Editor: **Christof Ebert**
Vector Consulting Services
christof.ebert@vector.com

Data Science: Technologies for Better Software

Christof Ebert, Jens Heidrich, Silverio Martínez-Fernández, and Adam Trendowicz

From the Editor

Data science is rapidly gaining relevance among software teams. Data science methods facilitate analyzing and predicting different quality aspects of the software product, process, and use by combining results from different data sources, such as versioning systems, issue-tracking systems, and static code analysis, in an efficient way. Authors Jens Heidrich, Silverio Martínez-Fernández, Adam Trendowicz, and I provide industry insights about the use of data science for improving software development. I look forward to hearing from both readers and prospective column authors about this column and the technologies you want to know more about. —*Christof Ebert*

DATA SCIENCE IS mandatory in today's business to capitalize on achievements and assets. This specifically holds for modern software development, where data science facilitates analyzing product, process, and usage and thus managing evolution and performance. With the convergence of embedded and IT domains, such as the Internet of Things (IoT) and automotive systems, software systems are becoming more complex. Complexity has two faces. On one hand it means more functionality and fluid delivery models, thus offering markets more value, such as the ability to deliver a single-customer focus. Complexity, however, also means the growth of technical debt, which slows productivity and lowers quality. As software engineering generates ever larger and more varied data sets, such as feature usage, code analysis, test coverage, error logs, and maintenance data, companies face the challenge of unlocking the value of that data.

Those failing to effectively apply data science in their software-engineering processes are increasingly putting themselves at a competitive disadvantage. Classical software measurement has played a key role in managing the complexity of software processes and products in recent decades but is reaching its limits because of the ever-increasing amount of available data and their dependencies.

With embedded software becoming increasingly pervasive and critical to our society, engineers must ensure that their software performs as intended and doesn't fail. Delivering defective software has many consequences. It damages prestige. It wastes money, time, or both. It makes customers unhappy. In the case of real-time software, it can also affect the health and integrity of human beings. Software defects can have dramatic consequences specifically with respect to safety and cybersecurity. So, the

Digital Object Identifier 10.1109/MS.2019.2933681
Date of current version: 22 October 2019

manufacturers of safety-critical products have long recognized the need to deliver software that is secure, reliable, and of the highest quality.

All software systems across domains and applications exhibit one clear trend: complexity, like cancer, grows. To stay in control, we must manage essential complexity and reduce accidental complexity. Software measurement has played a key role for managing the complexity of software processes and products in the last decades.[1] Nowadays, short development cycles as introduced by recent development paradigms, such as agile development or DevOps, expand opportunities to get vast amounts of data in a relatively short time, learn from that data during the project, and transfer this knowledge across projects.

Today, software measurement is evolving toward data science. It deals with big data and intelligent algorithms, such as machine learning and deep learning.[1,2] Even though more powerful (open source as well as commercial) tools and infrastructures are available, a fool with a tool is still a fool. So, modern software measurement requires profound knowledge about aspects of data science, including an understanding of which methods and techniques to use in which situation and which steps to follow for collecting, analyzing, and evaluating data.

This article demonstrates how software engineers benefit from data science approaches based on two practical cases, namely

- data-driven quality management
- automatic postprocessing of code analysis.

Essential Steps for Data Mining

Data science and big data analytics are methods to systematically collect, process, and analyze very large amounts of data. Conway[3] states that data science is a cross-domain discipline that requires in-depth basic computer science and mathematics skills as well as expertise in the application domain. However, in practice the term *data science* is often used as a buzzword addressing a variety of fields of expertise in different depths. One central consensus is that it is about expertise in dealing with big data (i.e., an increased volume of data, increased variety of data, and increased velocity for processing data). One of the fullest considerations of data science and related competences can be found in the Edison Data Science Framework (EDSF). In the EDSF, a data scientist refers to a user who has enough knowledge and expertise in the areas of business needs, domain knowledge, analytical skills, programming, and systems engineering to continuously advance the scientific process across all stages of the big-data lifecycle to deliver an expected scientific or business benefit to an organization or project.[4]

One critical competence in data analytics is the ability to build and reliably evaluate models. In this context, a de facto standard is the Cross-Industry Standard Process for Data Mining (CRISP-DM).[5] It describes six steps that are essential for data mining and is widely used in research and industry:

1. *Business understanding:* Understand the business objectives of the organization, define analysis goals, and set up a project plan.
2. *Data understanding:* Collect, characterize, und understand the data and their meaning and analyze the quality of available data.

