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

Michele Lanza, Andrea Mocci, Luca Ponzanelli
REVEAL @ Università della Svizzera Italiana (USI)

ABSTRACT

If measured in terms of number of published papers, defect prediction has become in the past decade an important research field, with many researchers continuously proposing novel approaches to predict defects in software systems. However, there is also a noticeable lack of impact on industrial practice of most of those approaches. This begs the question whether there is something amiss.

We present a series of reflections on how defect prediction approaches are evaluated, stating that there is something intrinsically wrong in how it is often conducted. This in turn might explain to a certain extent why defect prediction still lacks real-world impact. Our goal is not to criticize defect prediction *per se*, but how defect prediction approaches are evaluated.

CCS Concepts

•**Software and its engineering** → *Defect analysis;*

Keywords

Defect prediction, Mining software repositories

Act I: Prologue

> *Something is rotten in the state of Denmark.*
>
> William Shakespeare — *The Tragedy of Hamlet, Prince of Denmark.* Act I, Scene 4

Denmark is, in this case, not a Scandinavian country: It is the research field known as *defect prediction.* Throughout this paper we will discuss its intrinsic conceptual flaw, which however does not only pertain to defect prediction as such, but also bleeds into other research fields with which defect prediction shares a peculiar commonality. The commonality pertains, as we will see, to the infamous evaluation which has become a necessary evil of modern software engineering research. More about this later.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

FSE '16, Nov 13-19, 2016, Seattle, WA, USA
© 2016 ACM. ISBN 978-1-4503-3900-1/16/05...$15.00
DOI: http://dx.doi.org/10.1145/2884781.2884824

As we are treading dangerous territories here, we better take one step after another: What is defect prediction?

Defect prediction deals with the creation and empirical evaluation of approaches to know/estimate in advance where in a system defects will appear. The earliest approaches, devised in the 1980s, used simple regression models based on various software metrics [9]. The field has since then seen a staggering amount of novel and more refined approaches being invented, especially during the rise of *Mining Software Repositories (MSR)* as a research field. According to Andreas Zeller in his MSR keynote in 2007, MSR as a research community laid the groundwork for the birth of what he named "empirical software engineering 2.0". Indeed, if this were a brawl in a pub, we are picking on the largest guy in the room to have a beef with.

Our goal is not to criticise empirical software engineering as a whole, which does have many reasons to exist. However, defect prediction is a 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 when it comes to the evaluation of novel approaches, which seems to have surpassed the actual technical core of any approach in terms of importance and acceptance-inducing factor.

If one surveys the many publications on defect prediction, it is hard not to notice how important the evaluation is, filled with precision and recall percentages, p-values, and other success metrics. What is wrong with that, one might ask? Indeed, we do not argue against evaluating such approaches, quite the contrary. But, we do maintain that the *de facto* way of performing evaluations is intrinsically flawed, as we will see in the remainder of this paper. A tiny spoiler[1]: *"If my calculations are correct, when this baby hits 88 miles per hour... you're gonna see some serious shit."*

User guide. Before starting, let us explain the gist of the present paper. The call for papers for the Visions & Reflections track of FSE 2016 states "[...] a forum for innovative, thought-provoking research in software engineering [...]" and "[...] The writing style can even be narrative to the extent where this supports the motivation [...]". We have therefore taken the liberty to mimic[2] a theatre piece. Moreover, we will inject interludes in italics which are to be treated as side remarks for the reader. Lastly, the writing style is rather informal. Again, this is not accidental. In essence, a grain of humour certainly helps in the reading of the following pages.

[1]Movie citation. Apologies for the coarse language.
[2]In an admittedly clumsy way.

