Software Analytics: So What?

Tim Menzies, West Virginia University

Thomas Zimmermann, Microsoft Research

IN THIS SPECIAL issue of *IEEE Software*, we invited submissions that reflected the benefits (and drawbacks) of software analytics. The response was overwhelming. Software analytics is an area of explosive growth, and we had so many excellent submissions that we had to split this special issue into two volumes—you'll see even more content in the September/October issue. We divided the articles on conceptual grounds, so both volumes will feature equally excellent work.

To better frame these articles, we offer some definitions and historical perspectives on software analytics. Specifically, we describe where the field was, where it is, and where it might be going.

What Is Software Analytics?

Thanks to the Internet and open source, there's now so much data about software projects that it's impossible to manually browse through it all:

- As of late 2012, our Web searches show that Mozilla Firefox had 800,000 bug reports, and platforms such as Sourceforge.net and GitHub hosted 324,000 and 11.2 million projects, respectively.

TABLE 1 Repositories of software engineering data.

Repository	URL
Bug Prediction Dataset	http://bug.inf.usi.ch
Eclipse Bug Data	www.st.cs.uni-saarland.de/softevo/bug-data/eclipse
FLOSSMetrics	http://flossmetrics.org
FLOSSMole	http://flossmole.org
International Software Benchmarking Standards Group (IBSBSG)	www.isbsg.org
ohloh	www.ohloh.net
PROMISE	http://promisedata.googlecode.com
Qualitas Corpus	http://qualitascorpus.com
Software Artifact Repository	http://sir.unl.edu
SourceForge Research Data	http://zerlot.cse.nd.edu
Sourcerer Project	http://sourcerer.ics.uci.edu
Tukutuku	www.metriq.biz/tukutuku
Ultimate Debian Database	http://udd.debian.org

- The PROMISE repository of software engineering data has grown to more than 100 projects and is just one of more than a dozen open source repositories that are readily available to industrial practitioners and researchers; see Table 1 for more.

To handle this data, many practitioners and researchers have turned to analytics—that is, the use of analysis, data, and systematic reasoning for making decisions.[1] We can define software analytics as follows: "Software analytics is analytics on software data for managers and software engineers with the aim of empowering software development individuals and teams to gain and share insight from their data to make better decisions."[2]

Such insights include, but might not be limited to, actionable advice on how to improve software projects. Due to the volume of data, finding these insights typically requires some degree of automation, usually combined with human involvement. As the "Applications of Software Analytics" sidebar shows, augmenting analytics with automation has led to impressive results in a wide range of application areas.

Analytics Must Be Real Time and Actionable

An important part of software analytics is that they include actionable advice. For example, if the manager is driving toward a cliff, we don't want to distract her with analytics telling her about the clouds in the sky or the flowers on the side of the road. Instead, we want our smart analytics to shout in her ear, "There's a cliff up ahead! Turn left immediately!"

What happens next is up to the manager (who might elect to ignore this advice and make some decision based on his or her own gut instincts; http://newsroom.accenture.com/article_display.cfm?article_id=4777). But we know from our own experience that if we don't deliver relevant advice, then there's little hope that a manager will use any of our analytics advice.

In practice, actionable analytics means that those analytics must be available in real time—faster than the rate of change of effects within a project. Decisions should be based on recent, not outdated, data. This is an important point to make because traditional data collection and analysis techniques might be too slow. For example, at an International Conference on Software Engineering, Forrest Shull, the editor in chief of this magazine, told his audience in a tutorial that the current pace of manual methods in empirical software engineering might not keep up with the fast pace of modern agile software practices.[3]

As to what constitutes real time, that's a domain-dependent issue. For example, for stock trading, real time might mean microseconds, whereas for process change in an organization, it might mean before the December merit

APPLICATIONS OF SOFTWARE ANALYTICS

We offer here a partial list of software project artifacts that have been studied within software analytics. For more examples, see the rest of this special issue as well as recent conference proceedings of the PROMISE conference, the Mining Software Repositories (MSR) conference, and any other conference or journal on software engineering:

