012. IEEE Press.

[65] F. Peters, T. Menzies, L. Gong, and H. Zhang. Balancing privacy and utility in cross-company defect prediction. *IEEE Transactions on Software Engineering*, 39(8):1054–1068, 2013.

[66] F. Peters, T. Menzies, and L. Layman. Lace2: Better privacy-preserving data sharing for cross project defect prediction. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 801–811. IEEE Press, 2015.

[67] M. Pinzger, N. Nagappan, and B. Murphy. Can developer-module networks predict failures? In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 2–12. ACM, 2008.

[68] F. Rahman and P. Devanbu. Ownership, experience and defects: a fine-grained study of authorship. In *Proceedings of the 33rd International Conference on Software Engineering*, pages 491–500. ACM, 2011.

[69] F. Rahman and P. Devanbu. How, and why, process metrics are better. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 432–441. IEEE Press, 2013.

[70] F. Rahman, S. Khatri, E. T. Barr, and P. Devanbu. Comparing static bug finders and statistical prediction. In *Proceedings of the 36th International Conference on Software Engineering*, pages 424–434. ACM, 2014.

[71] F. Rahman, D. Posnett, and P. Devanbu. Recalling the imprecision of cross-project defect prediction. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 61. ACM, 2012.

[72] F. Rahman, D. Posnett, I. Herraiz, and P. Devanbu. Sample size vs. bias in defect prediction. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 147–157, New York, NY, USA, 2013. ACM.

[73] E. S. Raymond. *The Cathedral & the Bazaar: Musings on linux and open source by an accidental revolutionary.* " O'Reilly Media, Inc.", 2001.

[74] E. M. Rogers. Diffusion of innovations, 1995.

[75] G. Scanniello, C. Gravino, A. Marcus, and T. Menzies. Class level fault prediction using software clustering. In *Proceedings of the 28th IEEE/ACM International Conference on Automated Software Engineering*, pages 640–645. IEEE Press, 2013.

[76] M. Shepperd, D. Bowes, and T. Hall. Researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 40(6):603–616, June 2014.

[77] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang. An industrial study on the risk of software changes. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 62. ACM, 2012.

[78] S. Shivaji, E. J. Whitehead, R. Akella, and S. Kim. Reducing features to improve code change-based bug prediction. *IEEE Transactions on Software Engineering*, 39(4):552–569, 2013.

[79] Q. Song, Z. Jia, M. Shepperd, S. Ying, and J. Liu. A general software defect-proneness prediction framework. *IEEE Transactions on Software Engineering*, 37(3):356–370, May 2011.

[80] D. Spencer. *Card sorting: Designing usable categories*. Rosenfeld Media, 2009.

[81] M. D. Syer, M. Nagappan, B. Adams, and A. E. Hassan. Replicating and re-evaluating the theory of relative defect-proneness. *IEEE Transactions on Software Engineering*, 41(2):176–197, Feb 2015.

[82] M. Tan, L. Tan, S. Dara, and C. Mayeux. Online defect prediction for imbalanced data. In *Proceedings of the 37th International Conference on Software Engineering - Volume 2*, ICSE '15, pages 99–108, Piscataway, NJ, USA, 2015. IEEE Press.

[83] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto. The impact of mislabelling on the performance and interpretation of defect prediction models. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 812–823, Piscataway, NJ, USA, 2015. IEEE Press.

[84] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE), 2016 IEEE/ACM 38th International Conference on*, pages 321–332. IEEE, 2016.

[85] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. Comments on researcher bias: The use of machine learning in software defect prediction. *IEEE Transactions on Software Engineering*, 42(11):1092–1094, 2016.

[86] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. An empirical comparison of model validation techniques for defect prediction models. *IEEE Transactions on Software Engineering*, 43(1):1–18, 2017.

[87] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, and K. Matsumoto. The impact of automated parameter optimization on defect prediction models. *IEEE Transactions on Software Engineering*, 2018.

