Preprint of article published in IEEE Software April 2021

Department: Head
Editor: Name, xxxx@email

Actionable Analytics: Stop Telling Me What Is; Please Tell Me What To Do!

Chakkrit (Kla) Tantithamthavorn
Monash University, Australia.

Jirayus Jiarpakdee
Monash University, Australia.

John Grundy
Monash University, Australia.

The success of software projects depends on complex decision-making (e.g. which tasks should a developer do first, who should perform this task, is the software of high quality, is a software system reliable and resilient enough to deploy, etc). Bad decisions cost money (and reputation) so we need better tools for making better decisions. This paper discusses the "why" and "how" of explainable and actionable software analytics. For the task of reducing the risk of software defects, we show initial results from a successful case study that offers more actionable software analytics. Also, we present an interactive tutorial on the subject of Explainable AI tools for SE in our Software Analytics Cookbook (http://analytics-cookbook.github.io), and discuss some open questions that need to be addressed.

STOP TELLING ME WHAT IS.

While the adoption of software analytics enables software organizations to distill actionable insights and support decision-making, there are still many barriers to the successful adoption of such software analytics in software organizations [2].

First, most software practitioners do not understand the reason behind the predictions from software analytics [2]. They often ask the following questions:

- Why is this person best suited for this task?
- Why is this file predicted as defective?
- Why is this task required the highest development effort?
- Why should this task be done first?
- Why is this developer predicted to have low productivity?
- How can we improve the quality of software systems in following iterations?

These concerns about a lack of explanation often leads to a lack of trust and transparency, hindering the adoption of software analytics in practice.

Also, our recent work also found that prac-

titioners are very concerned about their privacy and fairness if defect prediction models were deployed in practice. Practitioners even asked "Would developers be laid-off due to the use of defect prediction models for identifying who are most likely to introduce software defects?" [4]. Article 22 of the European Union's General Data Protection Regulation (GDPR) states that the use of data in decision-making that affects an individual or group requires an **explanation** for any decision made by an algorithm. Unfortunately, current software analytics still often do not uphold privacy laws [3]. Thus, the risks of unjustified decision-making of software analytics systems can be catastrophic, leading to potentially erroneous and costly business decisions [2].

Is the Community Moving to the Right Direction?

Researchers' Focuses

We conducted a literature analysis to better understand what researchers are currently focusing on. We select defect prediction as a target research topic as it is one of the most active research topics in software engineering. We collected 96 primary defect prediction studies that were published in top-tier Software Engineering venues (i.e., TSE, ICSE, EMSE, FSE, and MSR) during 2015-2020 (as of 11 January 2021). We then characterized the key goals of each defect prediction study into three main goals: (1) predictions; (2) model explanation; and (3) instance explanation [4] (see Figure 1). We found that 91% (81/96) of the defect prediction studies only focus on making predictions, without considering explaining the predictions. As few as 4% of the defect prediction studies focus on explaining the predictions of defect prediction models (see Figure 1). This indicates that the explainability and actionability of software analytics is still very under researched.

Practitioners' Needs

We conducted a qualitative survey to better understand what practitioners perceive as the usefulness of each goal of defect prediction models. We found that practitioners perceive that the explainability and actionability of software analytics are as **equally useful** as the predictions [4]. 82% of our respondents said that the explanability goal (generating model explanations and instance explanations) is as useful as the prediction goal. Thus, we argue that **explainable and actionable software analytics is urgently and critically needed**. The research and practice communities should thus start answering the key question: "how can we make software analytics more explainable and actionable?".

Explainable AI for SE: A Way Forward

Explainable AI is a suite of AI/ML techniques that produce accurate predictions, while being able to explain such predictions. The purpose of increasing the explainability of software analytics (XAI4SE) is to make its behavior more intelligible to humans by providing explanations. The explainability of software analytics can be achieved by:

- **"Global Explanability":** Using interpretable machine learning techniques (e.g., decision tree, decision rules or logistic regression techniques) or intrinsic model-specific techniques (e.g., ANOVA, variable importance) so the entire predictions and recommendations process are transparent and comprehensible. Such intrinsic model-specific techniques aim to provide the global explainability. Thus, users can only understand how the model works globally (e.g., by inspecting a branch of decision trees). However, users often do not understand why the model makes that prediction.
- **"Local Explanability":** Using model-agnostic techniques (e.g., LIME [10]) to explain the predictions of the software analytics models (e.g., neural network, random forest). Such post-hoc model-agnostic techniques can provide an explanation for each prediction (i.e., an instance to be explained). Users can then understand why the prediction is made by the software analytics models.

