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

The Five Laws of SE for AI

Tim Menzies

From the Editor

The January/February 2020 issue of *IEEE Software* includes a landmark in the evolution of this column. Going forward, the column's name will be "SE for AI" (short for "Software Engineering for AI"). We will publish commentaries on the growing field of SE for AI. The rest of this issue's column outlines our new direction. As always, submissions are welcomed and encouraged (1,000–2,400 words, each figure and table counts as 250 words, try to use fewer than 12 references, and keep the discussion practitioner focused). Please submit your ideas to me at timm@ieee.org. —*Tim Menzies*

IT IS TIME to talk about software engineering (SE) for artificial intelligence (AI). As shown in Figure 1, industry is becoming increasingly dependent on AI software. Clearly, AI is useful for SE. But what about the other way around? How important is SE for AI? Many thought leaders in the AI industry are asking how to better develop and maintain AI software (see Figure 2). This column distills that discussion into the following five rules of SE for AI:

1) *AI software mostly isn't about AI*: Much of what we know about SE applies to AI.
2) *AI software needs software engineers*: Software engineers are necessary to tend to AI systems.
3) *Poor SE leads to poor AI*: AI tools suffer when SE is ignored.
4) *Better SE leads to better AI*: AI tools benefit when core SE principles are applied.
5) *SE needs special kinds of AI*: We must better understand how AI tools should be tuned to SE problems.

Digital Object Identifier 10.1109/MS.2019.2954841
Date of current version: 24 December 2019

Rule 1: AI Software Mostly Isn't About AI

In his 2015 talk at the Neural Information Processing Conference, David Sculley[7] offered Figure 3, which represents the size (in lines of code) of Google's software suite. Note how small the AI box is, buried away in the middle of all of the other software. He wrote that "only a small fraction of real-world (machine-learning) systems is composed of the (machine-learning) code. …The required surrounding infrastructure is vast and complex."

Other members of the industry agree that AI tools are a combination of many parts. While some of those components might be described as "core AI," many are not. For example, as shown in Figure 4, Saleema Amershi and her colleagues at Microsoft describe their industrial-AI work as a nine-step pipeline.[8] In it, step 6 (model training) might be called *core AI* since that is where machine-learning

- Data mining algorithms are being used to learn important patterns about software development.[1,2]
- Optimizers are being applied to design software test cases, localise and triage crashes to developers and to monitor their fixes.[3]
- Theorem provers are using used to (e.g.) check security and privacy concerns in cloud-based applications.[4]
- Neural networks are being used to test software for vision systems in self-driving cars.[5]
- AI software is helping pilots understand the requirements of how to land aircraft in adverse conditions.[6]
- And so on and so on.

FIGURE 1. A January 2019 Twitter post by Andrew Ng (https://www.andrewng.org). It was retweeted more than 1,000 times and garnered 3,400 likes.

algorithms are executed. But other steps in the pipeline would be familiar to any software engineer who has worked with databases (for example, steps 1, 2, and 3: requirements, data collection, and data cleaning). Further, according to Amershi, industrial-AI engineers at Microsoft spend only approximately 10% of their time working on core AI components. Hence, we can say:

- The next generation of software engineers should know more about AI.
- Much of the current SE knowledge will be relevant and useful, since a significant amount of the engineers' time will be spent outside of the core AI tools.

Rule 2: AI Software Needs Software Engineers

All software (be it AI or otherwise) needs installation, configuration, maintenance, interfacing to other software, testing, certification, user support, usability additions, and packaging (for distribution). For example, Figure 5 shows an illustration of the kind of AI pipelines being built by Red Hat and used by its clients to distribute and scale AI tools. Red Hat engineer Bill Benton[9] reports that that when we look at these data-mining pipelines, there is much overlap between the activities of data scientists and people in more traditional roles, such as data engineers and application developers. Figure 5 teaches us that the future of software cannot be described in terms of some trite, binary opposition. The future of software is not "SE or AI." Rather, it will be a rich and powerful mix of ideas from the two disciplines.

FIGURE 2. A sample of the industrial applications of AI to SE.

