Software Analytics in Practice

Dongmei Zhang, Shi Han, Yingnong Dang, Jian-Guang Lou, and Haidong Zhang, Microsoft Research Asia

Tao Xie, University of Illinois at Urbana-Champaign

// The StackMine project produced a software analytics system for Microsoft product teams. The project provided several lessons on applying software analytics technologies to make an impact on software development practice. //

THE SOFTWARE DEVELOPMENT process produces various types of data such as source code, bug reports, check-in histories, and test cases. Over the past few decades, researchers have used analytics techniques on such data to better understand software quality (for some examples, see the sidebar). Also during that period, Internet access has become widely available, and software services now play an indispensable role in everyday life. Much richer types of data at much larger scales now help practitioners better understand how software and services perform in real-world settings and how end users employ them.[1]

To address the increasing importance and abundance of data in the software domain, we formed Microsoft Research Asia's Software Analytics Group (http://research.microsoft.com/groups/sa) in 2009. We proposed the concept of software analytics to expand the scope of analyzing software artifacts. Software analytics aims to obtain insightful and actionable information from software artifacts that help practitioners accomplish tasks related to software development, systems, and users.

Although much research has covered software analytics, little work has covered its impact on software development practice.[2,3] Software analytics can affect software development practice in different ways.[4–6] For example,

- software practitioners could conduct software analytics by themselves,
- researchers from academia or industry could collaborate with software practitioners from software companies or open source communities and transfer their analytics technologies to real-world use, or
- practitioners could organically adopt analytics technologies developed by researchers.

However, no matter how this is pursued, it remains a huge challenge.

We've worked on a number of software analytics projects that have had a high impact on software development practice.[7–10] To illustrate the lessons we've learned from them, we describe our experiences developing StackMine, a scalable system for postmortem performance debugging.[7]

The Promise and Challenges of Software Analytics

As we mentioned before, software analytics aims to obtain insightful and actionable information. Insightful information conveys knowledge that's meaningful and useful for practitioners performing a specific task. Actionable information is information with which practitioners can devise concrete ways (better than any existing way) to complete that task.

For example, ranked performance bottlenecks represented by sequences of function calls can help performance analysts focus the investigation scope of the underlying performance issues. This information also provides guidance on where to investigate. So, it's both insightful and actionable.

Software analytics focuses on the trinity of software development, systems, and users, with the ultimate goal of improving development productivity, software quality, and user experience

0740-7459/13/$31.00 © 2013 IEEE

(see Figure 1).[11–13] Obviously, improvements in the software development process will improve development productivity. Software quality focuses on issues such as reliability, performance, and security, whereas assessment of the user experience focuses on the user's perspective. In general, software analytics employs these primary technologies:

- large-scale computing to handle large-scale datasets,
- machine-learning-based and data-mining-enabled analysis algorithms, and
- information visualization to help with data analysis and presenting insights.

The target audience of practitioners is broad, including developers, testers, program managers, software management personnel, designers, usability engineers, service engineers, and support engineers.

Software analytics has the potential to impact practice for two main reasons. First, the data sources under study come from real-world settings. For example, open source communities naturally provide a huge data vault of source code, bug reports, check-in history, and so on. Better yet, the vault is active and evolving, which makes the data sources fresh and live. Second, as Figure 1 illustrates, the discipline of software analytics spreads across the areas of system quality, user experience, and development productivity, indicating a wide scope and huge potential for impact on practice.

Despite these opportunities, putting software analytics technologies into real-world use involves significant challenges. How do you ensure the analysis output is insightful and actionable? How do you know you're using the appropriate data to answer the questions practitioners care about? How do you evaluate your analysis techniques in real-world settings?

RELATED WORK IN SOFTWARE ANALYTICS TECHNIQUES

A plethora of research exists on predicting software reliability in terms of the defect count at different levels of software systems.[1] Recent examples of using software artifacts to improve software development are software intelligence[2] and analytics for software development.[3] They both offer pertinent information to support developers' decision making.