PLEASE TELL ME WHAT TO DO!

We discuss some initial evidence from a successful case study of using Explainable AI for Software Engineering in the context of software defect prediction [9].

Background. In today's increasingly digitalized world, software defects are widespread and enormously expensive, but they are very hard to detect, predict, and prevent. Thus, a failure

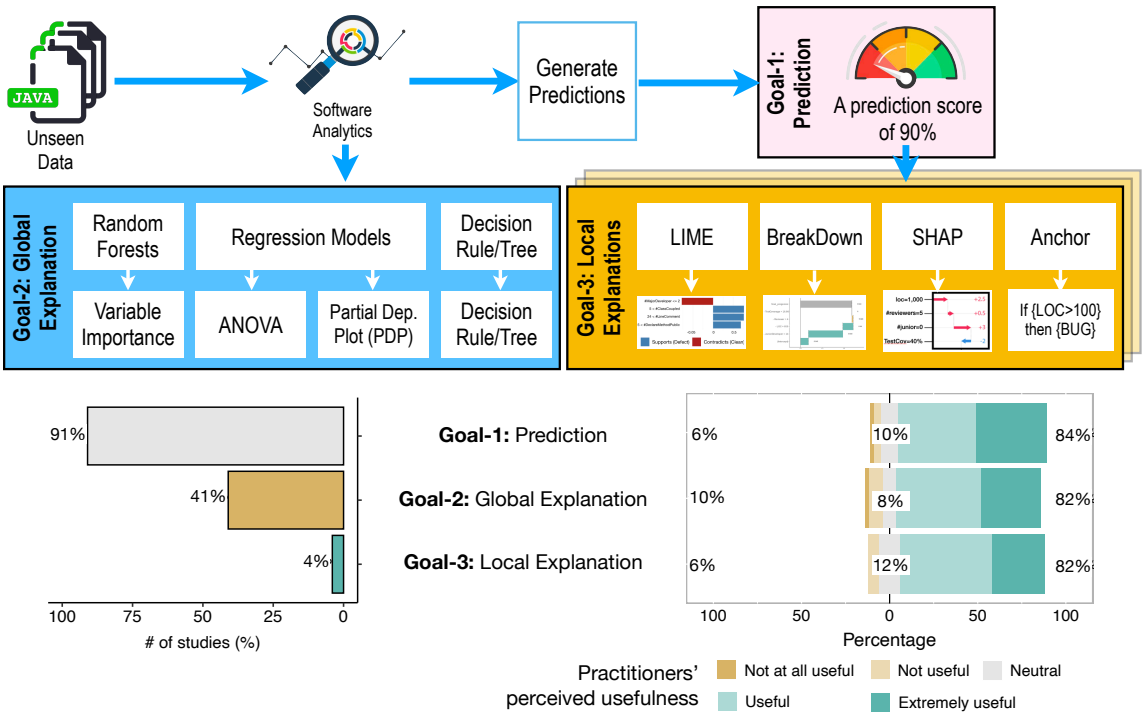

Figure 1: Most software analytics (i.e., defect prediction) studies have three main goals: (1) predictions; (2) model explanation; and (3) instance explanation. We found that 82% of our study respondents perceived the explanability goal (generating model explanations and instance explanations) is equally useful as the prediction goal. However, we found that 91% (81/96) of defect prediction studies only focus on the prediction goal, and as few as 4% of defect prediction studies focus on the explainability goal.

to eliminate software defects in safety-critical systems could result in serious injury to people, threats to life, death, and disasters.

Traditionally, software quality assurance activities – software testing and code review – are widely adopted to discover software defects in software systems. However, ultra-large-scale systems, such as, Google, can have more than two billion lines of code. Thus, exhaustively reviewing and testing every single line of code is not feasible with limited time and resources. Defect prediction—an AI/ML model trained on historical data to predict if a file/commit will be defective in the future—has been proposed to help developers prioritize their limited software quality assurance resources on the most risky files/commits.

