Software Analytics: What's Next?

Tim Menzies, North Carolina State University

Thomas Zimmermann, Microsoft Research

// Developers sometimes develop their own ideas about "good" and "bad" software, on the basis of just a few projects. Using software analytics, we can correct those misconceptions. //

FPO

SOFTWARE DEVELOPMENT IS a complex process. Human developers may not always understand all the factors that influence their projects. Software analytics is an excellent choice for discovering, verifying, and monitoring the factors that affect software development.

Software analytics distills large amounts of low-value data into small chunks of very high-value information.[1] Those chunks can reveal what factors matter the most for software projects. For example, Figure 1 lists some of the more prominent recent insights learned in this way.

Software analytics lets us "trust, but verify" human intuitions. If someone claims that "this or that" is important for a successful software project, analytics lets us treat that claim as something to be verified (rather than a sacred law that cannot be questioned). Also, once the claim is verified, analytics can act as a monitor to continually check whether "this or that" is now overcome by subsequent developments.

Anthropologists studying software projects warn that developers usually develop their personal ideas about good and bad software on the basis of just a few past projects.[2] All too often, these ideas are assumed to apply to all projects, rather than just the few seen lately. This can lead to too much reuse of too many old, and now outdated, ideas. A recent study of 564 software developers found that

> *a) programmers do indeed have very strong beliefs on certain topics b) their beliefs are primarily formed based on personal experience, rather than on findings in empirical research and c) beliefs can vary with each project, but do not necessarily correspond with actual evidence in that project.*[3]

The good news is that, using software analytics, we can correct those misconceptions (for examples, see the sidebar "Why We Need Software Analytics"). For example, after a project is examined, certain coding styles could be seen as more bug prone (and to be avoided).

There are many ways to distill that data, including quantitative methods (which often use data-mining algorithms) and qualitative methods (which use extensive human feedback to guide the data exploration). Previously, we have offered extensive surveys of those methods (see Figure 2).[4–7] The goal of this article is to update those surveys and offer some notes on current and future directions in software analytics.

Some History

As soon as people started programming, it became apparent that programming was an inherently buggy process. Maurice Wilkes, speaking of his programming experiences in the early 1950s, recalled the following:

> *It was on one of my journeys between the EDSAC room and the punching equipment that "hesitating at the angles of stairs" the realization came over me with full force that a good part of the*