References

1. T.L. Graves et al., "Predicting Fault Incidence Using Software Change History," *IEEE Trans. Software Eng.*, vol. 26, no. 7, 2000, pp. 653–661.
2. A.E. Hassan and T. Xie, "Software Intelligence: The Future of Mining Software Engineering Data," *Proc. FSE/SDP Workshop Future of Software Eng. Research* (FoSER 10), ACM, 2010, pp. 161–166.
3. R.P.L. Buse and T. Zimmermann, "Information Needs for Software Development Analytics," *Proc. Int'l Conf. Software Eng.* (ICSE 12), IEEE, 2012, pp. 987–996.

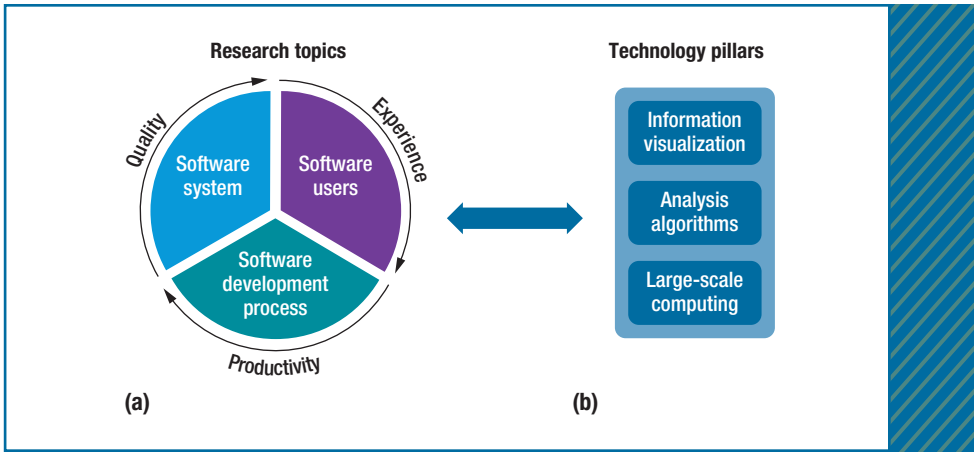

FIGURE 1. (a) The trinity of software development, systems, and users, with the ultimate goal of improving development productivity, software quality, and user experience. (b) There are three key technology pillars employed: information visualization, data analysis algorithms, and large-scale computing.

Experiences with StackMine

Performance debugging in the large has recently emerged, owing to available infrastructure support for collecting execution traces with performance issues from a huge number of users at deployment sites.[7] One example of such infrastructure support at Microsoft is PerfTrack (http://channel9.msdn.com/Blogs/Charles/Inside-Windows-7-Reliability-Performance-and-PerfTrack), which measures system

responsiveness to user actions on operating systems. Consider a user clicking on a Windows Explorer menu item to create a folder. PerfTrack measures how long it takes for the user to receive a response from the system. If the response time exceeds a predefined threshold, PerfTrack sends execution traces (containing call stacks collected during the preceding time interval) back to Microsoft for debugging. All the collected traces in total could contain more than 1 billion call stacks, which greatly exceed what performance analysts can afford to investigate.

> Performance debugging can be reduced to a software analytics problem.

Performance debugging can be reduced to a software analytics problem. Given enormous call stacks collected at the deployment sites, how can analytics help performance analysts effectively discover high-impact performance bugs (such as those causing long response delays for many users)? To tackle this problem, we developed StackMine.

Solving Essential Problems

Before the project started, a member of a Windows product team met with the Software Analytics Group. He introduced the team's tools and the state of the practice in inspecting a stream of system-event traces for Windows performance analysis. He also described the challenges the team faced in inspecting a large number of trace streams. We then took the usual research route of looking for existing research on state-of-the-art techniques for OS performance analysis. We found that analysis based on large sets of real-world trace data was an emerging topic with little published research. We found this exciting because of the potential opportunity to pursue pioneering research.

As researchers with machine learning backgrounds, we started the project in a way with which we were familiar. On the basis of our understanding of the product team's analysis records and our discussion with a Windows performance analyst, we quickly decided to study the high CPU consumption exhibited in traces. We then formulated issue detection as a classification problem and issue correlation as a clustering problem (both at the granularity of trace streams). After a few months, we got promising results from a machine learning perspective—for example, high accuracy for classification, and high purity and completeness for clustering.

