Editor: **Philippe Kruchten**
University of British Columbia
pbk@ece.ubc.ca

The Tragedy of Defect Prediction, Prince of Empirical Software Engineering Research

Michele Lanza, Andrea Mocci, and Luca Ponzanelli

IF MEASURED BY the number of published papers, defect prediction has become an important research field over the past decade, with many researchers continuously proposing novel approaches to predict defects in software systems. However, most of these approaches have had a noticeable lack of impact on industrial practice. We believe that the impact isn't there because something is intrinsically wrong in how defect prediction approaches are evaluated.

Act I: Empirical Quicksand

> *Something is rotten in the state of Denmark* —*Shakespeare*, The Tragedy of Hamlet, Prince of Denmark, *act I, scene 4*

In this case, Denmark isn't a Scandinavian country—it's the research field called defect prediction. We reflect on what we consider its intrinsic conceptual flaw. This flaw pertains not only to defect prediction but also to other research fields with which defect prediction shares a peculiar commonality. As you'll see, this commonality pertains to the infamous evaluation that has become a necessary evil of modern software engineering research.

We're heading into dangerous territory here, so we'd better take this one step at a time. First, what is defect prediction?

Defect prediction deals with the creation and empirical evaluation of approaches to know or estimate in advance where defects will appear in a system. The earliest approaches, devised in the 1980s, used simple regression models based on software metrics.[1] Since then, the field has seen the invention of a staggering number of novel and more refined approaches. This was especially the case during the rise of the research field of mining software repositories, which in turn gave birth to what some people called "empirical software engineering 2.0."

Our goal isn't to criticize empirical software engineering as a whole, which has many reasons to exist. However, defect prediction is an archetypal example of empirical software engineering research where in the middle of the many trees that need to be felled, the research community has lost sight of the forest. This is especially true concerning the evaluation of novel approaches, which seems to have surpassed the actual technical core of any approach in terms of importance and acceptance.

0740-7459/16/$33.00 © 2016 IEEE

If you survey the many publications on defect prediction, it's hard not to notice how important the evaluation is, filled with precision and recall percentages, *p*-values, and other success metrics. What's wrong with that, you might ask? Indeed, we don't argue against evaluating such approaches; quite the contrary. But we do maintain that the de facto way of evaluating them is intrinsically flawed. A tiny spoiler: "If my calculations are correct, when this baby hits 88 miles per hour, you're gonna see some serious ****."

Act II: Small-Scale Utopia

Assume for a second that the world is perfect (except for the existence of software bugs). In such a world, a bug consists of a specific software change, which leads to a defect, which someone then reports through a bug report. In reaction to the reported bug, a developer provides a fix, effectively closing the process.

This process is not only a simplification but also an idealization. The actual process is much more complex and probably never assumes the form we describe here. For example, the typical bug report life cycle allows loops to happen when bugs get reopened.

However, several defect prediction approaches rely on this simplified process, and for the sake of our thought experiment, we assume this process too.

The pieces of that assumption that are important for validating defect prediction approaches are primarily the bug report and secondarily the change and fix. This is because the report is the actual artifact that can be mined with ease from issue tracker repositories and because the change and the fix can be recovered—with some effort—from the versioning system.

(An intriguing conceptual question pertains to the defect itself. In reality, defect prediction approaches are validated not on the actual defects but on the creation of bug reports. In short, from the perspective of defect prediction evaluation, if there's no bug report, there's no bug.)

We now take this process and turn it into the (admittedly abstract) concept of "a bug," ignoring its internal details. We do this to widen the context in the next act.

Act III: Large-Scale Dystopia

When researchers evaluate defect prediction approaches, they use as ground truth the "past"—all the bugs reported during a reference time period. The present is placed into the past (which thus becomes a new present), and the predictor is run into the future, which in reality is the past as seen from the actual present.

Defect prediction approaches now try, irrespective of their underlying technicalities, to predict where (for example, in which class or module) defects will appear. To do so, they use the fix that was a response to the bug report and establish which parts of the system changed during the fix. If those parts match the predicted part of the system, the predictor worked correctly.

(Someone could point out that establishing what changed during a fix, and whether all those changes were a "response" to the bug report, isn't an exact science and is often based on [imprecise] heuristics.)

The evaluation process allows for an easy way to measure defect prediction approaches' performance—for example, using such common metrics as precision, recall, and F-measures. On the basis of those metrics, the approaches are then compared to each other.

(It's still sad but true that few usable benchmark datasets exist, so any comparison between different approaches on diverse datasets is an apples-and-oranges comparison with little validity.)

So far, so good. So what? The problem is that software is developed by humans, and more important, it evolves. Any decision in the system—for example, a bug fix or any other change—impacts, directly or indirectly, future decisions over the development time. And time is the keyword for our next act.

Act IV: 88 MPH

If a predictor were put into production as an in vivo tool, it would produce recommendations. Developers would see these recommendations, which would influence and (hopefully) significantly affect what they do from that moment on.

In other words, if a defect predictor predicted a bug in a particular area of the system, a developer would look at that area. However, this simple reaction has an influence, and potentially a cascading effect, on anything that follows in time.

Of course, you might ask, aren't we pointing out the obvious here? Indeed: a useful recommender must have some impact on a system's evolution. Well, here's the problem. As we mentioned before, defect prediction approaches are evaluated on the past history of a system's bugs, where that history is treated as the future. In essence, the predictors are being evaluated in a way equivalent to the situation in which they act as if developers will completely ignore them. But, if the point of an approach (that is, research) is to have some impact on a system (that is, the real world), doesn't this contradict that goal?