Rule 3: Poor SE Leads to Poor AI

Since AI software is still software, it follows that poor SE leads to poor AI. Sculley's 2015 talk offers us an example. He reported that Google's machine-learning developers used all of the attributes in the company's data dictionaries to learn predictive models of browsing habits. That led to problems, since any subsequent change in the data dictionary meant that all of the data mining had to be done again. In SE terms, Google had introduced a technical debt (that is, something that will consume maintenance money at some future date) by violating the principles of coupling and cohesion. Maintainable systems are loosely coupled (but internally tightly cohesive). Google's classifiers, on the other hand, were tightly coupled with their data

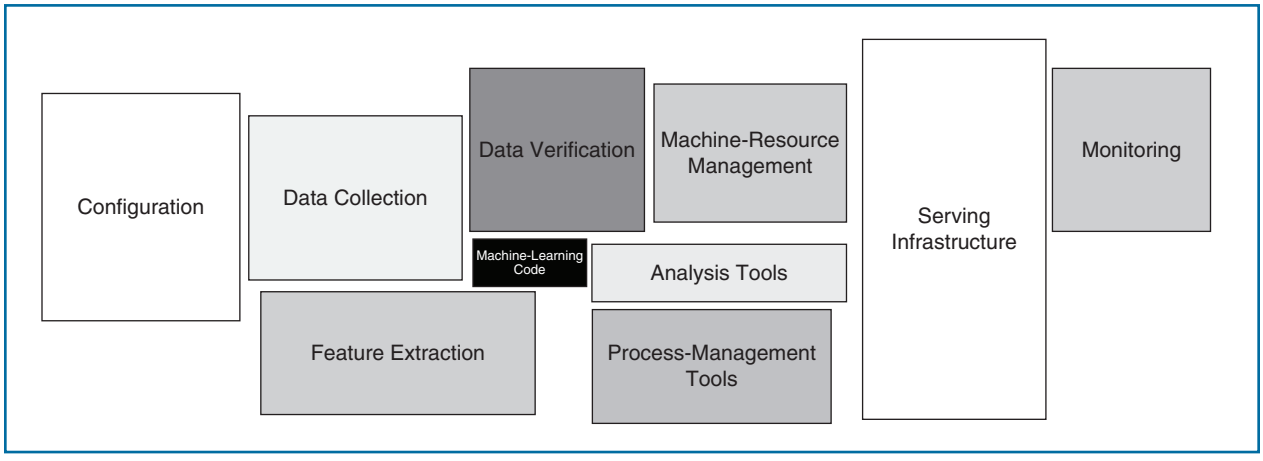

FIGURE 3. The lines of code in Google's AI-related software. The tiny black box in the middle represents the core AI code, while the rest of the diagram shows support software.[7]

dictionaries. A better design, which would have had looser coupling, might have been to apply some sort of feature weighting to the data and connect only to the features that were the most influential.

Stepping back from the details of this example, the more general point to be made is that much has been learned through the decades about how to build software. While not all of that knowledge applies to the new generation of AI tools, much of it does. Hence, while the wise AI engineer should study AI algorithms, he or she must also study SE.

Rule 4: Better SE Leads to Better AI

While poor SE can lead to problems with AI, the good news is that better SE can lead to better AI. For example, many industrial data scientists make extensive use of the Python scikit-learn toolkit data-mining package.[10] Started in 2007 as a Google Summer of Code project by David Cournapeau, numerous releases have appeared on a roughly three-month cycle, and a thriving international community has been leading the development effort. At the time of this writing, during the past month, this software has been maintained and extended by dozens of authors from across the planet (specifically, excluding merges, 50 authors have pushed 119 commits to master and 119 commits to all branches to make changes to thousands of lines of code in 279 files). All of this is possible because scikit-learn uses state-of-the-art open source SE methods (continuous integration, cloud-based testing with Travis CI, git, GitHub, and so forth).

> While the wise AI engineer should study AI algorithms, he or she must also study SE.