We then presented the promising results to the Windows performance analysts. Aside from making a few polite comments such as "interesting" and "good," they didn't provide much encouragement for us to continue along this direction. From further feedback, we learned we weren't solving the essential problems they cared about. First, we missed the most important problem category—"wait," representing delays owing to various reasons, including synchronization, resource contention, power state, and so on. Second, automatic detection and localization of high-CPU-consumption intervals didn't help the analysts solve the real problems. Tasks such as identifying the region of interest (ROI) should and could be handled well by instrumentation together with reliable domain-specific heuristics. Third, the trace stream wasn't the right granularity level for similarity modeling and clustering; it was too high a level to help performance analysts locate root causes of issues.

So, we refocused the project on identifying essential problems. We worked closely with the performance analysts on this throughout the rest of the project life cycle. Ultimately, we used StackMine to identify high-impact program-execution patterns from a large number of trace streams. We selected call stack patterns—sequences of function calls that happen during program execution (abbreviated as *stack patterns* for this article)—as the pattern representation and formulated the pattern discovery problem as a mining and clustering problem against hundreds of millions of call stacks.

The Need for Domain Knowledge

Deep domain knowledge is embedded in trace streams, in which more than 400 event types can appear. We couldn't afford to study all of them, and we didn't need to. We started with more than 10 key event types. After we presented our initial algorithms based on these event types, the analysts pointed out the missing event types that could compromise our algorithms' correctness. We learned those event types and revised our algorithms accordingly. Overall, learning the domain knowledge underlying these event definitions took more than a week.

Domain knowledge can often help scope relevant data for study, especially with large-scale data. Given the time interval of a trace stream, we first mined and clustered stack patterns against all the CPU-sampling and thread-context-switch events. The computation was expensive and the result wasn't satisfactory. To address these issues, we designed appropriate ROI-extraction

algorithms to properly scope the analysis and reduce computation cost.[7] We wouldn't have come up with such algorithms without observing the analysts' existing practices to more fully understand how the data relates to the targeted problem.

Removing noise in data is important, because noisy data can compromise an algorithm's overall effectiveness, regardless of how you improve the algorithm. One example from StackMine was that noisy data remarkably disturbed candidate algorithms' evaluation results. Such noisy data could mislead the algorithm selection or, even worse, mislead the entire project direction.

In StackMine, we used coverage of performance bottlenecks to measure identified patterns' effectiveness. We defined the coverage as the ratio of pattern-affected time periods to the total time periods of all ROIs. The higher the coverage, the higher the probability of discovering real high-impact bugs.

Our first coverage came back with extremely low results, regardless of how we improved the algorithms to discover more patterns. We found that a number of abnormally long trace streams negatively affected the coverage results. We verified that these streams were from computers that had experienced long periods of sleep. That is, the streams started before a long sleep and ended after the computer woke up. After filtering out these streams, we got reasonable coverage results.

Iterative Feedback

R&D of analytics technologies is usually an iterative process in which timely feedback from the target audience is important. We certainly benefited from the feedback loop we constructed when the project started.

One of the feedback loop's benefits involved defining the similarity model for clustering stack patterns, sequences of function calls shared by a set of call stacks. We used these patterns to represent performance issues. Typically, a performance issue can exhibit multiple stack patterns that share common major parts of bottlenecks and vary in minor parts. So, to better prioritize performance bottlenecks, these patterns should be clustered based on the similarity of the stack patterns. The generic sequence-edit-distance model was a natural choice for calculating the similarity of a pair of stack patterns.

However, some of the resulting clusters were huge and included many performance bottlenecks. Furthermore, some highly important performance bottlenecks weren't clustered and were distributed across many small clusters. In reviewing the results with the analysts, we quickly realized that the similarity model without appropriate domain knowledge couldn't reflect the true relationships of stack patterns.

After that, we redesigned the similarity model to incorporate essential domain knowledge. After a few iterations of experiments and discussion, we obtained an appropriate similarity model with satisfactory clustering results.

