Editor: **Tim Menzies**
North Carolina State University
tim@menzies.us

The Unreasonable Effectiveness of Software Analytics

Tim Menzies

Call for Submissions

Do you have a surprising result or industrial experience? Something that challenges decades of conventional thinking in software engineering? If so, email a one-paragraph synopsis to tim@menzies.us (use the subject line "REDIRECTIONS: Idea: [your idea]"). If that looks interesting, I'll ask you to submit a 1,000- to 2,400-word article (where each graph, table, or figure is worth 250 words) for review for *IEEE Software*. Note: Heresies are more than welcome (if supported by well-reasoned industrial experiences, case studies, or other empirical results). —*Tim Menzies*

AS RAFAEL PRIKLADNICKI and I commented in last issue's Voice of Evidence article, it's time to ask, "What's surprising about software engineering?"[1] Accordingly, in this article, I explore one of the great mysteries of software analytics: why does it work at all?

Software analytics distills large amounts of low-value data into small chunks of very-high-value data. Such chunks are often predictive; that is, they can offer a somewhat accurate prediction about some quality attribute of future projects—for example, the location of potential defects or the development cost.

In theory, software analytics shouldn't work because software project behavior shouldn't be predictable. Consider the wide, ever-changing range of tasks being implemented by software and the diverse, continually evolving tools used for software's construction (for example, IDEs and version control tools). Let's make that worse. Now consider the constantly changing platforms on which the software executes (desktops, laptops, mobile devices, RESTful services, and so on) or the system developers' varying skills and experience.

Given all that complex and continual variability, every software project could be unique. And, if that were true, any lesson learned from past projects would have limited applicability for future projects.

This turns out not to be the case. One of the lessons of software analytics is that software projects have predictable properties[2] and that at least some of those properties hold for future projects. Stranger still, the number of variables required to make those predictions is small—which means that most of the things we think might affect software quality have little impact in practice.

Not as Complex as We Thought

Consider the task of predicting how long it takes to build software. Given dozens of attributes describing a software project, we can usually guess that project's development time. We can do this using qualitative methods (for example, planning poker,[3] which is favored by the agile community) or parametric-modeling methods (favored by large government projects[4,5]). However we do it, such estimates are surprisingly accurate.[3,5,6]

0740-7459/18/$33.00 © 2018 IEEE

Furthermore, to make those estimates, we need only a remarkably small number of attributes. For example, *feature subset selection* (FSS) is an automatic technique for finding what attributes we can remove without damaging our ability to make a prediction from the data. Recent results show that traditional FSS methods (for example, stepwise regression) can be improved by AI search algorithms that quickly search very large subsets of the attributes to find the most useful ones.[7] Applying FSS to software defect predictors or software effort estimators often reduces datasets with 24 to 42 attributes to sets with only two or three attributes.[5,8] This means that (24 – 3)/24 = 88 percent to (42 – 3)/42 = 93 percent of the collected attributes aren't essential to predicting software quality. That is, much of what we thought was important for quality prediction turns out to be mostly irrelevant. And, more interestingly, we can't tell beforehand what attributes will be the most useful[9] until we test those attributes on real project data.

This result is surprising, to say the least. Software engineers tend to emphasize the complexities, rather than the simplicity, of software projects. Much has been written about what factors might influence a software project, so developers often spend much effort collecting dozens of attributes. Yet for the projects I've mentioned, nearly all those attributes are irrelevant for prediction.

Important Questions

Why are so many things irrelevant? Software engineering data often contains much noise. Collecting data from multiple projects is difficult because the collected data's meaning can vary from project to project.