 0740-7459/18/$33.00 © 2018 IEEE

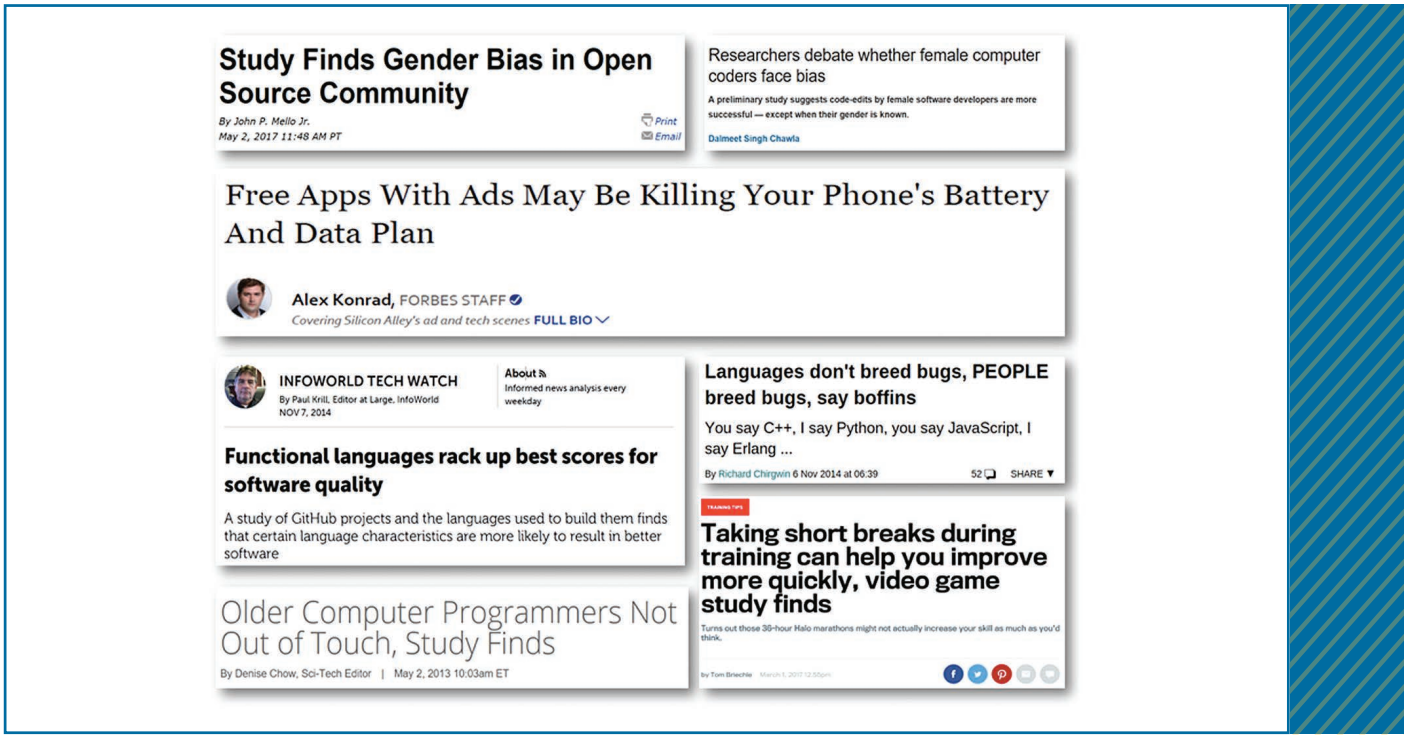

FIGURE 1. Surprising software analytics findings in the press, from *Linux Insider*, *Nature*, *Forbes*, *InfoWorld*, *The Register*, *Live Science*, and *Men's Fitness*.

> *remainder of my life was going to be spent in finding errors in my own programs.*[8]

It took several decades to gather the experience required to quantify any kind of size–defect relationship. In 1971, Fumio Akiyama described the first known "size" law, saying that the number of defects D was a function of the number of LOC.[9] In 1976, Thomas McCabe argued that the number of LOC was less important than the complexity of that code.[10] He argued that code is more likely to be defective when his *cyclomatic complexity* measure is over 10.

Not only is programming an inherently buggy process, it's also inherently difficult. On the basis of data from dozens of projects, Barry Boehm proposed in 1981 an

WHY WE NEED SOFTWARE ANALYTICS

Prior to the 21st century, researchers often had access to data from only one or two projects. This meant theories of software development were built from limited data. But in the data-rich 21st century, researchers have access to all the data they need to test the truisms of the past. And what they've found is most surprising:

- In stark contrast to the assumptions of much prior research, pre- and post-release failures are not connected.[25]
- Static code analyzers perform no better than simple statistical predictors.[26]
- The language construct GOTO, as used in contemporary practice, is rarely considered harmful.[27]
- Strongly typed languages are not associated with successful projects.[28]
- Developer beliefs are rarely backed by any empirical evidence.[3]
- Test-driven development is not any better than "test last."[29]
- Delayed issues are not exponentially more expensive to fix.[30]
- Most "bad smells" should not be fixed.[31,32]

FIGURE 2. Researchers have collaborated extensively to record the state of the art in software analytics.

estimator for development effort (that was exponential with program size) using a set of *effort multipliers* M_i inferred from the current project:[11]

$$effort = a * KLOC^b * \Pi M_i,$$

where $2.4 \leq a \leq 3$ and $1.05 \leq b \leq 1.2$. While the results of Akiyama, McCabe, and Boehm were useful in their home domains, it turns out that those results required extensive modification to work on other projects. One useful feature of the current generation of data-mining algorithms is that it is now relatively fast and simple to learn, for example, defect or effort predictors for other projects, using whatever attributes they have available.