Another of the feedback loop's benefits involved selecting the evaluation criteria for StackMine. When we presented the preliminary results (a ranked list of stack pattern clusters) to the analysts, we used purity and completeness metrics as the evaluation criteria, which are common. However, the analysts couldn't easily relate these metrics to their practice; they cared about the coverage of performance bottlenecks provided by the top pattern clusters against all performance bottlenecks in stack traces in terms of total wait time from the user perspective. We learned that their primary interest was to quickly achieve high coverage against all performance bottlenecks in a given trace stream set. High coverage would indicate that stack patterns from top clusters could identify and explain most of the performance bottlenecks captured in the trace stream set. On the basis of this feedback, we designed the performance-bottleneck-coverage metric to measure the stack pattern clusters.[7]

A third benefit involved using StackMine to simultaneously help analysts and improve algorithms. When analyzing a new dataset, the analysts first used StackMine to obtain the worst performance bottlenecks from the top clusters. They quickly went through the top clusters to verify the bottlenecks and provided feedback. They then used their traditional workflow to study the trace streams' not-covered parts. By leveraging the analyzed datasets and feedback as partial labels, we improved our algorithms. In this way, we weren't blocked or limited by a lack of labeled data. Meanwhile, the practitioners could benefit from the improved solutions. This interactive, iterative model helped the StackMine project make steady progress.

> By leveraging the analyzed datasets and feedback as partial labels, we improved our algorithms.

Making an Impact

StackMine resulted from two years' continuous development and improvement. In December 2010, a Microsoft

team first applied StackMine in performance debugging. Since then, the team's performance analysts have used it to analyze hundreds of millions of call stacks. The team estimated that StackMine has reduced human investigation effort by 90 percent. They stated,

> *We believe that the StackMine tool is highly valuable and much more efficient for mass trace streams [more than 100 trace streams] analysis. For 1,000 trace streams, we believe the tool saves us four to six weeks to create new performance signatures [representations of performance issues], which is quite a significant productivity boost.*

Lessons Learned

Ultimately, we learned five main lessons from StackMine.

Identify Essential Problems

It's often easy to grab some datasets, apply certain data analysis techniques, and make some observations. However, these observations, even with good evaluation results from a data analysis perspective, might not help practitioners accomplish their tasks. We made such a mistake at the beginning of StackMine. It's important to first identify essential problems for accomplishing the task and then obtain the right datasets to help solve the problems. These problems are those whose solution would substantially improve the overall effectiveness of the task, such as improving practitioner productivity, software quality, or user experience.

As the StackMine project demonstrated, one way to identify essential problems is through close collaboration between researchers and practitioners, interactively and iteratively. Sometimes, practitioners might not be able to articulate essential problems when they're emerged in various challenges from daily work. Researchers should work in an agile way to construct a quick feedback loop to identify essential problems early. Such loops will help avoid detours or wasting time during a software analytics project.

Understand the Domain Semantics

Software artifacts often carry semantics specific to a software domain. Understanding artifacts' semantics is a prerequisite for data analysis. The StackMine project might be an extreme example of this; there was a steep learning curve for us to understand the performance traces before we could conduct any analysis.

In practice, understanding data is threefold: data interpretation, data selection, and data filtering. To interpret data, researchers must understand basic definitions of domain-specific terminologies and concepts. To select data, researchers must understand the connections between it and the problem being solved. To filter data, researchers must understand its defects and limitations to avoid incorrect inferences.

Data interpretation. Researchers need essential domain knowledge about data, including domain-specific terms, concepts, and principles. Typically, researchers don't have such knowledge at the beginning of a software analytics project; rather, they usually ask practitioners for such information. The knowledge is often scattered among practitioners or undocumented, and few practitioners will instinctively know which portion of the knowledge will be important or not for a given problem. So, on the basis of our experience, we suggest that the learning or transferring of data knowledge should be driven by demand. It could be driven by either researchers or practitioners, interactively and iteratively.

Data selection. Some data will be irrelevant to solving a given problem. Researchers often must select an appropriate subset of data to conduct effective analysis. Besides knowledge about data, practitioners' experiences and skills can help researchers scope the data. Too large a scope might incur high computation cost and introduce noise; too small a scope might miss important information.

Data filtering. Data might contain defects that could lead to incorrect analysis results. Example defects are data points with abnormal values that might indicate untrue situations. Detecting such defects might be difficult. Researchers should review analysis results and filter data to eliminate or reduce the defects' impact.

> Creating software analytics solutions for real-world problems is an iterative process.