Gaps. However, developers often do not understand why a file is being predicted as defective. In addition, such predictions and global explanations are still often not actionable [5]—i.e., developers do not know what to actually do – or what to avoid doing – in order to improve the quality of the software system. These limitations often lead to a lack of trust in the predictions, hindering the adoption of defect prediction models in practice.

Let's consider a scenario where a defect prediction model indicates that *file size* is associated with defect-proneness i.e. code in bigger files is likely to be more defective. This insight may help managers develop quality improvement plans that, in the next development iteration, developers should pay attention to the file size to mitigate the risk of having defects. However, such insights do not provide a concrete suggestion of what to do and what to avoid (i.e. does increasing or decreasing size reduce or increase defects?). Thus, a lack of such actionable guidance remains an extremely challenging problem, often leading to ineffective software quality improvement plans.

Approach. To generate actionable guidance,

we used a rule-based model-agnostic technique to generate a rule-based explanation for each prediction of defect prediction models [9]. We first build file-level defect prediction models that are trained using traditional software features (e.g., lines of code, code complexity, the number of developers who edited a file) with a random forest classification technique. For each prediction, we applied a rule-based model-agnostic technique to generate two types of actionable guidance (i.e., what developers should do to mitigate the risk of having defects and what developers should not do to avoid increasing the risk of having defects).

Visualizing the actionable guidance. For each guidance, we translated a rule-based explanation into an actionable guidance. Then, we developed a proof-of-concept to visualize the actionable guidance using a bullet plot (see Figure 2). The visualization is designed to provide the following key information: (1) the list of guidance that practitioners should follow and should avoid; (2) the actual feature value of that file; and (3) its threshold and range values for practitioners to follow to mitigate the risk of having defects. The green shades indicate the non-risky range values of features, while the red shades indicate the risky range values of features. The vertical bars indicate the actual values of features for a given file. The green arrows provide directions of how a feature should be changed (i.e., increase or decrease). The provided guidance is structured into two parts: (1) what to do to *decrease the risk* of being defective; and (2) what to avoid to *not increase the risk* of being defective.

Figure 2 shows an example of such actionable guidance to help managers develop their quality improvement plans. In this example, to decrease the risk of having defects, developers should consider (1) decrease the number of class and method declaration lines to less than 29 lines, (2) decrease the number of distinct developers to less than 2 developers, (3) increase the proportion of code ownership to more than 0.85, (4) decrease the number of blank lines to less than 8 lines, (5) decrease the number of output variables to less than 2 variables. Nevertheless, to not increase the risk of having defects, developers should consider avoid decreasing the comment to code ratio and avoid increasing the number of minor or junior developers.

User Study Evaluation. We conducted a qualitative survey to investigate the practitioners' perceptions of our visualization approach. We also used the visualization of Microsoft's Code Defect AI as a baseline comparison.

Results. We found that 80% of our respondents agree that our visualization is better for providing actionable guidance when compared to the visualization of Microsoft's Code Defect AI. In addition, 63%-90% of the respondents agree with the seven actionable guidance provided by our approach.

Lessons Learned

Based on these findings, we summarise our key lessons learned:

1) Explainable AI is very important in software engineering, but is still under-researched [4].
2) Explainable AI techniques can be used in software engineering to provide explanations of the predictions and actionable guidance to support software engineering tasks [9]. Other initial successful evidence can be found at [1], [3], [6], [7], [8], [11].

However, there are several open research questions that remain largely unexplored:

- What is the best form of explanations for software engineering tasks that are most understandable to software developers?
- Do different stakeholders need different forms of explanations in software engineering?
- How can we measure the quality and value of explanations in software engineering?
- How do explanations from explainable AI domains impact software engineering practices?
- What are the other application areas in software engineering that need explanations?
- How can we improve the explanations for other complex AI/ML algorithms (e.g., deep learning, optimization, natural language processing) to address other software engineering tasks?