Current Status

Thousands of recent research papers all make the same conclusion: data from software projects contains useful information that can be found by data-mining algorithms. We now know that with modest amounts of data collection, it is possible to

- build powerful recommender systems for software navigation or bug triage, or
- make reasonably accurate predictions about software development effort and defects.

Those predictions can rival those made by much more complex methods (such as static-code-analysis tools, which can be hard to maintain).

Various studies have shown the commercial merits of this approach. For example, for one set of projects, software analytics could predict 87 percent of code defects, decrease inspection effort by 72 percent, and hence reduce post-release defects by 44 percent.[12] Better yet, when multiple models are combined using ensemble learning, very good estimates can be generated (with very low variance in their predictions), and these predictors can be incrementally updated to handle changing conditions.[13] Also, when there is too much data for a manual analysis, it is possible to automatically analyze data from hundreds to thousands of projects, using software analytics.[14]

Recent research explains why software analytics has been so successful: artifacts from software projects have an inherent simplicity. As Abram Hindle and his colleagues explained,

> *Programming languages, in theory, are complex, flexible and powerful, but the programs that real people actually write are mostly simple and rather repetitive, and thus they have usefully predictable statistical properties*

that can be captured in statistical language models and leveraged for software engineering tasks.[15]

For example, here are just some of the many applications in this growing area of research:

- Patterns in the tokens of software can recognize code that is unexpected and bug prone.
- Sentiment analysis tools can gauge the mood of the developers, just by reading their issue comments.[16]
- Clustering tools can explore complex spaces such as Stack Overflow to automatically detect related questions.[17]

But What's Next?

When we reflect over the past decade, several new trends stand out. For example, consider the rise of the data scientist in industry. Many organizations now pay handsomely to hire large teams of data scientists. For example, at the time of this writing, there are more than 1,000 Microsoft employees exploring project data using software analytics. These teams are performing tasks that a decade ago would have been called cutting-edge research. But now we call that work standard operating procedure.

Several new application areas have emerged, such as green engineering. Fancy cellphones become hunks of dead plastic if they run out of power. Hence, taming power consumption is now a primary design concern. Software analytics is a tool for monitoring, and altering, power usage profiles.[18]

Another emerging area is social patterns in software engineering. Our society divides users into different groups. By adopting the perspectives of different social groupings, developers can build better software.[19] Models built from social factors (such as how often someone updates part of the code) can be more effective for predicting code quality than code factors (such as function size or the number of arguments). For example, when you're studying software built in multiple countries, a good predictor for bugs is the complexity of the organizational chart. (The fewest bugs are introduced when people working on the same functions report to the same manager, even if they are in different countries.[20])

Software analytics also studies the interactions of developers, using biometric sensors. Just as we mine software (and the social processes that develop them), so too can we mine data, collected at the millisecond level, from computer programmers. For example, using eye-tracking software or sensors for skin and brain activity, software analytics can determine what code is important to, or most difficult for, developers.[21]

When trading off different goals, the data-mining algorithms used in conventional software engineering often use hard-wired choices. These choices may be irrelevant or even antithetical to the concerns of the business users who are funding the analysis. For example, one way to maximize accuracy in unbalanced datasets (where, say, most of the examples are not defective) is to obsess on maximizing the true negative score. This can mean that, by other measures such as precision or recall, the learner fails. What are required are "learners" that guide the reasoning according to a user-specified goal. Such *search-based software engineering* tools can implement other useful tasks such as automatically tuning the control parameters of a data-mining algorithm[22] (which, for most developers, is a black art).