Create Feedback Loops Early, with Many Iterations

Creating software analytics solutions for real-world problems is an iterative process. It's much more effective to start a feedback loop with software practitioners early on, rather than later. The feedback is often valuable for formulating research problems and researching appropriate analysis algorithms. In addition, software analytics projects can benefit from early feedback in terms of

building trust between researchers and practitioners, as well as evaluating results in real-world settings.

To provide end-to-end solutions, software analytics projects typically produce an analysis system, such as StackMine, with researchers as solution providers and practitioners as end users. Such a system should be both effective and easy to use. If it's not effective—that is, if it cannot help practitioners solve their problem—they won't use it. If it's difficult to use, they probably won't adopt it.

Employ Scalability and Customization

Owing to the scale of data in real-world settings, real-world problems often require scalable analytics solutions. Scalability can directly affect the analysis algorithms used to solve the problems.

Owing to variations in software and services, another common requirement is customization that incorporates domain knowledge. To be effective, such customization involves

- filtering noisy and irrelevant data,
- specifying the intrinsic relationships between data points that cannot be derived from the data itself, and
- providing empirical and heuristic guidance to make the algorithms robust against biased data.

This customization typically should be conducted iteratively through close collaboration between researchers and practitioners.

The StackMine project demonstrated the importance of scalability and customization. Without the distributed computing system, it couldn't have handled the scale of call stacks that the Windows team analyzed. Without customization, it wouldn't have been able to incorporate the analysts' knowledge.

ABOUT THE AUTHORS

DONGMEI ZHANG is a principal researcher at Microsoft Research Asia and the founder and manager of its Software Analytics Group. Her research interests include data-driven software analysis, machine learning, information visualization, and large-scale computing platforms. Zhang received a PhD in robotics from Carnegie Mellon University. Contact her at dongmeiz@microsoft.com.

SHI HAN is a researcher in Microsoft Research Asia's Software Analytics Group. His research interests include software analytics, particularly software performance analysis by leveraging machine learning and data mining. Han received an MS in computer science from Zhejiang University. Contact him at shihan@microsoft.com.

YINGNONG DANG is a lead researcher in Microsoft Research Asia's Software Analytics Group. His research interests include software engineering, software analytics, data analytics, and human-computer interaction. Dang received a PhD in control science and engineering from Xi'an Jiaotong University. He's a member of ACM and the China Computer Federation's Software Engineering Technical Committee. Contact him at yidang@microsoft.com.

JIAN-GUANG LOU is a lead researcher in Microsoft Research Asia's Software Analytics Group. His research interests include performance analysis and diagnosis of online services as well as data mining for software engineering. Lou received a PhD in pattern recognition from the Chinese Academy of Sciences Institute of Automation. He's a member of IEEE and ACM. Contact him at jlou@microsoft.com.

HAIDONG ZHANG is a principal software architect in Microsoft Research Asia's Software Analytics Group. His research interests include software analytics and information visualization. Zhang received a PhD in computer science from Peking University. Contact him at haizhang@microsoft.com.

TAO XIE is an associate professor in the Department of Computer Science at the University of Illinois at Urbana-Champaign. His research interests include software testing, program analysis, and software analytics. Xie received a PhD in computer science from the University of Washington. He's a senior member of IEEE and ACM. Contact him at taoxie@illinois.edu.

Correlate Evaluation Criteria with Real Tasks in Practice

Because of the natural connection with software development practice, software analytics projects should be (at least partly) evaluated using the real tasks for which they're targeted. Researchers can use common evaluation criteria for data analysis, such as precision and recall, to measure intermediate results. However, these often aren't the most appropriate criteria when real tasks are involved. For example, in StackMine, we used the coverage of detected performance bottlenecks, which was directly related to the Windows analysts' tasks. When conducting real-world evaluations with practitioners, researchers must be aware of both the benefits and costs to practitioners.

In particular, a typical evaluation of the solution has two aspects. First, what do you evaluate? The evaluation criteria could measure the solution's effectiveness or efficiency, or compare it to alternatives. Second, how do you evaluate? That is, what activities do you perform to obtain the evaluation criteria's metric values? For software analytics tasks, you must select evaluation criteria based on practical needs and design evaluation activities that don't distract practitioners from their normal work.