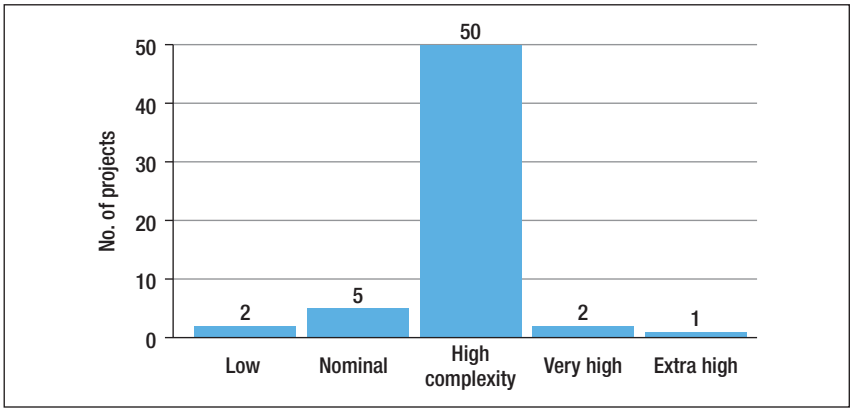

FIGURE 1. The distribution of complexity in a NASA project dataset.[5] Complexity is a constant across nearly all the data, so it could be removed from consideration as an attribute during software analytics.

We can remove such noisy attributes without damaging predictive prowess.

We should also remove most of the closely associated attributes. Suppose a software company assigns its most skilled programmers to mission-critical projects. In that data, "programming skill" would be associated with "criticality." We could dispense with either (but not both) of those attributes without losing important information.

In addition, there's the effect of context. Figure 1 shows data from NASA regarding software projects at the Jet Propulsion Laboratory (JPL). Most of the projects are of high complexity. Thus, feature selection would tend to delete "high complexity" because it's (mostly) a constant across all the data. That is, although no one doubts that software complexity contributes to software cost, for the JPL data, it's mostly irrelevant.

So, what does this mean for the practice of analytics? The previous examples tell us that real software projects can surprise and confound our expectations. In new projects, we should check all expectations (that some factor contributes to software quality). That's the bad news. The good news is that such checks are now fast to run, given the ready availability of data-mining tools (and developers skilled in using them).

More generally, note how these examples are all motivation for this new Redirections department. Our field is rife with any number of truisms that are commonly quoted but rarely checked. Perhaps it's time to reverse that trend. Let's all look over old results in software engineering with a fresh eye and ask, "Which of those results are most applicable?" and "Can we confirm those results using contemporary data?" Hopefully, this department will prompt many such inquiries.

References

1. R. Prikladnicki and T. Menzies, "From Voice of Evidence to Redirections," *IEEE Software*, vol. 35, no. 1, 2018, pp. 11–13
2. T. Menzies and T. Zimmermann, "Software Analytics: So What?," *IEEE Software*, vol. 30, no. 4, 2013,

ABOUT THE AUTHOR

TIM MENZIES is a full professor at North Carolina State University, where he leads the RAISE (Real-World AI for Software Engineering) research group. Contact him at tim@menzies.us; menzies.us.

pp. 31–37; doi:10.1109/MS.2013.86; goo.gl/aGS7wP.

3. K. Molokken-Ostvold and N.C. Haugen, "Combining Estimates with Planning Poker—an Empirical Study," *Proc. 18th Australian Software Eng. Conf.* (ASWEC 07), 2007, pp. 349–358; doi:10.1109/ASWEC.2007.15.
4. F. Sarro, A. Petrozziello, and M. Harman, "Multi-objective Software Effort Estimation," *Proc. 38th Int'l Conf. Software Eng.* (ICSE 16), 2016, pp. 619–630; doi:10.1145/2884781.2884830.
5. Z. Chen et al., "Finding the Right Data for Software Cost Modeling," *IEEE Software*, vol. 22, no. 6, 2005, pp. 38–46; doi:10.1109/MS.2005.151.
6. F. Zhang et al., "Towards Building a Universal Defect Prediction Model with Rank Transformed Predictors," *Empirical Software Eng.*, vol. 21, no. 5, 2016, pp. 2107–2145.
7. M.A. Hall and G. Holmes, "Benchmarking Attribute Selection Techniques for Discrete Class Data Mining," *IEEE Trans. Knowledge and Data Eng.*, vol. 15, no. 6, 2003, pp. 1437–1447.
8. T. Menzies et al., "Defect Prediction from Static Code Features: Current Results, Limitations, New Approaches," *Automated Software Eng.*, vol. 17, no. 4, 2010, pp. 375–407; doi:10.1007/s10515-010-0069-5.
9. R. Krishna and T. Menzies, "Bellwethers: A Baseline Method for Transfer Learning," 3 Dec. 2017; arxiv.org/abs/1703.06218.