Other important trends show the maturity of software analytics. For decades now, there have been extensive discussions about the reproducibility of software research—much of it having the form, "wouldn't it be a good idea." We can report here that, at least for quantitative software analytics based on data-mining algorithms, such reproducibility is common practice. Many conferences in software engineering reward researchers with "badges" if they place their materials online (see tiny.cc/acmbadges). Accordingly, researchers place their scripts and data in GitHub. Some even register those repositories with tools such as Zenodo, in order to obtain unique and eternal digital object identifiers for their artifacts. For an excellent pragmatic discussion on organizing repositories in order to increase reproducibility, see "Good Enough Practices in Scientific Computing."[23]

Another challenge is the security and privacy issues raised by software analytics. Legislation being enacted around the world (e.g., the European General Data Protection Regulation, GDPR) means that vendors collecting data will face large fines unless they address privacy concerns. One way to privatize data is to obfuscate (mutate) it to reduce the odds that a malevolent agent can identify who or what generated that data. But this raises a problem: the more we mutate data, the harder it becomes to learn an effective model from it. Recent results suggest that sometimes this problem can be solved,[24] but much more work is required on how

ISSUES IN SOFTWARE ANALYTICS

Not everything that happens in software development is visible in a software repository,[33] which means that no data miner can learn a model of these subtler factors. Qualitative case studies and ethnographies can give comprehensive insights on particular cases, and surveys can offer industry insights beyond what can be measured from software repositories. Due to such methods' manual workload, they have issues with scaling up to cover many projects. Hence, we see a future where software is explored in alternating cycles of qualitative and quantitative methods (where each can offer surprises that prompt further work in the other[34]).

Another issue with software analytics is the problem of context.[35] That is, models learned for one project may not apply to another. Much progress has been made in recent years in learning context-specific models[36,37] or building operators to transfer data or models learned from one project to another.[38] But this is clearly an area that needs more work.

Yet another issue is how actionable are our conclusions. Software analytics aims to obtain actionable insights from software artifacts that help practitioners accomplish tasks related to software development and systems. For software vendors, managers, developers, and users, such comprehensible insights are the core deliverable of software analytics.[39] Robert Sawyer commented that actionable insight is the key driver for businesses to invest in data analytics initiatives.[40]

But not all the models generated via software analytics are actionable; i.e., they cannot be used to make effective changes to a project. For example, software analytics can deliver models that humans find hard to read, understand, or audit. Examples of those kinds of models include anything that uses complex or arcane mathematical internal forms, such as deep-learning neural networks, naive Bayes classifiers, or random forests. Hence, some researchers in this field have explored methods to reexpress complex models in terms of simpler ones.[41]

to share data without compromising confidentiality.

And every innovation also offers new opportunities. There is a flow-on effect from software analytics to other AI tasks outside of software engineering. Software analytics lets software engineers learn about AI techniques, all the while practicing on domains they understand (i.e., their own development practices). Once developers can apply data-mining algorithms to their data, they can build and ship innovative AI tools. While sometimes those tools solve software engineering problems (e.g., recommending what to change in source code), they can also be used on a wider range of problems. That is, we see software analytics as the training ground for the next generation of AI-literate software engineers working on applications such as image recognition, large-scale text mining, and autonomous cars or drones.

Finally, with every innovation come new challenges. In the rush to explore a promising new technology, we should not forget the hard-won lessons of other kinds of research in software engineering. This point is explored more in the sidebar "Issues in Software Analytics."

We end with this frequently asked question: "What is the most important technology newcomers should learn to make themselves better at data science (in general) and software analytics (in particular)?" To answer this question, we need a workable definition of "science," which we take to mean a community of people collecting, curating, and critiquing a set of ideas. In this community, everyone does each other the courtesy to try to prove this shared pool of ideas.

By this definition, most data science (and much software analytics) is not science. Many developers use software analytics tools to produce conclusions, and that's the end of the story. Those conclusions are not registered and monitored. There is nothing that checks whether old conclusions are now out of date (e.g., using anomaly detectors). There are no incremental revisions that seek minimal changes when updating old ideas.

If software analytics really wants to be called a science, then it needs to be more than just a way to make conclusions about the present. Any scientist will tell you that all ideas should be checked, rechecked, and incrementally revised. Data science methods such as software analytics should be a tool for assisting in complex discussions about ongoing issues.

Which is a long-winded way of saying that the technology we most need to better understand software analytics and data science is … science.

