



Expert Perspectives on Explainability

Jürgen Cito, Satish Chandra, Chakkrit Tantithamthavorn¹, Hadi Hemmati

JÜRGEN CITO IS interviewing Vijayaraghavan Murali (VM), a software engineer at Meta, and Eddie Aftandilian (EA), a principal researcher at GitHub Next.

Q. What do you think is the role of machine learning (ML) and artificial intelligence (AI) in the broader sense in relation to software engineering?

VM: ML is going to play a very critical role in pretty much all stages of software engineering. Typically, we talk about the software engineering cycle as having an inner loop and an outer loop. The inner loop is where developers write code, debug it, test it, and build it. They produce an artifact we call a *diff* or a *commit*. And then there's an outer loop, where continuous integration runs automated tests and, additionally, a code review is performed on that artifact. Further stages include the product release and potentially the debugging and attribution of postproduction bugs, which constitute the outer loop. We have initially focused on the outer loop with projects like Minesweeper,¹ trying to attribute post release bugs and

crashes. We are now also starting to focus on the inner loop, where people are actually authoring code, and on what we can do to use models to assist them in writing code faster, for example, by performing code search.

EA: I think AI and ML are becoming a critical part of the software engineering process. We're already seeing an impact on how developers write code with tools like GitHub Copilot. And I think we're just starting to see the impact of tools like ChatGPT on answering questions, especially answering technical questions. For instance, how do I do X in PyTorch? ChatGPT will give me a pretty good answer. And I don't have to read pages and pages of PyTorch documentation.

Q. What is the spectrum of models and software artifacts you're seeing?

EA: In the near future, we're going to see ML transform many other aspects of the software development process, for example, of all the things that software developers do that are not just writing code. They're doing code reviews, they're debugging issues, they're fixing bugs, and they're writing documentation. We're right on the cusp of ML transforming how

those activities are done. It's interesting to ask whether all of those activities still exist in a world with very smart ML models. Do I need a human to review my code if I have a model that can review my code synchronously with me? Or maybe the model just never suggests the bad code that I would have written and doesn't need to be reviewed. I see documentation as another example of that. If a model can do a good job generating documentation from source code, does the human ever have to write documentation? Maybe not. This is all very speculative, and who knows how much of all this will pan out. At the pace at which we're seeing AI improve today, things will shift very soon.

VM: We have a lot of interest in modeling all kinds of software artifacts that are produced by developers. For instance, we are looking at code commits, which are different in the distribution than other code because they constitute a particular unit of code that a developer deems complete, rather than incomplete code as they are typing and forming an idea at the same time (in the context of generative models). We are also looking at what happens in code review: comments that reviewers make, requests

Digital Object Identifier 10.1109/MS.2023.3255663
Date of current version: 18 April 2023

for changes, and these sorts of signals. We are also looking to capture discussions that happen around code in internal discussion forums (a bit like Stack Overflow) as part of our models. This is particularly useful to teach models how natural language interacts with code elements. We are also looking at crashes or bug reports that are collected through telemetry. Training our models with these various artifacts can aid in generative tasks, like helping developers write code or addressing code review comments automatically. We also build models that are discriminative in nature, for example, attributing a particular regression to a team.

Q. Where do you see the role of explainability in those powerful models?

EA: I've been thinking about this question, but from the perspective of someone building the tools. Currently, the process, especially for prompt engineering, is very trial and error based. It's in its early days, and we're continuing to evolve our principles around it. You try random things and you need some way to evaluate them. If they work, they work; if they don't, they don't. And you have no idea why. So, as someone who has spent a lot of time crafting prompts, it'd be really helpful for me to know why a certain generation was wrong. What about this prompt caused this incorrect generation? And how could I change this prompt to get the generation I want? From the user's perspective, one thing I observed from my use of Copilot over time is that I've learned how to redirect it when it gives me the wrong generation. And I do that by writing comments. Often I'll first look at what it suggests for me, and if it's not what I want, I'll write a comment, telling it very specifically what I want.