- combining software product information with apps store data[1,2];
- using process data to predict overall project effort[3];
- using software process models to learn effective project changes[4];
- using operating system logs that predict software power consumption[5];
- exploring product line models to configure new applications[6];
- mining natural language requirements to find links between components[7];
- mining performance data[8,9];
- using XML descriptions of design patterns to recommend particular designs[10];
- using email lists to understand the human networks inside software teams[11];
- linking emails to source code artifacts and classifying their content[12];
- using execution traces to learn normal interface usage patterns[13];
- using bug databases to learn defect predictors that guide inspection teams to where the code is most likely to fail[14–16] and to classify changes as clean or buggy[17];
- using security data to identify indicators for software vulnerabilities[18];
- using visualization to support program comprehension[19];
- using software ontologies to enable natural language queries[20]; and
- mining code clones to assess the implications of cloning and copy/paste in software.[21,22]

References

1. M. Harman, Y. Jia, and Y. Zhang, "App Store Mining and Analysis: MSR for App Stores," *Proc. Mining Software Repositories*, IEEE, 2012, pp. 108–111.
2. I.J.M. Ruiz et al., "Understanding Reuse in the Android Market," *Proc. 20th IEEE Int'l Conf. Program Comprehension* (ICPC), IEEE, 2012, pp. 113–122.
3. E. Kocaguneli, T. Menzies, and J. Keung, "On the Value of Ensemble Effort Estimation," to be published in *IEEE Trans. Software Eng.*; http://menzies.us/pdf/11comba.pdf.
4. D. Rodríguez et al., "Multiobjective Simulation Optimisation in Software Project Management," *Proc. Genetic and Evolutionary Computation Conf.*, ACM, 2011, pp. 1883–1890.
5. A. Hindle, "Green Mining: A Methodology of Relating Software Change to Power Consumption," *Proc. Mining Software Repositories*, IEEE, 2012, pp. 78–87.
6. A. Salam Sayyad, T. Menzies, and H. Ammar, "On the Value of User Preferences in Search-Based Software Engineering: A Case Study in Software Product Lines," to be published in *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2013.
7. J. Huffman Hayes et al., "Advancing Candidate Link Generation for Requirements Tracing: The Study of Methods," *IEEE Trans. Software Eng.*, vol. 32, no. 1, 2006, pp. 4–19.
8. Z. Ming Jiang et al., "Automated Performance Analysis of Load Tests," *Proc. Intl. Conf. Software Maintenance*, IEEE, 2009, pp. 125–134.
9. S. Han et al., "Performance Debugging in the Large via Mining Millions of Stack Traces," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2012, pp. 145–155.
10. F. Palma, H. Farzin, and Y.-G. Gueheneuc, "Recommendation System for Design Patterns in Software Development: A DPR Overview," *Proc. 3rd Int'l Workshop Recommendation Systems for Software Eng.*, IEEE, 2012, pp. 1–5.
11. C. Bird et al., "Mining Email Social Networks," *Proc. Mining Software Repositories*, ACM, 2006, pp. 137–143.
12. A. Bacchelli et al., "Content Classification of Development Emails," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2012, pp. 375–385.
13. N. Gruska, A. Wasylkowski, and A. Zeller, "Learning from 6,000 Projects: Lightweight Cross-Project Anomaly Detection," *Proc. 19th Int'l Symp. Software Testing and Analysis* (ISSTA), ACM, 2010, pp. 119–130.
14. T. Menzies, J. Greenwald, and A. Frank, "Data Mining Static Code Attributes to Learn Defect Predictors," *IEEE Trans. Software Eng.*, Jan. 2007; http://menzies.us/pdf/06learnPredict.pdf.
15. T.J. Ostrand, E.J. Weyuker, and R.M. Bell, "Where the Bugs Are," *Proc. 2004 ACM SIGSOFT Int'l Symp. Software Testing and Analysis,* ACM, 2004, pp. 86–96.
16. S. Kim et al., "Predicting Faults from Cached History," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2007, pp. 489-498.
17. S. Kim, E.J. Whitehead Jr., and Y. Zhang, "Classifying Software Changes: Clean or Buggy?," *IEEE Trans. Software Eng.*, vol. 34, no. 2, 2008, pp. 181–196.
18. Y. Shin et al., "Evaluating Complexity, Code Churn, and Developer Activity Metrics as Indicators of Software Vulnerabilities," *IEEE Trans. Software Eng.*, vol. 37, no. 6, 2011, pp. 772–787.
19. R. Wettel, M. Lanza, and R. Robbes, "Software Systems as Cities: A Controlled Experiment," *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2011, pp. 551–560.
20. M. Würsch et al., "Supporting Developers with Natural Language Queries," *Proc. Int'l Conf. Software Eng.,* IEEE CS, 2010, pp. 165–174.
21. M. Kim et al., "An Empirical Study of Code Clone Genealogies," *Proc. European Software Eng. Conf.,* ACM, 2005, pp. 187–196.
22. C. Kapser and M.W. Godfrey, "Cloning Considered Harmful," *Proc. Working Conf. Reverse Eng.,* IEEE, 2006, pp. 19–28.