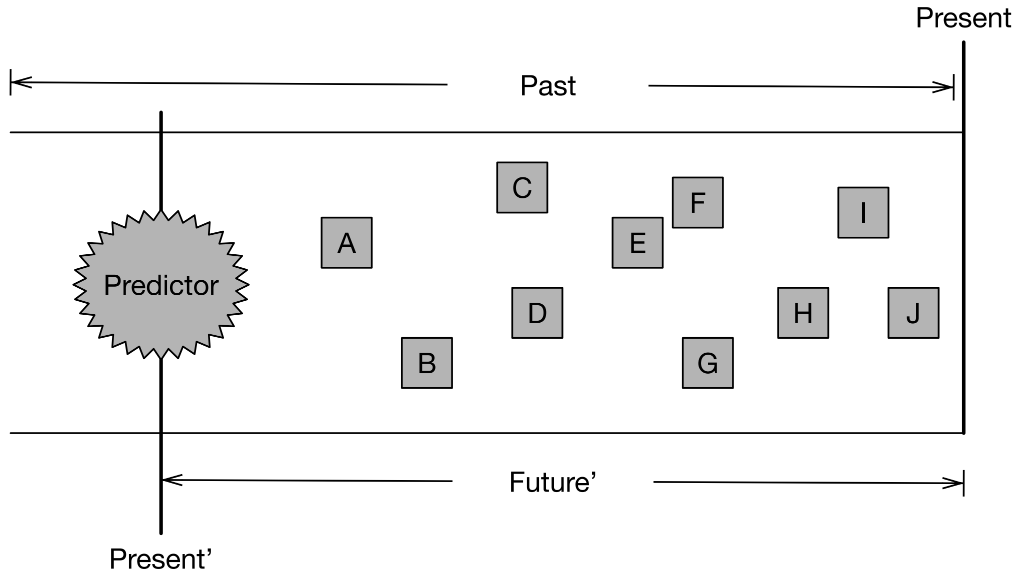

Figure 1: The essence of defect prediction evaluation as a diagram.

Act II: Small-scale Utopia

We will now describe a thought experiment: Let us assume for a second that the world is perfect.

In such a perfect world, a bug B in a software system consists of: a specific software change C, which leads to a software defect D, which is then reported by someone through a bug report R. As a reaction to the reported bug, a developer then provides a fix F, 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 intrinsically 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 the same process as well.

The pieces of that assumption which are important for the validation of defect prediction approaches are primarily R and secondarily C and F, because R is the actual artifact which can be mined with ease from issue tracker repositories, and C and F because they can be recovered — with some effort — from the versioning system.

> *An intriguing conceptual question is the one pertaining to the defect itself, D. In reality, defect prediction approaches are not validated on the actual defects, but on the creation of bug reports. In short, from the perspective of defect prediction evaluation, if there is no bug report, there is simply no bug. However, for the present thought experiment we ignore this for now.*

We now take the above process and abstract it into the (admittedly abstract) concept of "a bug" B, ignoring its internal details. We need B as an abstraction to widen the context in the next act.

Act III: Large-scale Dystopia

When defect prediction approaches are evaluated, researchers use as ground truth the "past", *i.e.,* all bugs reported during a reference time period. The present is placed into the past (becoming present'), and the predictor is run onto 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 (*e.g.,* in which class or module) defects will appear. To do so they use the fix F which was a response to the report R, 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. Figure 1 depicts in a simplified way this evaluation process.

> *One could point out that establishing what changed during a fix, and whether all those changes were a "response" to R is not an exact science, and often based on (imprecise) heuristics.*

The mentioned evaluation process allows for an easy way to measure the performance of defect prediction approaches, for example using precision, recall, and f-measures, to mention some often used metrics. It is then on those success metrics that approaches are compared against one another.

> *It is still sad but true that there are very few (re)usable benchmark datasets, and therefore 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 the following: Software is developed by humans, and more importantly, it *evolves.* Any decision in the system, *e.g.,* 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 Miles per Hour

If a predictor were to actually be put into production as an *in vivo* tool, it would produce recommendations. These recommendations would be seen by developers, and this would influence, and hopefully have a significant impact, on what they will do from that moment on.