From there, it does give it to me, but that's not very discoverable. In that sense, it would be nice if somehow the model could tell the user: I need to know more to give you the generation that you're looking for.

Sometimes I would like to know: Did Copilot produce this generation because there was an example of something similar somewhere in the context? Or is it because I gave it the definition of the application programming interface (API) or saw the definition of the API somewhere and read the documentation and learned how to use it? Or did it just see this kind of thing in the training set? At a user level, it would be nice to collect a set of tips and tricks of how to make the best use of Copilot, and that probably already exists in some GitHub repo. Or maybe you could automate that a bit so that the user doesn't have to read a document to learn how to do things. If Copilot itself could provoke you and push you into the right path for it to give good completions, I think that would be helpful to a lot of users.

In some of the literature where people have investigated productivity analyses of Copilot, it seems that new programmers, like junior programmers and undergraduates, tend to have more trouble with it than experienced programmers. I wonder if that is a case where new programmers are just not experienced using the tool? Are they not prompting it in the right way and then getting low-quality completions that take them longer to verify? As you can see, there's a lot we're looking to understand here.

VM: I think explainability is going to become a really important feature because we need to build trust among developers. There are certain tasks for which I do not see an immediate need for explainability from a user's

perspective, for example, code generation. When that happens right in the ideation stage, developers are constantly in the flow typing code, and we are using a model to suggest the next few tokens or maybe a few lines of code that the developer has to write. In that kind of very fast-paced setting, explainability is tricky to pull in. It's very hard to show the developer that piece of code and also an explanation about why the model generated that code. It doesn't really fit into that kind of product. The understanding is that the model may not be entirely accurate. And past literature has shown that developers are fine with reworking some of those additions, as long as they don't need to rewrite the entire thing. So, as long as the suggestions are sort of accurate, developers are fine with accepting it and then reworking things here and there.

But there are other settings where explainability is really critical—where there is less involvement from the human side. For instance, sometimes we are operating in a setting where we are trying to help automate code review comments. In that case, the involvement of the human is not as tight as in code authoring because code review is asynchronous. It happens offline. The goal would be to see if we can use an AI system to automatically suggest some patches for review comments that are suggested. This is also similar for discriminatory models that make judgments about code, for example, bug localization, detecting privacy-sensitive dataflow, or unoptimized code. In those kinds of settings, it is really critical for the model to offer an explanation along with the prediction because, if you're just pointing to a line of code and saying, "Oh, this line of code has a bug," or "This line of code is not optimized," developers would want

to know why you came up with this prediction. The explanation serves an educational purpose and also builds trust in the model. In these other kinds of applications involved in very similar technology, explainability is going to be very critical. To follow up on what I mentioned earlier: I think in a setting of the inner loop, where you write code and you're getting suggestions from the AI, that might not be the right time to induce an explanation, but it would definitely be right in stages of the outer loop.

Q. What would be the properties of your ideal explainability tool?

VM: There are certain applications where the explainability needs to be really fast, but I would say that, typically, when we think of a discriminatory problem, we would imagine that the model would come up with some probability numbers. To me, the main incarnation of explainability should be that rather than just coming up with a number, the model should also be able to point out and say why it specifically came up with that number. It could be as simple as pointing to certain parts of the input that caused it to come up with that prediction. Or it could be more conversational in nature, like what ChatGPT currently does. It explicitly asked the system why it made a particular prediction, and the model comes up with a natural language explanation for why it made the prediction. For instance, you could provide a piece of code, tell it is there a bug in this code, and ask it to explain it; literally, "What is the bug there?" And it actually offers an elaborate explanation. However, that explanation could itself be faulty, but confident.

EA: Interactivity, for example, interactive speed, would be really helpful.