We developed an interactive tutorial of Explainable AI tools for SE to support future research. This can be found at our Software Analytics Cookbook http://analytics-cookbook.github.io. We hope this article will motivate future work to address these important questions, which require the SE community to work together with other disciplines and communities (e.g., Explain-

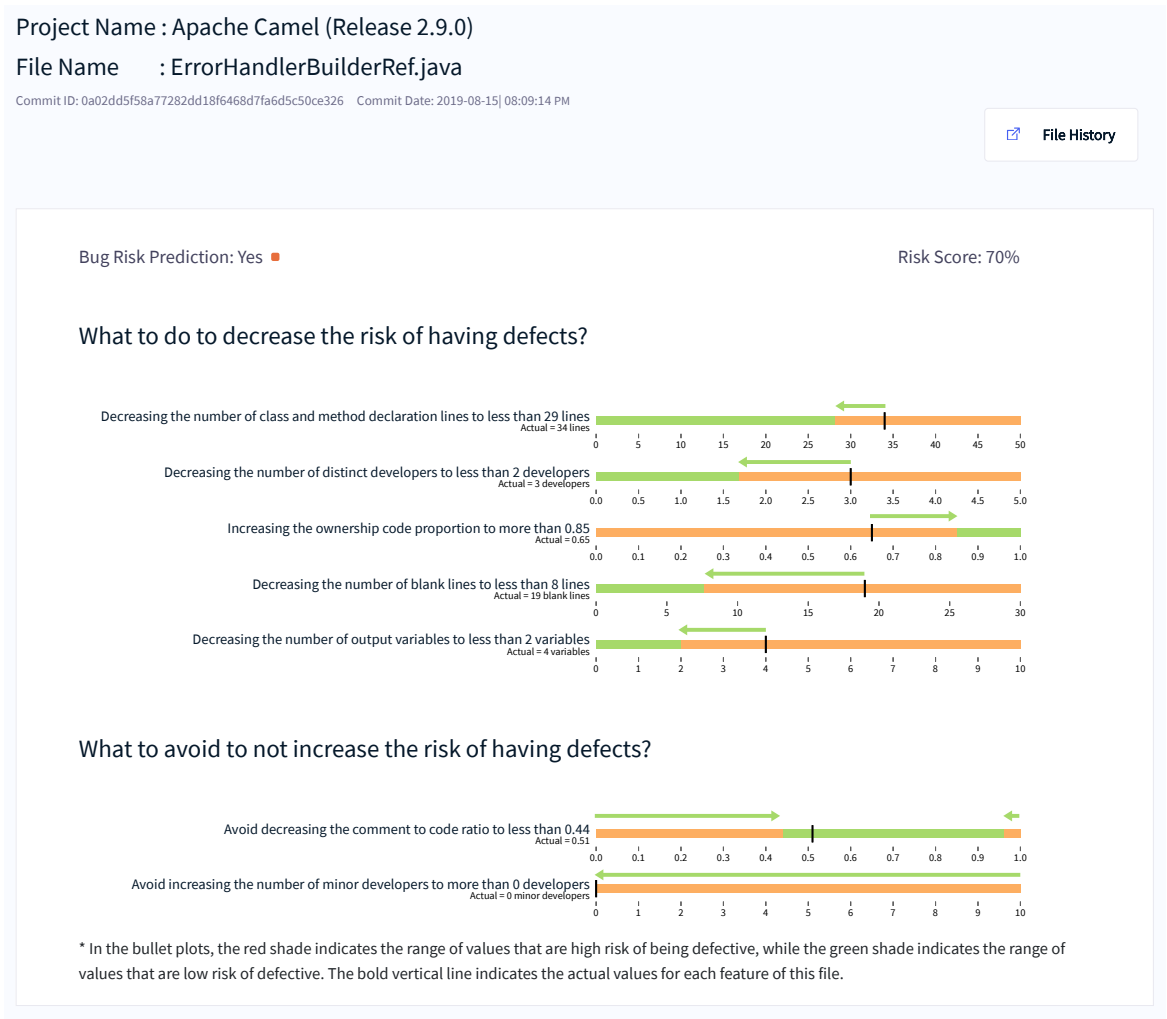

Figure 2: An example of actionable guidance to help managers developing quality improvement plans.

able AI, Visual Analytics, Natural Language Processing, Human-Centric Software Engineering).