EARLY "GLOBAL" MODELS AND SOFTWARE ANALYTICS

As soon as people started programming, it became apparent that programming was an inherently buggy process. As recalled by Maurice Wilkes,[1] speaking of his programming experiences from the early 1950s: "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 remainder of my life was going to be spent in finding errors in my own programs."

It took several decades to gather the experience required to quantify the size/defect relationship. In 1971, Fumio Akiyama[2] described the first known "size" law, saying the number of defects D was a function of the number of LOC; specifically, $D = 4.86 + 0.018 * i$. In 1976, Thomas McCabe argued that the number of LOC was less important than the complexity of that code.[3] He argued that code is more likely to be defective when his "cyclomatic complexity" measure was over 10.

Not only is programming an inherently buggy process, it's also inherently difficult. Based on data from 63 projects, Barry Boehm[4] proposed in 1981 an estimator for development effort that was exponential on program size: effort = a * KLOCb * EffortMultipliers, where $2.4 \le a \le 3$ and $1.05 \le b \le 1.2$.

References

1. M. Wilkes, *Memoirs of a Computer Pioneer*, MIT Press, 1985.
2. F. Akiyama, "An Example of Software System Debugging," *Information Processing*, vol. 71, 1971, pp. 353–359.
3. T. McCabe, "A Complexity Measure," *IEEE Trans. Software Eng.*, vol. 2, no. 4, 1976, pp. 308–320.
4. B. Boehm, *Software Engineering Economics*, Prentice-Hall, 1981.

awards. For more details on this real-time approach to analytics, see "Leveraging the Crowd: How 48,000 Users Helped Improve Lync Performance" by Robert Musson and his colleagues in this special issue.

Analytics Means—Sharing Information

To a large extent, analytics is about what software projects can learn from themselves and each other. Looking back at decades of research in this area, we see claims that software analytics let us share different things between projects:

- sharing models (one of the early models was proposed by Fumio Akiyama and says that we should expect more than a dozen bugs per KLOC; see the sidebar "Early 'Global' Models and Software Analytics"),
- sharing insights (for example, Christian Bird and his colleagues found that in the case of Windows Vista, it's possible to build high-quality software using distributed teams, just as long as the management is structured around code functionality—who works on what—and not merely on geography—who sits where[4]),
- sharing data (as in Table 1's list of repositories), and
- sharing methods (the techniques by which we can convert data into models to find local models).

The ordering of the items in this list is very significant. We've seen a major change in the nature of software analytics. In the past, most research focused on the start of this list and searched for general models; today, it focuses on the end of this list (sharing general methods) to enable the discovery of local lessons.

Forty years ago, the goal was to find and share "the" model of software development, which assumed a single model existed that was global to all software projects. But more recently, the goal of analytics has changed—the research community has accepted that lessons learned from one project can't always be applied verbatim to another. For example, as one of us (Zimmermann) reported at the 2009 International Symposium on the Foundations of Software Engineering, a model that predicts defects in Internet Explorer may not be able to predict defects in Firefox.

Rather than searching for global models, the focus is now on local methods. This shift was caused by two developments. In the late 20th century, a whole new generation of data mining algorithms became available, along with large amounts of data from open source projects. Consequently, more researchers applied more mining algorithms to more data.[5] A side effect of all that work was the growing realization that one global model wouldn't cover all software projects.

The ability to quickly reason about large amounts of data from a particular site has become of vital importance—we used to expect that we could share models, but if project models are project-specific, and if we reuse your model on our project, it could lead to inappropriate and costly management decisions. It has become more important today to discuss and document the methods by which a data scientist might turn local data into relevant and useful local models (a catalog of such methods appears at http://research.microsoft.com/en-us/events/dapse2013).