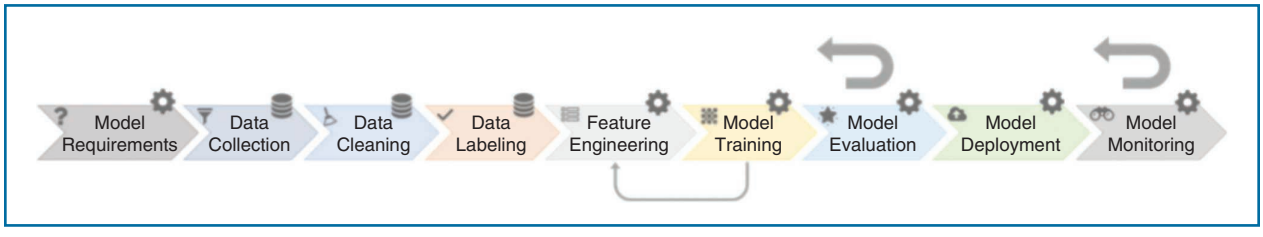

FIGURE 4. The nine-step process used at Microsoft to build and deploy machine-learning applications.[8]

To be fair, it is not necessarily true that applying methods such as open source development always improve AI. Houssem Braiek and his colleagues offer a detailed study of the role of open source development in modern machine-learning software.[11] Their analysis is a nuanced discussion of what can go right and wrong when modern SE is applied to machine-learning tools. That said, their paper agrees that there is much potential value in applying SE to AI.

> It is not necessarily true that applying methods such as open source development always improve AI.

Rule 5: SE Needs Special Kinds of AI

Another reason to apply SE to AI is that someone (that is, a software engineer) has to tinker with the AI software to make it usable. In the following examples, off-the-shelf AI tools were successfully adapted to SE domains but only after software engineers careful adjusted them.

Software analytics usually applies standard AI data-mining algorithms as "black boxes"[12] where researchers do not dabble with the internal design choices of those AI tools. This is not ideal. Much recent work has shown that such automatic tools can adjust the control parameters of, say, a data miner to greatly improve its predictive performance.[13, 14]

Dave Binkley and others note that text-mining methods for understanding software code often mistakenly equate word frequency with word importance. This is clearly the wrong thing to do, since even the number of occurrences of a variable name, such as "tmp," is not necessarily indicative of its importance. Hence, they strongly advise against applying off-the-shelf AI text mining in the software domain.

Another example of how not to use AI text-mining tools comes from research into sentiment analysis. Sentiment analysis is the art of measuring the mood of, say, the writer of a code commit message (for example, angry or happy). The problem is that most sentiment-analysis tools are trained on non-SE data (for instance, *The Wall Street Journal* or Wikipedia). Nicole Novielli and her colleagues show that this is a poor way to apply these tools. They recently developed their own sentiment analysis for SE. After retraining those tools on an SE corpus, they found better performance at predicting sentiment and more agreement between different sentiment tools.[15]

Yet another example of "SE needs to adjust AI tools" comes from the "naturalness" work of Prem Devanbu and his colleagues.[16] Software has the property that some terms are used very often, and everything else is used with exponentially less frequency. This means that n-gram "language models" (where frequency counts of the last N terms are used to predict the next term) can be very insightful for 1) recommending what should be used next and 2) predicting that the next token is unusual and possibly erroneous. Note that this "natural" approach uses AI tools but does so in a novel manner that better adjusts them to the SE task.

In summary, so-called general AI tools may not be general at all. Rather, they are powerful in their home domain but must be used with

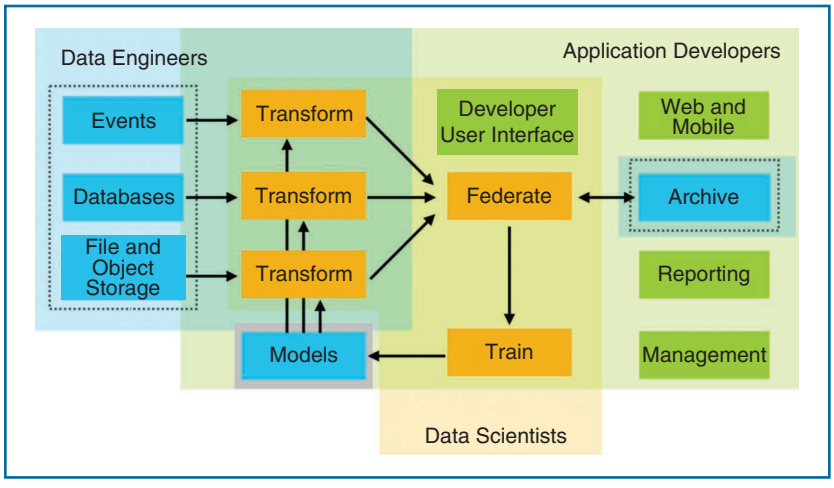

FIGURE 5. The overlap between the activities of data scientists, data engineers, and application developers.[9]

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 timm@ieee.org or http://menzies.us.

care if applied to new areas, such as SE. Hence, it is not good enough to take AI tools developed elsewhere and apply them verbatim to SE problems. Software engineers are needed to develop AI tools that are better suited to the particulars of SE problems.