> Those failing to effectively apply data science in their software-engineering processes are increasingly putting themselves at a competitive disadvantage.

3. *Data preparation:* Select the data to analyze; clean and format the data.
4. *Modeling:* Choose the right technique for model building, create the model, and analyze the quality of the model.
5. *Evaluation:* Evaluate and review the results and determine follow-up steps.
6. *Deployment:* Plan how to deploy and maintain the model in practice; document the outcomes and lessons learned.

These steps are performed iteratively, and you may go back and forth between different steps. These six steps form the basis for describing our activities performed in the two cases that follow.

Case 1: Data-Driven Quality Management

Q-Rapids, a research and innovation project of the European Union (Grant 732253), aims to create a way

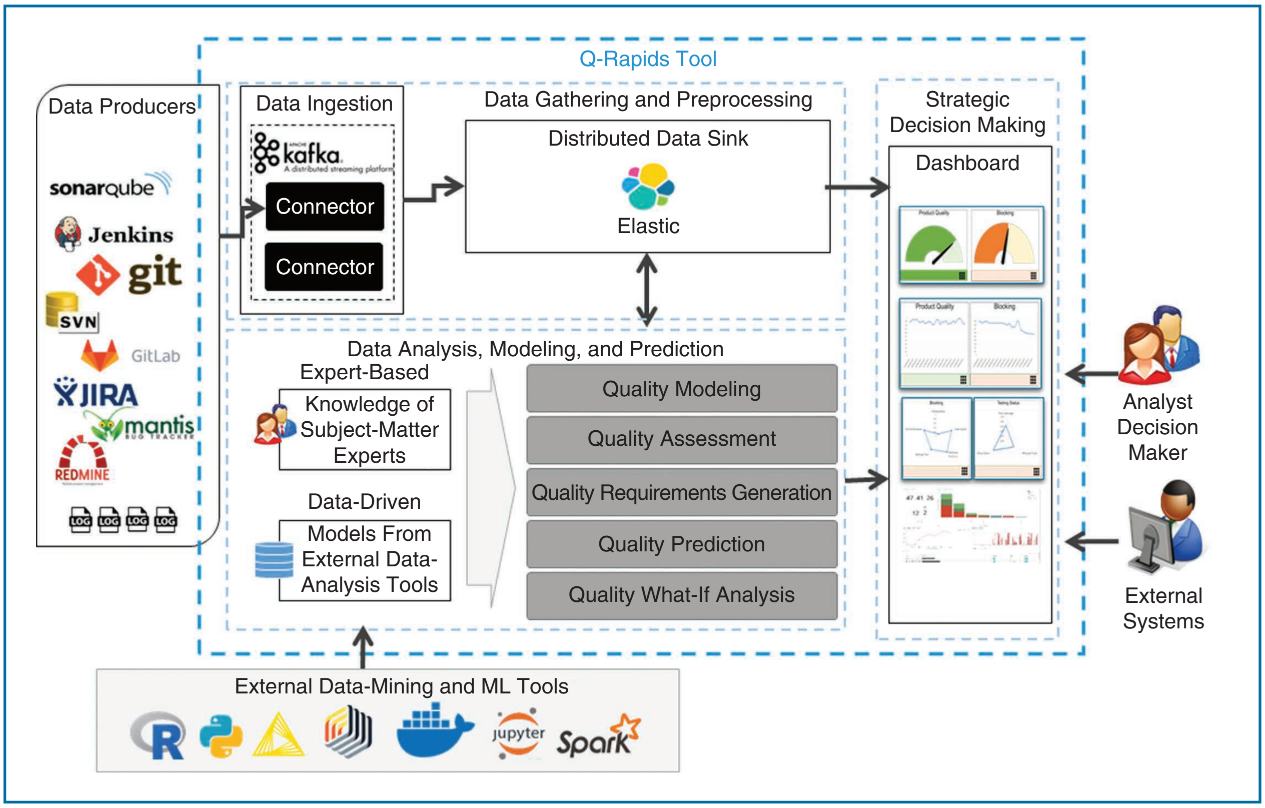


FIGURE 1. A chart showing the Q-Rapids framework architecture.