PRINCIPLES FOR SOFTWARE ANALYTICS

Recently, in the *Inductive Engineering Manifesto*, we made some notes on what characterizes the difference between academic and industrial data mining.[1] We systematized the results of that analysis into the following seven principles:

1. *Users before algorithms.* Data mining algorithms are only useful in industry if users fund their use in real-world applications. The user perspective is vital to inductive engineering. The space of models that can be generated from any dataset is very large. If we understand and apply user goals, then we can quickly focus an inductive engineering project on the small set of most crucial issues.
2. *Plan for scale.* In any industrial application, the data mining method is repeated multiples time to answer an extra user question, make some enhancement and/or bug fix to the method, or to deploy it to a different set of users. That is, for serious studies, to ensure repeatability, the entire analysis should be automated using some high-level scripting language.
3. *Early feedback.* Continuous and early feedback from users allows needed changes to be made as soon as possible and without wasting heavy up-front investment. Prior to conducting very elaborate studies, tray applying very simple tools to gain rapid early feedback.
4. *Be open-minded.* It's unwise to enter into an inductive study with fixed hypotheses or approaches, particularly for data that hasn't been mined before. Don't resist exploring additional avenues when a particular idea doesn't work out. We advise this because data likes to surprise: initial results often change the goals of a study when business plans are based on issues irrelevant to local data.
5. *Do smart learning.* Important outcomes are riding on your conclusions. Make sure that you check and validate them. There are many such validation methods such as repeat the analysis *N* times on, say, 90 percent of the available data—then check how well your conclusions hold across all those samples.
6. *Live with the data you have.* You go mining with the data you have, not the data you might want or wish to have at a later time. Because we may not have control over how data is collected, it's wise to cleanse the data prior to learning. For example, before learning from a dataset, conduct instance or feature selection studies to see what spurious data can be removed.[2]
7. *Broad skill set, big toolkit.* Successful inductive engineers routinely try multiple inductive technologies. To handle the wide range of possible goals of different goals, an inductive engineer should be ready to deploy a wide range of tools. Note that the set of useful inductive technologies is large and constantly changing. So use tools supported by a large ecosystem of developers who are constantly building new learners and fixing old ones.

References

1. T. Menzies et al., "The Inductive Software Engineering Manifesto: Principles for Industrial Data Mining," *Proc. Int'l Workshop Machine Learning Technologies in Software Eng.* (MALETS), 2011; http://menzies.us/pdf/11manifesto.pdf.
2. M. Shepperd et al., "Data Quality: Some Comments on the NASA Software Defect Data Sets," to be published in *IEEE Trans. Software Eng.*, 2013.

General Principles for Analytics and the Need for Skilled People

Any field needs some general principles to guide

- novices in their journey from beginner to expert,
- managers when they assess potential hires,
- academics when they design subjects or degrees,
- professional bodies when they design accreditation programs,
- the identification of gaps in current techniques, and
- the design and implementation of new and better techniques.

The sidebar "Principles for Software Analytics" offers some tips on how to do this. Note that they're more concerned with usage patterns and interactions with the consumer of data insights than with particular algorithms.

In the era of Google-style inference and cloud computing, it's a common belief that a company can analyze large amounts of data merely by building (or renting) a CPU farm, then running some distributed algorithms, perhaps using Hadoop (http://hadoop.apache.org) or some other distributed inference mechanism.

This isn't the case. In our experience, while having many CPUs is (sometimes) useful, the factors that select successful software analytics rarely include the hardware. More important than the hardware is how that hardware is used by skilled data scientists.

Another misconception we often see relates to the role of software. Some managers think that if they acquire the right software tools—Weka, Matlab, and so on—then all their analytical problems will be instantly solved.