If an explainability tool could operate in the integrated development environment while I'm working and proactively tell me, "If you did this, you'd get a better completion," that would be really helpful. It's much better to try to teach these things in context when the users are actually trying to do something. Precision is very important. It would be really frustrating if I were told: this is why you got this generation, and then as a result I tried to change it and still didn't get what was expected. You do want to have high precision. You don't want to ask yourself whether your debugger is buggy.

Q. What do you think are good ways we can systematically evaluate and measure the quality of these AI systems and also the explanations that we produce?

EA: Evaluating these sorts of natural language processing (NLP)-type things is very difficult. In terms of evaluating model outputs, we try to execute the code and compare them to test cases and those kinds of things. One of the things that I think is interesting about the latest large language models is that they can kind of act like human raters by themselves. You can ask them to compare two paragraphs of text and determine whether they are factually consistent. And they can kind of answer you. It'll be interesting to see if this all evolves into models producing output and then other models evaluating the quality of the outputs, like reinforcement learning from human feedback. There are various techniques from the NLP literature that could be useful for validating the accuracy of explanations. Potentially, we could perform back translation, where you provide the prompt or some context, then you

give the explanation, and then you check whether the explanation follows from the prompt. I'm wondering if there's something clever here you can do with the models themselves to evaluate the outputs.

VM: I think it basically would boil down to user adoption rate in terms of online metrics, that is, essentially conducting some A/B experiments on how often a particular metric is moved. Let's say you are predicting if a particular piece of code is optimized or not. One particular metric we can track there is how often a user made some changes based on prediction of the model. Essentially, we want to compare two things: how often users are taking an action based on the prediction of the model, and if the model just offered the prediction alone versus if it offered the prediction along with an explanation. That is the kind of the split that we want to make. Of course, controlling for randomness and biases, we want to be able to show that users are more likely to take some action on a model prediction if it came along with that explanation. This is one way we can validate that the explanation was key to actually making the user take further action based on the model. It's also sort of a measure of trust because it means that the user trusted the model prediction more with the explanation and trusted it enough to take an action.

Q. Explainability can be seen as a tool we show to developers that use these models, but we can also use explainability tools to debug models so that we understand how end users may see them. How would you think these approaches for explainability differ, or is there one tool that can rule them all?

EA: I don't think that end users want to be exposed to the details of prompt generation. I think they want you to automate that as much as you possibly can. Then you run into the following problem: if they don't get what they want, if you take too much of the prompt generation away from them, how do they push the model back to giving them what they want? Especially from the Copilot perspective, you don't want the users to really have to think about how they're generating their prompt. You want them to just write code the way they're going to write code, and then we magically give them the completion they want. We're not there yet, so you have to give them some control to do that. The current way of doing this is not very discoverable and depends a lot on experience. I don't know how long it would take a user to figure out that's how they're supposed to do it. There was an interesting article recently from Microsoft Research, where they had around 20 subjects complete a task using Copilot, and they recorded their actions.² They then had them retrospectively label what they were doing at each time interval during the session. They found that 11.6% of the time was spent in prompt crafting. They were already doing it.

VM: I think explainability can help in both cases, but there's a slightly different notion of explainability in my opinion. In the first case, we are trying to show explanations per prediction to users, so that is essentially going to quantify the consistency between the model's prediction and the explanation that it offers. It's going to be per example or per prediction. For debugging, I think we need maybe a slightly different notion of explainability, which is explaining what the model has actually learned. As model designers, that is

ABOUT THE AUTHORS



JÜRGEN CITO is an assistant professor at TU Wien, 1040 Vienna, Austria. His research interests include machine learning for software engineering applications. Cito received his Ph.D. in computer science from the University of Zurich, Switzerland and was a postdoctoral research scholar at the Massachusetts Institute of Technology. Contact him at juergen.cito@tuwien.ac.at.



SATISH CHANDRA is a research scientist at Google, CA 94043 USA. His research interests include programming languages and software engineering, including program analysis, type systems, software synthesis, bug finding and repair, software testing and test automation, and, most recently, applications of machine learning to developer tools. Chandra received his Ph.D. in computer science from the University of Wisconsin-Madison. He is an Association for Computing Machinery Distinguished Scientist. Contact him at schandra@acm.org.