The evaluation of defect prediction approaches using a system's bug history is intrinsically flawed.

If a predictor correctly predicted a bug, that recommendation would impact any subsequent bug and might produce unexpected consequences. Two possibilities are that subsequent bugs either don't appear where they previously appeared or don't appear at all. In essence (apologies for the upcoming high-flying wording), a real prediction perturbs the space–time continuum.

Let's fly at a lower altitude, so to speak. Bugs are causally connected because software is produced by humans, and if they're doing something, they're not doing something else. Short of supporting the theory of parallel universes, the main message is this:

> *The evaluation of defect prediction approaches using a system's bug history is intrinsically flawed.*

Tying back to the spoiler: defect prediction approaches are evaluated using the fading-picture metaphor from the movie *Back to the Future*. Although the movie is very nice, its time travel logic is full of evident paradoxes.

Act V: The Angel's Advocate

Because the devil's advocate seems to be a coauthor of this article, we summon the help of the angel's advocate, who dislikes what we claim.

"Not all bugs are causally connected: your base assumption is wrong. If two bugs reside in very distant parts of a system and aren't structurally related, they have no causal connection whatsoever."

We concede that if two bugs appear at very close points in time in very distant parts of the system, they might have no causal connection. However, given enough time distance, even bugs that are far from each other and aren't structurally related are causally connected because software is developed by humans who follow a process.

"If a defect prediction approach uses only the code's structural properties, you can ignore or factor out the recommendations' impact on the system's evolution."

Indeed, if the recommendations are based on only an entity's structural properties, the impact of the system's evolution on the code met-

rics can be factored out. In fact, the more the predictor recommends the entities that have been really fixed, the more the predicted evolution resembles the one that was observed in the system. However, current approaches don't rely only on purely structural metrics. If we consider Marco D'Ambros and his colleagues' study,[2] process metrics actually exhibit the best performance and lowest variability, outperforming other metrics based on source code. This is because process metrics are intrinsically evolutionary and are thus strongly influenced by any change of the development process itself.

"You're wrong; a simple *n*-fold cross validation on the bug dataset is enough to do away with all this time-traveling nonsense."

Such cross validation can't take into account the defect predictor's potential impact on the system. In other words, recombining the training and testing datasets with *n*-fold cross validation won't help you reconstruct the defect predictor's potential impact on the system by simulating the changes that such a recommender would make. Without an in vivo adoption, you simply can't measure the predictor's effect.

"Wait; if you did defect prediction research yourself, aren't you biting the hand that fed you?"

Yes, we are. We don't deny that, nor do we regret it. Some of our most impactful (that is, cited) papers are in that area. Back in those days, we believed that was the way to do things. This insight of ours came only recently, but hey, better late than never.

Epilog

Although this article's tone isn't exactly serious, the point we're making is a serious and honest criticism of how defect prediction approaches have been evaluated all these years.

We reiterate that we're not criticizing defect prediction approaches per se. In fact, many approaches are based on sound, meaningful assumptions. These are some well-known conjectures formulated by researchers: bugs might appear in parts of the system where bugs have already occurred,[3] in parts that change frequently,[4] or where complex changes have been performed.[5]

These conjectures are clearly reasonable and well founded, and it's interesting to investigate them to prove or disprove them. In the end, the overall goal has been and remains to advance the state of the art and the software engineering discipline, in which the engineering is to be considered as a set of proven best practices.

The problem isn't the approaches. The problem lies in how the approaches are evaluated and how they're being compared to each other.

The unpleasant truth is that a field such as defect prediction makes sense only if it's used in vivo.

In other words, researchers should seriously consider putting their predictors out into the real world and having them used by developers who work on a live code base. Of course, this comes at a high cost. But then again, consider that if a predictor manages to correctly predict one single bug, this will have a real, concrete impact. That's more than can be said about any approach relying on in vitro validation, no matter how extensive.

References

1. V.Y. Shen et al., "Identifying Error-Prone Software: An Empirical Study," *IEEE Trans. Software Eng.*, vol. 11, no. 4, 1985, pp. 317–324.
2. M. D'Ambros, M. Lanza, and R. Robbes, "Evaluating Defect Prediction Approaches: A Benchmark and an Extensive Comparison," *Empirical Software Eng.*, vol. 17, nos. 4–5, 2012, pp. 531–577.
3. S. Kim et al., "Predicting Faults from Cached History," *Proc. 29th Int'l Conf. Software Eng.* (ICSE 07), 2007, pp. 489–498.
4. N. Nagappan and T. Ball, "Use of Relative Code Churn Measures to Predict System Defect Density," *Proc. 27th Int'l Conf. Software Eng.* (ICSE 05), 2005, pp. 284–292.
5. A.E. Hassan, "Predicting Faults Using the Complexity of Code Changes," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 78–88.

MICHELE LANZA is a full professor and the head of the REVEAL (Reverse Engineering, Visualization, Evolution Analysis Lab) research group at the Università della Svizzera italiana. Contact him at michele.lanza@usi.ch.

ANDREA MOCCI is a postdoctoral researcher in the REVEAL research group at the Università della Svizzera italiana. Contact him at andrea.mocci@usi.ch.

LUCA PONZANELLI is a PhD student in informatics and a member of the REVEAL research group at the Università della Svizzera italiana. Contact him at luca.ponzanelli@usi.ch.