References

1. D. Zhang et al., "Software Analytics in Practice," *IEEE Software*, vol. 30, no. 5, 2013, pp. 30–37.
2. C. Passos et al., "Analyzing the Impact of Beliefs in Software Project Practices," *Proc. 2011 Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 11), 2011, pp. 444–452.
3. P. Devanbu, T. Zimmermann, and C. Bird, "Belief & Evidence in Empirical Software Engineering," *Proc. 38th Int'l Conf. Software Eng.* (ICSE 16), 2016, pp. 108–119.
4. C. Bird, T. Menzies, and T. Zimmermann, eds., *The Art and Science of Analyzing Software Data*, Morgan Kaufmann, 2015; goo.gl/ZeyBhb.
5. T. Menzies, L. Williams, and T. Zimmermann, eds., *Perspectives on Data Science for Software Engineering*, Morgan Kaufmann, 2016; goo.gl/ZeyBhb.
6. T. Menzies et al., *Sharing Data and Models in Software Engineering*, Morgan Kaufmann, 2014; goo.gl/ZeyBhb.
7. T. Menzies and T. Zimmermann, "Software Analytics: So What?," *IEEE Software*, vol. 30, no. 4, 2013, pp. 31–37.
8. M. Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985.
9. F. Akiyama, "An Example of Software System Debugging," *Information Processing*, vol. 71, 1971, pp. 353–359.
10. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, 1976, pp. 308–320.
11. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.
12. A.T. Misirli, A. Bener, and R. Kale, "AI-Based Defect Predictors: Applications and Benefits in a Case Study," *AI Magazine*, Summer 2011, pp. 57–68.
13. L.L. Minku and X. Yao, "Ensembles and Locality: Insight on Improving Software Effort Estimation," *Information and Software Technology*, vol. 55, no. 8, 2013, pp. 1512–1528.
14. R. Krisha et al., "What Is the Connection between Issues, Bugs, and Enhancements? (Lessons Learned from 800+ Software Projects)," presentation at 40th Int'l Conf. Software Eng. (ICSE 18), 2018; https://arxiv.org/abs/1710.08736.
15. A. Hindle et al., "On the Naturalness of Software," *Proc. 34th Int'l Conf. Software Eng.* (ICSE 12), 2012, pp. 837–847.
16. A. Murgia et al., "Do Developers Feel Emotions? An Exploratory Analysis of Emotions in Software Artifacts," *Proc. 11th Working Conf. Mining Software Repositories* (MSR 14), 2014, pp. 262–271.
17. B. Xu et al., "Predicting Semantically Linkable Knowledge in Developer Online Forums via Convolutional Neural Network," *Proc. 31st ACM/IEEE Int'l Conf. Automated Software Eng.* (ASE 16), 2016, pp. 51–62.
18. A. Hindle et al., "GreenMiner: A Hardware Based Mining Software Repositories Software Energy Consumption Framework," *Proc. 11th Working Conf. Mining Software Repositories* (MSR 14), 2014, pp. 12–21.
19. M. Burnett et al., "GenderMag: A Method for Evaluating Software's Gender Inclusiveness," *Interacting with Computers*, vol. 28, no. 6, 2016, pp. 760–787.
20. C. Bird et al., "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 518–528.
21. T. Fritz et al., "Using Psycho-physiological Measures to Assess Task Difficulty in Software Development," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 402–413; doi:10.1145/2568225.2568266.
22. C. Tantithamthavorn et al., "Automated Parameter Optimization of Classification Techniques for Defect Prediction Models," *Proc. IEEE/ACM 38th Int'l Conf. Software Eng.* (ICSE 16), pp. 321–332.
23. G. Wilson et al., "Good Enough Practices in Scientific Computing," *PLoS Computational Biology*, vol. 13, no. 6, 2017; https://doi.org/10.1371/journal.pcbi.1005510.
24. Z. Li et al., "On the Multiple Sources and Privacy Preservation Issues for Heterogeneous Defect Prediction," to be published in *IEEE Trans. Software Eng.*; https://ieeexplore.ieee.org/document/8168387.
25. N.E. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System," *IEEE Trans. Software Eng.*, vol. 26, no. 8, 2000, pp. 797–814.
26. F. Rahman et al., "Comparing Static Bug Finders and Statistical Prediction," *Proc. 36th Int'l Conf. Software Eng.* (ICSE 14), 2014, pp. 424–434.
27. M. Nagappan et al., "An Empirical Study of Goto in C Code from GitHub Repositories," *Proc. 10th Joint Meeting Foundations of Software Eng.* (ESEC/FSE 15), 2015, pp. 404–414.
28. B. Ray et al., "A Large-Scale Study of Programming Languages and Code Quality in Github," *Comm. ACM*, vol. 60, no. 10, 2017, pp. 91–100.
29. D. Fucci et al., "A Dissection of the Test-Driven Development Process: Does It Really Matter to Test-First or to Test-Last?," *IEEE Trans. Software Eng.*, vol. 43, no. 7, 2017, pp. 597–614.