CHAKKRIT TANTITHAMTHAVORN is a senior lecturer of software engineering and a 2020 Australian Research Council Discovery Early Career Research Award Fellow in the Faculty of Information Technology, Monash University, Clayton, Victoria 3800, Australia. His current focus is on pioneering an emerging research area of explainable artificial intelligence (AI) for software engineering and inventing many AI/machine learning/deep learning/natural language processing-based technologies to improve developers' productivity and make software systems more reliable and secure, while being explainable to practitioners. Contact him at chakkrit@monash.edu.



HADI HEMMATI is an associate professor in the Department of Electrical Engineering and Computer Science, York University, Toronto, ON M3J1P3, Canada. He is also an adjunct associate professor at Calgary, AB, Canada. His main research interests include automated software engineering (with a focus on software testing, debugging, and repair) and trustworthy artificial intelligence (with a focus on robustness and explainability). His research has a strong focus on empirically investigating software/machine learning engineering practices in large-scale industrial systems. He is a Senior Member of IEEE. Contact him at hemmati@yorku.ca.

essentially what we want to know: not necessarily individual explanations per prediction, but something that actually looks at the model as a whole. Maybe

it's an aggregation of individual predictions. For these really large models, such as ChatGPT, the model designers probably ask themselves: What has


this model actually learned? What are the blind spots for this model? Where can it use offensive content or content that we don't want to show to users? Where is it more accurate; where is it less accurate? As model designers, we want to understand the model as a whole and ensure that we have all of those things covered, so we are not, for instance, generating vulnerable code.

Q. Where do you think the future of these AI systems for software engineering is going, and how do you think explainability can support that?

EA: We're on the cusp of these models being integrated into all aspects of the development process—activities beyond just writing code. To make that work and to make sure the outputs

are accurate, you're going to need to be able to understand the “why.” Why did you get the generation that you didn't want? Why did this bit of the generation come out this way? I'm very focused on the side of the person making these tools, but right now the process is in its infancy. We're touching on a wide set of problems, and our principles are evolving. You can see this in the various prompts. There are a lot of tricks. A lot of it is just experience based. But also, who knows if those tricks will hold up with newer models? I see explainability as critical in helping to build these tools and, of course, in enabling the users to actually use them and build trust in them.

VM: I think we need to identify the systems where explanations are more

critical compared to others and build the explanation into the model itself. I think that the output that comes out of the model itself is the best way to get explanations in front of users and start making real impact. 

References

1. V. Murali, E. Yao, U. Mathur and S. Chandra, “Scalable statistical root cause analysis on app telemetry,” in *Proc. IEEE/ACM 43rd Int. Conf. Softw. Eng.: Software Engineering in Practice (ICSE-SEIP)*, Madrid, Spain, 2021, pp. 288-297, doi: 10.1109/ICSE-SEIP52600.2021.00038.
2. H. Mozannar, G. Bansal, A. Fournay, and E. Horvitz, “Reading between the lines: Modeling user behavior and costs in AI-assisted programming,” 2022, *arXiv-2210*.

Computing in Science & Engineering

The computational and data-centric problems faced by scientists and engineers transcend disciplines. There is a need to share knowledge of algorithms, software, and architectures, and to transmit lessons-learned to a broad scientific audience. *Computing in Science & Engineering (CiSE)* is a cross-disciplinary, international publication that meets this need by presenting contributions of high interest and educational value from a variety of fields, including physics, biology, chemistry, and astronomy. *CiSE* emphasizes innovative applications in cutting-edge techniques. *CiSE* publishes peer-reviewed research articles, as well as departments spanning news and analyses, topical reviews, tutorials, case studies, and more.

Read *CiSE* today! www.computer.org/cise



IEEE
COMPUTER
SOCIETY



IEEE

Digital Object Identifier 10.1109/MS.2023.3262314