to exploit software product, process, and usage data with data science to improve software quality.[6] The Q-Rapids architecture (Figure 1) provides scalable data ingestion from heterogeneous data sources, interfaces with data-mining frameworks to derive a data-driven quality model, and evidence-based recommendations to decision makers. These recommendations can include, for example, an alert about identified or predicted external quality problems and proposals related to quality requirements.

Let's focus on an example of how the Q-Rapids framework helped improve external quality management for a software platform providing several digital services that focused on rural areas and included, for example, online shopping, a news portal, and car sharing.

- *Business understanding*: There was much potential to improve the software by understanding runtime data. The goals were to identify within the development time of the platform the key aspects that influence software quality at runtime and to predict the expected number of runtime issues based on that. The business goal was to reduce the time and effort expended exploiting runtime data and its history.
- *Data understanding*: To obtain these goals, we had external data (e.g., access/error logs, application crashes, directly received bugs from hotlines or emails) and internal data (e.g., code quality metrics, quality rule violations, commits) available. Based on the Q-Rapids tooling, we used Apache connectors (Figure 1) to ingest data in real time. We ended up creating a centralized repository with heterogeneous data, including usage data (e.g., crashes and logs from mobile applications). Then, we made an analysis on relevant features of our data as not all collected data were of interest.[7]
- *Data preparation*: We integrated the data stored in different source systems and structures, handled data-quality deficits (e.g., incomplete, inconsistent, and incorrect data), and transformed data into the format acceptable for specific methods and tools used during the analysis. This has been the most time-consuming step. Software data are created automatically,

but not for the purpose of being readily analyzed.

- *Modeling*: Our goal was to find a fitting model for software reliability. We applied two alternative ensemble methods: random forest and extreme gradient boosting. With these methods, we investigated the probability of software bugs related to runtime issues in association with structural properties of software code as well as the amount and type of software changes along the software's evolution. The analysis provided two outcomes: relevancy of individual metrics as predictors of external quality and a quality model that captures quantitatively dependencies between the most relevant metrics and the external quality.
- *Evaluation*: A preliminary evaluation showed the ability to show an overview of the system crashes and exceptions over time and predict them based on their historical relationships with internal quality metrics.
- *Deployment*: The developed models were integrated into the Q-Rapids tool for guiding software developers regarding the predicted evolution of external quality or to give an indication of the most relevant factors influencing it.