ABOUT THE AUTHORS

TIM MENZIES is a full professor of computer science at North Carolina State University. He works on software engineering, automated software engineering, and the foundations of software science. Menzies received a PhD in computer science from the University of New South Wales. Contact him at timm@ieee.org; http://menzies.us.

THOMAS ZIMMERMANN is a senior researcher at Microsoft. He works on the productivity of Microsoft's software developers and data scientists. Zimmermann received a PhD in computer science from Saarland University. He is a Distinguished Member of ACM and a Senior Member of IEEE and the IEEE Computer Society. Contact him at tzimmer@microsoft.com; http://thomas-zimmermann.com.

30. T. Menzies et al., "Are Delayed Issues Harder to Resolve? Revisiting Cost-to-Fix of Defects throughout the Lifecycle," *Empirical Software Eng.*, vol. 22, no. 4, 2017, pp. 1903–1935.
31. R. Krishna, T. Menzies, and L. Layman, "Less Is More: Minimizing Code Reorganization Using XTREE," *Information and Software Technology*, vol. 88, 2017, pp. 53–66.
32. M. Kim, T. Zimmermann, and N. Nagappan, "A Field Study of Refactoring Challenges and Benefits," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations of Software Eng.* (FSE 12), 2012, article 50.
33. J. Aranda and G. Venolia, "The Secret Life of Bugs: Going past the Errors and Omissions in Software Repositories," *Proc. 31st Int'l Conf. Software Eng.* (ICSE 09), 2009, pp. 298–308.
34. D. Chen, K.T. Stolee, and T. Menzies, "Replicating and Scaling Up Qualitative Analysis Using Crowdsourcing: A Github-Based Case Study," 2018; https://arxiv.org/abs/1702.08571.
35. K. Petersen and C. Wohlin, "Context in Industrial Software Engineering Research," *Proc. 3rd Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 09), 2009, pp. 401–404.
36. T. Dybå, D.I.K. Sjøberg, and D.S. Cruzes, "What Works for Whom, Where, When, and Why? On the Role of Context in Empirical Software Engineering," *Proc. 2012 ACM/IEEE Int'l Symp. Empirical Software Eng. and Measurement* (ESEM 12), 2012, pp. 19–28.
37. T. Menzies, "Guest Editorial Best Papers from the 2011 Conference on Predictive Models in Software Engineering (PROMISE)," *Information and Software Technology*, vol. 55, no. 8, 2013, pp. 1477–1478.
38. R. Krishna and T. Menzies, "Bellwethers: A Baseline Method for Transfer Learning," 2018; https://arxiv.org/abs/1703.06218.
39. S.-Y. Tan and T. Chan, "Defining and Conceptualizing Actionable Insight: A Conceptual Framework for Decision-Centric Analytics," presentation at 2015 Australasian Conf. Information Systems (ACIS 15), 2015; https://arxiv.org/abs/1606.03510.
40. R. Sawyer, "BI's Impact on Analyses and Decision Making Depends on the Development of Less Complex Applications," *Principles and Applications of Business Intelligence Research*, IGI Global, 2013, pp. 83–95.
41. H.K. Dam, T. Tran, and A. Ghose, "Explainable Software Analytics," 2018; https://arxiv.org/abs/1802.00603.