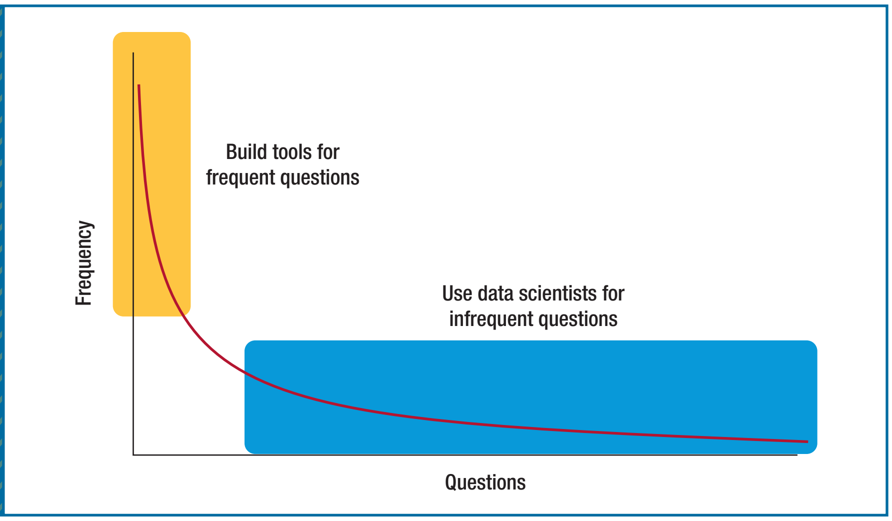


FIGURE 1. The frequency of analytics questions. A small number of questions are very frequent and therefore should be supported by tools (orange region), whereas the long tail of questions that are more unique and asked less frequently should be addressed by data scientists (blue). As the analytics domain matures, we expect the orange area to grow because tools will become more powerful and cheaper to develop.

Nothing could be further from the truth. All the standard data analysis toolkits come with built-in assumptions that might be suitable for particular domains. Hence, a premature commitment to particular automatic analysis tools can be counterproductive. When it isn't clear what the important factors in a domain are, a data scientist must make many ad hoc queries, to clarify the issues in that domain. Subsequently, once the analysis method stabilizes, it becomes possible to build tools to automate the routine and repeated analysis tasks.

In practice, most analytics projects mature by moving up along the curve in Figure 1. Initially, we might start in the blue and then move into the orange region. This diagram leads to one of our favorite mantras for software analytics: for new or infrequent problems, deploy the data scientists before deploying tools.

Different and Distinct Kinds of Analytics

Expert data scientists know that they must choose their methods to best match the distinctive features of their particular kind of analytics. As this field matures, we'll see more and more recognition of distinct subtypes of analytics, each requiring different tools and techniques. Here are the distinctions we can see within current work on software analytics—it's hardly a complete set, and it will certainly change over time. However, we make this list to make the point that "analytics" is a broad area with many exciting and challenging issues and room for development.

The first important distinction is between internal and external analytics, both of which have significant implications. In this context, one issue is live versus stale data: while the internal team can access current data, there are many practical challenges associated with shipping a copy of the data outside the organization's firewalls to an external team. A second issue is privacy. If an external team wants to access the data, it might have to perform some anonymization of the information. This can be a problem because altering data even to ensure privacy can damage the signal in that data. Recently, we've had some success with an instance-based privacy algorithm that clears the space around each example, then jumps the remaining data some small distance into that cleared space. The resulting data contains none of the original individuals, but preserves the hyperspace boundaries between conclusions.[6]

Another important distinction is between quantitative and qualitative methods. Quantitative methods include the traditional automatable tasks performed by statistical packages and data mining tools,[7,8] whereas qualitative methods are typically more manual and require extensive user interaction. A common myth with qualitative methods is that they're less-than-rigorous and somehow less useful than quantitative methods, but we haven't found this to be the case. See "Developer Dashboards: The Need for Qualitative Analytics" by Olga Baysal, Reid Holmes, and Michael Godfrey in this issue for an excellent example.

Yet another important distinction is between data mining tools and interactive tools. Data mining tools are typically automatic and run to produce one conclusion, whereas visualization tools give users more control over the output and the ability to generate ad hoc on-the-fly reports on any aspect of the data. Much current software analytics work is focused on data mining, but this issue features some exceptions to this rule, which can be found in both the roundtable discussion and "Looking under the Lamppost for Useful Software Analytics" by Philip Johnson.

It's also important to distinguish the audience for analytics results, which could include developers, testers, development leads, test leads, managers, and researchers, all of whom have different analysis and data needs.[9]

Finally, we wish to distinguish exploratory versus deployment analytics. In an exploratory analytical study, it's often unclear how to add value to the business using analytics. Hence, in this stage, a data scientist's work is often preliminary and perhaps includes many dead ends. If the exploratory results are successful, there might be a business case for moving to deployment analytics. In this second phase, the goals are better understood, and the team (which might be much larger than the preliminary analytics team) works to integrate the analysis method into the organization's information systems. If the preliminary analysis takes weeks to months, deployment analytics can take months to years. For an example of the cost and benefits of deployment analytics, see "CODEMINE: Building a Software Development Data Analytics Platform at Microsoft" by Jacek Czerwonka and his colleagues in this special issue.