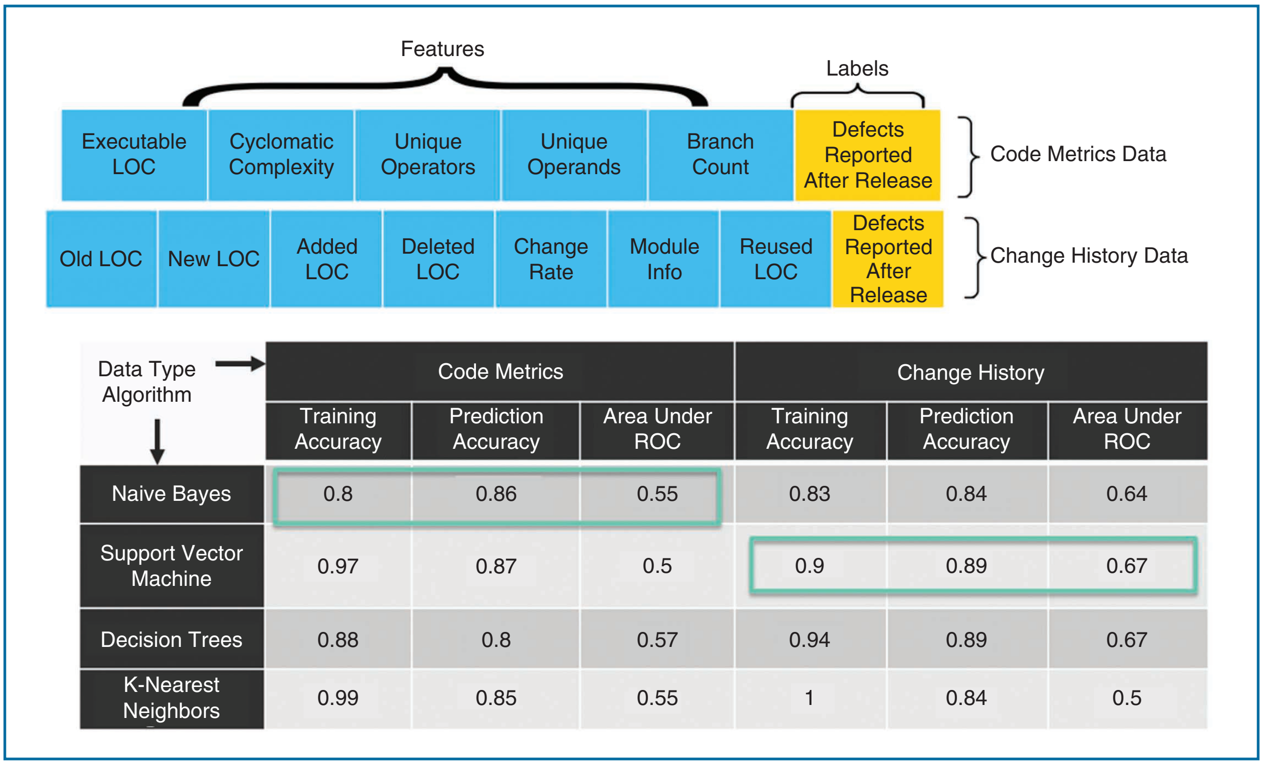
Case 2: Data-Driven Prioritization of Code Findings

Companies increasingly use code analysis to improve quality. Often static analysis is incorporated in techniques using Jenkins pipelines in continuous build to achieve strong entry checks before further integration of code changes. For safety-critical software, static analysis is even mandatory. However, static analysis tools are far from easy to use and provide quite complex results. In fact, it is almost impossible to use the results without guidance of experts. Designers are overwhelmed with the many tool reports, of which some 90% are irrelevant. The problem of too many false positives and the heterogeneous strengths of various tools demand the use of several tools in parallel. After few trials, most companies stop using such tools due to their inherent complexity. At Vector Consulting we have over many years built a competence for code-quality analysis where we deploy multiple analysis tools and automatically postprocess the data for the benefit of our clients.

- *Business understanding*: Detecting code defects in the early stages of software development not only decreases the costs of the software but also increases quality by decreasing the technical debt and the propagation of errors to other phases.[1] With 20–30% of all software containing 40–60% of all defects, static code analysis has been shown to reduce postrelease failures by more than 50% and thus cut development cost by 20–30%.[1] It doesn't replace other verification and validation steps but helps remove certain defect types that otherwise wouldn't be found. However, the warning reports by different static analysis tools do not present the defects in the order of priorities according to the criticality of software modules but according to predefined hard-coded defect priorities.

The basic idea of the case is to use a ranking approach that considers the criticality of software modules to rank the defects shown to the developer. This is essential because often the project managers need to decide about where to focus the quality-assurance resources to improve the software product.

- *Data understanding*: Typically, plenty of data are available to train a classifier. These data include, for example, 1) code-complexity metrics, such as cyclomatic complexity; 2) the change history of modules, such as changes in the modification rate and additions/deletions in modules; 3) code reuse due to imports between modules; and 4) metrics that capture the software development process.
- *Data preparation*: To give a proof of concept, code metrics and change history were used to train separate predictive models. Initially the data of real-time projects regarding code complexity and change history were used from the Promise repository[8] for training classifiers.
- *Modeling*: The proposed method makes use of criticality prediction and defect density in a module to rank them in the order of criticality from highest to lowest. The criticality prediction is done by training an ML algorithm to generate a criticality classifier. The features for training were selected from the data based on the metrics generated by the tools used for evaluation. The upper part of Figure 2 shows the features and labels used from the data for training. Using the data sets, two different classifiers were trained using four different ML algorithms to measure the performance and select the best algorithm that suits a particular type of data.
- *Evaluation*: The performance metrics used included training accuracy, prediction accuracy, and area under the receiver operating characteristics (ROC) curve. The lower part of Figure 2 shows the results and the best algorithm for each data set.