ACKNOWLEDGMENT

We thank a Guest Editor Professor Tim Menzies and our collaborators for their joint work including Supatsara Wattanakriengkrai, Patanamon Thongtanunam, Hideaki Hata, Kenichi Matsumoto, Hoa Khanh Dam, Chanathip Pornprasit, Dilini Rajapaksha, Christoph Bergmeir, and Wray Buntine.

C. Tantithamthavorn was partially supported by the Australian Research Council's Discovery Early Career Researcher Award (ARC DECRA) funding scheme (DE200100941). J. Grundy was partially supported by the Australian Research Council's Laureate Fellowship funding scheme (FL190100035).

References

1. D. Chen, W. Fu, R. Krishna, and T. Menzies, "Applications of Psychological Science for Actionable Analytics," in *Proceedings of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018, pp. 456–467.
2. H. K. Dam, T. Tran, and A. Ghose, "Explainable Software Analytics," in *Proceedings of the International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, 2018, pp. 53–56.
3. J. Jiarpakdee, C. Tantithamthavorn, H. K. Dam, and J. Grundy, "An Empirical Study of Model-Agnostic Techniques for Defect Prediction Models," *Transactions on Software Engineering (TSE)*, p. To Appear, 2020.
4. J. Jiarpakdee, C. Tantithamthavorn, and J. Grundy,

"Practitioners' Perceptions of the Goals and Visual Explanations of Defect Prediction Models," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.

5. J. Jiarpakdee, C. Tantithamthavorn, and A. E. Hassan, "The Impact of Correlated Metrics on Defect Models," *Transactions on Software Engineering (TSE)*, p. To Appear, 2019.
6. R. Krishna and T. Menzies, "From prediction to planning: Improving software quality with belltree," *Empirical Software Engineering*, 2020.
7. K. Peng and T. Menzies, "Defect reduction planning (using timelime)," *IEEE Transactions on Software Engineering*, 2021.
8. C. Pornprasit and C. Tantithamthavorn, "JITLine: A Simpler, Better, Faster, Finer-grained Just-In-Time Defect Prediction," in *Proceedings of the International Conference on Mining Software Repositories (MSR)*, 2021, p. To Appear.
9. D. Rajapaksha, C. Tantithamthavorn, J. Jiarpakdee, C. Bergmeir, J. Grundy, and W. Buntine, "SQAPlanner: Generating Data-Informed Software Quality Improvement Plans," *IEEE Transactions on Software Engineering (TSE)*, 2021.
10. M. T. Ribeiro, S. Singh, and C. Guestrin, "Why should I trust you?: Explaining the Predictions of Any Classifier," in *Proceedings of the International Conference on Knowledge Discovery and Data Mining (KDD)*, 2016, pp. 1135–1144.
11. S. Wattanakriengkrai, P. Thongtanunam, C. Tantithamthavorn, H. Hata, and K. Matsumoto, "Predicting defective lines using a model-agnostic technique," *IEEE Transactions on Software Engineering (TSE)*, 2020.

Jirayus Jiarpakdee is a Ph.D. candidate at Monash University, Australia. His research interests include empirical software engineering and mining software repositories (MSR). The goal of his Ph.D. is to apply the knowledge of statistical modelling, experimental design, and software engineering to improve the explainability of defect prediction models. Contact him at jirayus.jiarpakdee@monash.edu.

Chakkrit Tantithamthavorn is a Senior Lecturer in Software Engineering and a 2020 ARC DECRA Fellow in the Faculty of Information Technology, Monash University, Australia. His current fellowship is focusing on the development of "Practical and Explainable Analytics to Prevent Future Software Defects". His work has been published at several top-tier software engineering venues, such as TSE, ICSE, EMSE, MSR, IST. Contact him at chakkrit@monash.edu.

John Grundy is Australian Laureate Fellow and Professor of Software Engineering at Monash University, Australia. He has published widely in automated software engineering, domain-specific visual languages, model-driven engineering, software architecture, and empirical software engineering, among many other areas. He is Fellow of Automated Software Engineering and Fellow of Engineers Australia. Contact him at john.grundy@monash.edu.