[88] P. K. Tyagi. The effects of appeals, anonymity, and feedback on mail survey response patterns from salespeople. *Journal of the Academy of Marketing Science*, 17(3):235–241, Jun 1989.

[89] Q. Wang, C. Parnin, and A. Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, pages 1–11. ACM, 2015.

[90] S. Wang, T. Liu, and L. Tan. Automatically learning semantic features for defect prediction. In *Proceedings of the 38th International Conference on Software Engineering*, ICSE '16, pages 297–308, New York, NY, USA, 2016. ACM.

[91] E. J. Weyuker, T. J. Ostrand, and R. M. Bell. Do too many cooks spoil the broth? using the number of developers to enhance defect prediction models. *Empirical Software Engineering*, 13(5):539–559, 2008.

[92] J. Witschey, O. Zielinska, A. Welk, E. Murphy-Hill, C. Mayhorn, and T. Zimmermann. Quantifying developers' adoption of security tools. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 260–271, New York, NY, USA, 2015. ACM.

[93] X. Xia, L. Bao, D. Lo, and S. Li. automated debugging considered harmful considered harmful: A user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*, pages 267–278. IEEE, 2016.

[94] X. Xia, D. Lo, S. J. Pan, N. Nagappan, and X. Wang. Hydra: Massively compositional model for cross-project defect prediction. *IEEE Transactions on software Engineering*, 42(10):977–998, 2016.

[95] S. Xiao, J. Witschey, and E. Murphy-Hill. Social influences on secure development tool adoption: Why security tools spread. In *Proceedings of the 17th ACM Conference on Computer Supported Cooperative Work and Social Computing*, CSCW '14, pages 1095–1106, New York, NY, USA, 2014. ACM.

[96] Y. Yang, M. Harman, J. Krinke, S. Islam, D. Binkley, Y. Zhou, and B. Xu. An empirical study on dependence clusters for effort-aware fault-proneness prediction. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE 2016, pages 296–307, New York, NY, USA, 2016. ACM.

[97] Y. Yang, Y. Zhou, J. Liu, Y. Zhao, H. Lu, L. Xu, B. Xu, and H. Leung. Effort-aware just-in-time defect prediction: Simple unsupervised models could be better than supervised models. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE 2016, pages 157–168, New York, NY, USA, 2016. ACM.

[98] Y. Yang, Y. Zhou, H. Lu, L. Chen, Z. Chen, B. Xu, H. Leung, and Z. Zhang. Are slice-based cohesion metrics actually useful in effort-aware post-release fault-proneness prediction? an empirical study. *IEEE Transactions on Software Engineering*, 41(4):331–357, April 2015.

[99] F. Zhang, A. E. Hassan, S. McIntosh, and Y. Zou. The use of summation to aggregate software metrics hinders the performance of defect prediction models. *IEEE Transactions on Software Engineering*, 43(5):476–491, 2017.

[100] F. Zhang, Q. Zheng, Y. Zou, and A. E. Hassan. Cross-project defect prediction using a connectivity-based unsupervised classifier. In *Proceedings of the 38th International Conference on Software Engineering*, pages 309–320. ACM, 2016.

[101] Y. Zhao, Y. Yang, H. Lu, J. Liu, H. Leung, Y. Wu, Y. Zhou, and B. Xu. Understanding the value of considering client usage context in package cohesion for fault-proneness prediction. *Automated Software Engineering*, 24(2):393–453, Jun 2017.

[102] Y. Zhou, B. Xu, H. Leung, and L. Chen. An in-depth study of the potentially confounding effect of class size in fault prediction. *ACM Trans. Softw. Eng. Methodol.*, 23(1):10:1–10:51, Feb. 2014.

[103] T. Zimmermann, N. Nagappan, H. Gall, E. Giger, and B. Murphy. Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 91–100, New York, NY, USA, 2009. ACM.

[104] T. Zimmermann, R. Premraj, and A. Zeller. Predicting defects for eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society, 2007.