Features
Labels
Executable LOC
Cyclomatic Complexity
Unique Operators
Unique Operands
Branch Count
Defects Reported After Release
Code Metrics Data
Old LOC
New LOC
Added LOC
Deleted LOC
Change Rate
Module Info
Reused LOC
Defects Reported After Release
Change History Data

Data Type → / Algorithm ↓	Code Metrics			Change History		
	Training Accuracy	Prediction Accuracy	Area Under ROC	Training Accuracy	Prediction Accuracy	Area Under ROC
Naive Bayes	0.8	0.86	0.55	0.83	0.84	0.64
Support Vector Machine	0.97	0.87	0.5	0.9	0.89	0.67
Decision Trees	0.88	0.8	0.57	0.94	0.89	0.67
K-Nearest Neighbors	0.99	0.85	0.55	1	0.84	0.5

FIGURE 2. The features and labels for training data and performance of criticality classifiers. LOC: lines of code.

- *Deployment*: After training the criticality classifier by choosing the appropriate algorithm for the data set, the next phase was to generate module rankings using these predictions. Since there were two predictive models with binary output predictions (critical or noncritical), this makes four possible prediction combinations, which were assigned a criticality factor from 1 to 4 (most critical to least critical). The classifier that has better prediction accuracy and area under the ROC curve gets assigned a higher criticality factor. The ranking algorithm starts first by extracting the code metrics from tool reports and change histories from the version control system for each module. Each of these feature vectors for code metrics and change history is fed to the relevant classifiers to get predictions. The prediction combination for each module is assigned a criticality factor according the criteria described first. The modules are then sorted according to their criticality factors.

This article demonstrates how software engineers benefit from data science approaches. Table 1 illustrates some lessons learned from these two applications sorted by the CRISP-DM steps we took. We have seen in two cases that data science approaches clearly provide additional value compared to classical software measurement. However, some aspects require special care: data science cannot help in building better software unless you first understand your business and your data and prepare the data accordingly.

Data science in software engineering is not about applying ML/AI algorithms on cleaned data sets. You must invest on the initial steps of business and data understanding (i.e., establish a clear objective) and data preparation (i.e., obtain relevant and high-quality data) for delivering business value. Without these steps, there is no guarantee of finding valuable insights. Yes, tons of data in software engineering are available, but they were rarely generated as a foundation for building analysis and prediction models.

Using data science for software engineering demands competences that go beyond software engineering skills. Training software developers won't help, as such use of data science demands

Table 1. A summary of lessons learned.