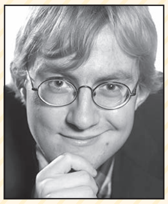
This is an exciting time for those of us involved in data science and software analytics. Looking into the very near future, we can only predict more use of analytics. By 2020, we would predict

- more and different data,
- more algorithms,
- faster decision making with the availability of more data and faster release cycles,
- more people involved in analytics as it becomes more routine to mine data,
- more education as more people analyze and work with data,
- more roles for data scientists and developers as this field matures with specialized subareas,
- more real-time analytics to address the challenges of quickly finding patterns in big data,
- more analytics for software systems such as mobile apps and games, and
- more impact of social tools in analytics.

As an example of this last point, check out "Human Boosting" by Harsh Pareek and Pradeep Ravikumar, which discusses how to boost human learning with the help of data miners.[10] In the very near future, this kind of human(s)-in-the-loop analytics will become much more prevalent.

ABOUT THE AUTHORS

TIM MENZIES is a full professor in computer science at West Virginia University. His research focuses on combining carbon and silicon intelligence to produce smarter communities. He's an associate editor of *IEEE Transactions on Software Engineering*, the *Automated Software Engineering Journal*, and the *Empirical Software Engineering Journal*. Menzies received a PhD in artificial intelligence from the University of New South Wales. Contact him at tim@menzies.us or via http://menzies.us.



THOMAS ZIMMERMANN is a researcher in the Empirical Software Engineering Group at Microsoft Research, adjunct assistant professor at the University of Calgary, and affiliate faculty at the University of Washington. His research interests include empirical software engineering, mining software repositories, development tools, social networking, and games analytics. He's an associate editor of *IEEE Software* and the *Empirical Software Engineering Journal*. Zimmermann received a PhD from Saarland University. Contact him at tzimmer@microsoft.com or via http://thomas-zimmermann.com.

Acknowledgements
The work of such a special issue falls mostly on the authors and reviewers, and we're very appreciative of all those who took the time to write and comment on these articles. The work of the reviewers was particularly challenging because their feedback was required in a very condensed timetable. Accordingly, we offer them our heartfelt thanks. We're also grateful to the *IEEE Software* production team for their hard work in assembling and editing all these articles.

References
1. T.H. Davenport, J.G. Harris, and R. Morison, *Analytics at Work: Smarter Decisions, Better Results*, Harvard Business Review Press, 2010.
2. R. Buse and T. Zimmermann, "Information Needs for Software Development Analytics," *Proc. Int'l Conf. Software Eng.* (ICSE), IEEE CS, 2012; http://thomas-zimmermann.com/publications/details/buse-icse-2012.
3. T. Menzies and F. Shull, "Empirical Software Engineering," tech. briefing, *Proc. Int'l Conf. Software Eng.*, IEEE CS, 2011, http://2011.icse-conferences.org/technical-briefings/#381.
4. C. Bird et al., "Does Distributed Development Affect Software Quality? An Empirical Case Study of Windows Vista," *Proc. 31st Int'l Conf. Software Eng.*, IEEE CS, 2009, pp. 518–528.
5. A. Porter and R. Selby, "Empirically Guided Software Development Using Metric-Based Classification Trees," *IEEE Software*, vol. 7, no. 2, 1990, pp. 46–54.
6. F. Peters et al., "Balancing Privacy and Utility in Cross-Company Defect Prediction," to be published in *IEEE Trans. Software Eng.*, 2013.
7. I. Witten, E. Frank, and M. Hall, *Data Mining: Practical Machine Learning Tools and Techniques*, Morgan Kaufmann, 2011.
8. R. Duda, P. Hart and D. Stork, *Pattern Recognition*, 2nd ed., Wiley-Interscience, 2000.
9. E. Kocaganeli et al., "Distributed Development Considered Harmful?," to be published in *Proc. Int'l Conf. Software Eng.* (ICSE), IEEE CS, 2013.
10. H. Pareek and P. Ravikumar, "Human Boosting," to be published in *Proc. Int'l Conf. Machine Learning*, 2013; http://jmlr.csail.mit.edu/proceedings/papers/v28/pareek13.pdf.