In other words, if a defect predictor were to predict the presence of a bug in a particular area of the system, a developer would go and have a look at that part. This simple reaction does however have an influencing, and potentially cascading effect on anything that follows in time.

Of course, one might say, aren't we pointing out the obvious here? Indeed: A useful recommender must have some impact in the evolution of a system. Well, the problem is the following: Defect prediction approaches are evaluated on the past history of bugs of a system, where that history is treated as the future. In essence, the way that defect predictors are being evaluated is equivalent to the situation where the very same *defect predictors act as if they would be completely ignored* by developers. But, if the point of an approach (*i.e.,* research) is to have any sort of impact on a system (*i.e.,* the real world), does this not contradict the root mission of research itself?

To summarize again with the help of Figure 1: If a predictor were to correctly predict the future presence of bug A, that recommendation would impact any subsequent bug and as a matter of act might produce unexpected consequences. One possibility is that bugs B to J either do not appear when they actually appeared, or they do not appear at all. In essence, apologizing ahead for the high-flying wording, a real prediction *perturbs the time-space continuum.*

Flying at a lower altitude, bugs are causally connected because software is produced by humans, and if they are doing something they are not doing something else. Short of supporting the theory of parallel universes, the main message of this set of reflections is that *the evaluation of defect prediction approaches using the past bug history of a system 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" (see Figure 2). Although a very nice movie, its time travel logic is full of evident paradoxes.

Figure 2: The essence of defect prediction evaluation as a picture.

Act V: The Angel's Advocate

As the devil's advocate seems to be among the co-authors of this paper, we summon the help of the angel's advocate, who dislikes what we just wrote.

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

Indeed, we must concede that if two bugs were to appear at very close points in time in very distant parts of the system, indeed they might not have any causal connection. However, given enough time distance, even bugs who are far away from each other and are not structurally related, are causally connected, because software is developed by humans who follow a process.

If a defect prediction approach uses only structural properties of the code, the impact of the recommendations on the system's evolution can be ignored or factored out.

Indeed, if the recommendations are based only on structural properties of an entity, the impact of system's evolution on the code metrics can be considered relatively similar to the one observed. In fact, the more the predictor recommends the entities that have been really fixed, the more the process resembles the one that has been actually observed in the system. However, current approaches do not only rely on purely structural metrics. If we consider the study performed by D'Ambros *et al.* [2], process metrics [7] exhibit the best performance and lowest variability, outperforming other metrics sets based on source code [1]. Process metrics are evolutionary metrics, and they would be strongly influenced by any change of the process, like the *in vivo* usage of a defect prediction recommender.

You are wrong, a simple n-fold cross validation on the bug dataset is enough to make do with all this time-traveling nonsense.

The benefit of n-fold cross validation cannot take into account the potential impact of the defect predictor on changing the system. In other words, it is not by recombining the training and testing datasets with n-fold cross validation that one can reconstruct the potential impact of the defect predictor in the system, simulating the changes that such a recommender would do. Without an in vivo adoption, measuring the effect of the predictor is indeed impossible.

Wait, didn't you also do defect prediction research?

Yes, we did, and not only don't we deny that, nor do we regret that. Some of our most impactful (as in: cited) papers are in that area. Our bug dataset[3] has been extensively used by other researchers. Back in those days we believed that was the right thing to do. However, this little reflection and insight of ours came only recently.

Care to explain this *in vivo*? Performed or taking place in a living organism. The opposite of *in vitro.*

[3] See http://bug.inf.usi.ch

Act VI: Aftermath

Although the tone of this paper is not exactly on the serious side, we do want to emphasize that the point we are making is a serious and honest criticism towards how defect prediction approaches have been evaluated all these years.

We want to reiterate that we do not criticize defect prediction approaches *per se*. In fact, many approaches are based on sound and meaningful assumptions. Let us look at some well-known conjectures formulated by researchers:

- New bugs might appear in parts of the system where there have already been bugs [6].
- Bugs might appear in the part of the system where complex changes have been performed [5].
- Bugs might appear in parts of the system which are badly designed [1, 4, 10].
- Bugs might appear in part of the system which change frequently [8].