References

1. J. Czerwonka, N. Nagappan, W. Schulte, and B. Murphy, "CODEMINE: Building a software development data analytics platform at Microsoft," *IEEE Softw.*, vol. 30, no. 4, pp. 64–71, July–Aug. 2013. doi: 10.1109/MS.2013.68.
2. A. T. Misirli, A. B. Bener, and R. Kale, "AI-based software defect predictors: Applications and benefits in a case study," *AI Mag.*, vol. 32, no. 2, pp. 57–68, June 2011. doi: 10.1609/aimag.v32i2.2348.
3. N. Alshahwan et al., "Deploying search based software engineering with Sapienz at Facebook," in *Proc. Int. Symp. Search Based Software Engineering (SSBSE)*, 2018, pp. 3–45.
4. J. Backes, B. Cook, A. Gacek, N. Rungta, and M. W. Whalen, "One-click formal methods," *IEEE Softw.*, vol. 36, no. 6, pp. 61–65, Nov.–Dec. 2019. doi: 10.1109/MS.2019.2930609.
5. R. B. Abdessalem, S. Nejati, L. C. Briand, and T. Stifter, "Testing vision-based control systems using learnable evolutionary algorithms," in *Proc. 2018 IEEE/ACM 40th Int. Conf. Software Engineering (ICSE)*. doi: 10.1145/3180155.3180160.
6. J. Krall, T. Menzies, and M. Davies, "Learning mitigations for pilot issues when landing aircraft (via multiobjective optimization and multiagent simulations)," *IEEE Trans. Human–Mach. Syst.*, vol. 46, no. 2, pp. 221–230, Apr. 2016. doi: 10.1109/THMS.2015.2509980.
7. D. Sculley et al., "Hidden technical debt in machine learning systems," in *Proc. 28th Int. Conf. Neural Information Processing Systems*, vol. 2. Cambridge, MA: MIT Press, pp. 2503–2511. [Online]. Available: http://dl.acm.org/citation.cfm?id=2969442.2969519
8. S. Amershi et al., "Software engineering for machine learning: A case study," in *Proc. 41st Int. Conf. Software Engineering: Software Engineering in Practice (ICSE-SEIP '19)*, pp. 291–300. doi: 10.1109/ICSE-SEIP.2019.00042.
9. W. Benton, "Machine learning and discovery with Kubernetes," in *Proc. Software Engineering Machine Learning Int. Symp.*, 2019.
10. F. Pedregosa et al., "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, Oct. 2011.
11. H. B. Braiek, F. Khomh, and B. Adams, "The open-closed principle of modern machine learning frameworks," in *Proc. 2018 IEEE/ACM 15th Int. Conf. Mining Software Repositories (MSR)*, pp. 353–363.
12. D. Binkley, D. Lawrie, and C. Morrell, "The need for software specific natural language techniques," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 2398–2425, Aug. 2018.
13. C. Tantithamthavorn, S. McIntosh, A. Hassan, and K. Matsumoto, "Automated parameter optimization of classification techniques for defect prediction model," in *Proc. 2016 IEEE/ACM 38th Int. Conf. Software Engineering (ICSE)*. doi: 10.1145/2884781.2884857.
14. A. Agrawal and T. Menzies, "Is AI different for SE?" NC State Univ., Aug. 26, 2019. [Online]. Available: https://repository.lib.ncsu.edu/bitstream/handle/1840.20/36964/TR-2019-4.pdf?sequence=1&isAllowed=y
15. N. Novielli, D. Girardi, and F. Lanubile, "Benchmark study on sentiment analysis for software engineering research," in *Proc. 2018 IEEE/ACM 15th Int. Conf. Mining Software Repositories (MSR)*. doi: 10.1145/3196398.3196403.
16. B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the 'naturalness' of buggy code," in *Proc. 38th Int. Conf. Software Engineering (ICSE)*, 2016, pp. 428–439.