Practical evaluation criteria are important for measuring a software analytics solution's ultimate effectiveness and efficiency, particularly in helping practitioners with their daily work. These practical criteria could differ considerably from the traditional evaluation criteria of classic machine learning or data mining tasks. The latter are typically used for measuring the effectiveness of intermediate steps (for example, accuracy for classification, precision and recall for information retrieval, and purity and completeness for clustering). In contrast, practical evaluation criteria typically are based on practitioners' daily practices.

Evaluation criteria typically comprise a set of metrics calculated by corresponding objective functions. Researchers devise optimal solutions by optimizing those functions and comparing alternative solutions. Among a set of objective functions, a priority might exist; the highest-priority function should relate to the top interest—the practitioners' biggest concern.

These valuable lessons learned from the StackMine project help increase researchers' and practitioners' awareness of the issues and practices of putting software analytics

technologies to real-world use. In addition to the StackMine project, Microsoft Research Asia's Software Analytics Group has conducted a number of software analytics projects (such as XIAO[8] and Service Analysis Studio[9,10,13]) with a high impact on software development practice. These projects both share commonalities and have their own unique characteristics in terms of offering lessons learned for the community, calling for our future efforts in summarizing and analyzing such lessons learned across various projects.

Acknowledgments

We thank the engineers from the Microsoft product teams for collaborating with Microsoft Research Asia's Software Analytics Group on StackMine and other projects.

References

1. J. Czerwonka et al., "CRANE: Failure Prediction, Change Analysis and Test Prioritization in Practice—Experiences from Windows," *Proc. IEEE 4th Int'l Conf. Software Testing, Verification and Validation* (ICST 11), IEEE CS, 2011, pp. 357–366.
2. E. Shihab et al., "An Industrial Study on the Risk of Software Changes," *Proc. ACM SIGSOFT 20th Int'l Symp. Foundations Software Eng.* (FSE 12), ACM, 2012, article 62.
3. K. Glerum et al., "Debugging in the (Very) Large: Ten Years of Implementation and Experience," *Proc. ACM SIGOPS 22nd Symp. Operating Systems Principles* (SOSP 09), ACM, 2009, pp. 103–116.
4. T. Gorschek et al., "A Model for Technology Transfer in Practice," *IEEE Software*, vol. 23, no. 6, 2006, pp. 88–95.
5. A. Sandberg, L. Pareto, and T. Arts, "Agile Collaborative Research: Action Principles for Industry-Academia Collaboration," *IEEE Software*, vol. 28, no. 4, 2011, pp. 74–83.
6. C. Wohlin et al., "The Success Factors Powering Industry-Academia Collaboration," *IEEE Software*, vol. 29, no. 2, 2012, pp. 67–73.
7. S. Han et al., "Performance Debugging in the Large via Mining Millions of Stack Traces," *Proc. Int'l Conf. Software Eng.* (ICSE 12), IEEE, 2012, pp. 145–155.
8. Y. Dang et al., "XIAO: Tuning Code Clones at Hands of Engineers in Practice," *Proc. 28th Ann. Computer Security Applications Conf.* (ACSAC 12), ACM, 2012, pp. 369–378.
9. R. Ding et al., "Healing Online Service Systems via Mining Historical Issue Repositories," *Proc. 27th IEEE/ACM Int'l Conf. Automated Software Eng.* (ASE 12), ACM, 2012, pp. 318–321.
10. Q. Fu et al., "Performance Issue Diagnosis for Online Service Systems," *Proc. 31st IEEE Symp. Reliable Distributed Systems* (SRDS 12), IEEE CS, 2012, pp. 273–278.
11. D. Zhang et al., "Software Analytics as a Learning Case in Practice: Approaches and Experiences," *Proc. Int'l Workshop Machine Learning Technologies in Software Eng.* (MALETS 11), ACM, 2011, pp. 55–58.
12. D. Zhang and T. Xie, "Software Analytics in Practice: Mini Tutorial," *Proc. Int'l Conf. Software Eng.* (ICSE 12), IEEE, 2012, p. 997.
13. J.-G. Lou et al., "Software Analytics for Incident Management of Online Services: An Experience Report," to appear in *Proc. Int'l Conf. Automated Software Eng.* (ASE 13), IEEE, 2013.