These conjectures are clearly reasonable and well-founded, and it is indeed interesting to investigate them with the goal of proving/disproving them. In the end the overall goal is and remains to advance the state of the art and the software engineering discipline as such, where the engineering is to be considered as a set of proven best practices.

The problem are not the approaches, and therefore this present paper is by no means comparable to Fenton and Ohlsson's article "A critique of Software Defect Prediction Models" [3]. We believe instead the problem lies in how the approaches are evaluated and how they are being compared against each other.

However, pointing out problems is too easy, we want to conclude the paper by proposing a solution.

We start with the the lyrics of Megadeth's song *Ashes in Your Mouth*:

> *Now we've rewritten history*
> *The one thing we've found out*
> *Sweet taste of vindication*
> *It turns to ashes in your mouth*
>
> *Where do we go from here?*
> *And should we really care?*
> *The answer to your question is*
> *"Welcome to tomorrow!"*
>
> Megadeth — *Ashes in Your Mouth.* Symphony of Destruction, 1992.

The truth, albeit unpleasant, is that a field like **defect prediction only makes sense if it is used *in vivo***.

In other words, researchers active in this area should seriously consider putting their predictors out into the real world, and have them being used by developers who work on a live code base. Of course this comes at a high cost, but then again one should consider that even if a predictor manages to correctly predict one single bug, this would have a real and concrete impact, which is more than that can be said about any approach which relies on an *in vitro* validation, no matter how extensive.

We end the paper with a series of suggestions:

The central point raised by our paper in fact also applies to other research fields beyond defect prediction, where change prediction is an obvious example. We conjecture it applies in general to any approach where the past is treated as the future for the sake of evaluating any approach which deals with evolving software systems.

In case the reader is not aware of the movie "Back to the Future", we strongly suggest to watch it. Not only does that make it easier to understand some allusions in the paper, but the movie is a superb way to spend two hours of your precious time.

Following up on the previous point, please abstain from watching "Back to the Future II" and "Back to the Future III". You have been warned. It will be literally a waste of your... time.

1. REFERENCES

[1] V. R. Basili, L. C. Briand, and W. L. Melo. A validation of object-oriented design metrics as quality indicators. *IEEE Transactions on Software Engineering*, 22(10):751–761, Oct 1996.

[2] M. D'Ambros, M. Lanza, and R. Robbes. Evaluating defect prediction approaches: A benchmark and an extensive comparison. *Empirical Software Engineering*, 17(4-5):531–577, 2012.

[3] N. E. Fenton and N. Ohlsson. Quantitative analysis of faults and failures in a complex software system. *IEEE Transactions on Software Engineering*, 26(8):797–814, Aug. 2000.

[4] T. Gyimothy, R. Ferenc, and I. Siket. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, 31(10):897–910, Oct 2005.

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

[6] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller. Predicting faults from cached history. In *Proceedings of ICSE 2007 (29th International Conference on Software Engineering)*, pages 489–498. IEEE Computer Society, 2007.

[7] R. Moser, W. Pedrycz, and G. Succi. A comparative analysis of the efficiency of change metrics and static code attributes for defect prediction. In *Proceedings of ICSE 2008 (30th International Conference on Software Engineering)*, pages 181–190. ACM, 2008.

[8] N. Nagappan and T. Ball. Use of relative code churn measures to predict system defect density. In *Proceedings of ICSE 2005 (27th International Conference on Software Engineering)*, pages 284–292. ACM, 2005.

[9] V. Y. Shen, T.-J. Yu, S. M. Thebaut, and L. R. Paulsen. Identifying error-prone software an empirical study. *IEEE Transactions on Software Engineering*, 11(4):317–324, Apr. 1985.

[10] R. Subramanyam and M. S. Krishnan. Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, 29(4):297–310, Apr. 2003.