Data science step	Lessons learned
Business understanding	• Defining the motivation for collecting data and building models is essential for not getting lost in the later steps. One should think that after more than three decades after introducing goal–question–metric and related approaches for goal-oriented measurement, this should be common sense. However, our experience says that especially with the big data hype, there is a risk of collecting all of the data that one can get and hope to discover miracles later. Unfortunately, these miracles seldomly happen. • It is essential to keep in mind the associated potential of data science and your capabilities. Data science comes in different flavors and specializations. It is important to get a clear picture of which area to build up or what competences to buy in for capitalizing the intended benefits.
Data understanding	• Data availability and their quality are among the biggest issues. Even though a lot of data in software development can be collected automatically (such as static code analysis tools), many data still need to be collected manually (such as defect-tracking data), some data are likely to be missing (such as effort data), and some data may not be suitable to use with other data (such as manually collected process metrics). • Collecting your own high-quality data is laborious but offers the best chance to get the most benefits from the data later. It is essential to invest into collecting the data you need according to your goals, which is not necessarily equal to the data you can easily get. • Data analysis starts already in this phase, because understanding large amounts of data requires an analytical approach. • It is essential to understand the meaning of the data and not just their format. This requires the combination of data science competences with domain-related competencies (e.g., from a developer, tester, or project manager).
Data preparation	• The process of cleaning and correcting messed up data is very expensive. Artificial intelligence (AI)/ML could help improve data cleansing in the future, i.e., learning data deficits and applying this knowledge when curating the data. • Correcting data creation/collection processes means starting data collection from scratch. This approach makes it easy to focus on data quality in the first place, but it takes a long time to gather enough data for model building. • The application of hybrid approaches is needed for starting small and iteratively expanding. This also requires vigorous interaction between data preparation, modeling, and evaluation.
Modeling	• Thanks to the big data and AI hype, several tools for model building and data analytics are available, including a whole stack of free and open-source tools. However, a fool with a tool is still a fool. So, organizations must build up data science-related competences for knowing which analysis technique and modeling approach to use on what type of data and for building meaningful models. • Horizontal scaling supports the "start small and expand" approach so that small and medium-size enterprises also have an opportunity to take advantage of data science for software.
Evaluation	• Evaluation should make use of the Plan, Do, Check, Act principle such as quality improvement paradigm. • All steps up to evaluation make up an iterative process of model development. When setting up the goals related to building a model, it is essential to be clear and honest about success criteria (e.g., a certain accuracy or power of the model). Like agile development, it oftentimes makes sense to strive for short iterations and increments when building a model instead of spending endless time trying to understand the data and making preparations only to discover that the analysis technique or modeling approach chosen does not work or deliver suitable results.
Deployment	• Starting small, developing incrementally, keeping cycles short, and focusing on minimal viable products are best practices for putting your data science methods to work. • People putting the methods into in practices are not likely to be data scientists and will therefore need extra support. • You may have a model that works perfectly on test data but fails in practice. This can occur for any number of reasons, such as overfitting or because the application context of the model is changing. When deploying such models, you should take special care to maintain the model. This can be done by, for example, regularly evaluating the quality of model decisions and relearning the model if its performance falls below a certain threshold.

ABOUT THE AUTHORS

CHRISTOF EBERT is the managing director of Vector Consulting Services. He is on the *IEEE Software* editorial board and a professor at the University of Stuttgart, Germany, and the Sorbonne in Paris. Contact him at christof.ebert@vector.com.

SILVERIO MARTÍNEZ-FERNÁNDEZ is a researcher at the Fraunhofer Institute for Experimental Software Engineering IESE, Kaiserslautern, Germany. Contact him at silverio.martinez@iese.fraunhofer.de.

JENS HEIDRICH is division manager for process management at the Fraunhofer Institute for Experimental Software Engineering IESE, Kaiserslautern, Germany. Contact him at jens.heidrich@iese.fraunhofer.de.

ADAM TRENDOWICZ is a senior consultant in data engineering at the Fraunhofer Institute for Experimental Software Engineering IESE, Kaiserslautern, Germany. Contact him at adam.trendowicz@iese.fraunhofer.de.

skills way beyond those needed for software development, such as skills in statistics and algorithm design. Instead we recommend that software decision makers work with data scientists and consultants with the necessary skills. They would collaborate in such tasks as the following:

- identifying use cases and the benefits of data science (and big data, AI) in the context of software and system engineering (e.g., for improving quality or managing risk)
- acquiring and storing system development and runtime metrics and data quality analyses and curating data (such as software repositories or log files)
- analyzing measurement data, modeling, the application of AI-based techniques (e.g., ML, neural networks, or deep learning)
- creating tools for collecting, storing, analyzing, and visualizing big data and building AI models
- developing guidelines for data ethics and culture, data exchange, data security, and model evaluation.

Nobel Prize Laureate Herbert Simon once remarked that "a wealth of information creates a poverty of attention." He coined the initial approach to data science as a method to focus on what really matters and thus focus our attention on the stories and essence behind the data. Let us follow his advice also in software engineering. Rather than add quantity, and thus get entangled in ever more complex software, let us abstract and analyze with the goal of better quality.

References

1. C. Ebert and R. Dumke, *Software Measurement*. Springer, Heidelberg, Germany, 2007.
2. J. Heidrich, A. Trendowicz, and C. Ebert, "Exploiting big data's benefits," *IEEE Softw.*, vol. 33, no. 4, pp. 111–116, July 2016.
3. D. Conway, "The data science Venn diagram," Drew Conway Data Consulting, 2015. [Online]. Available: http://drewconway.com/zia/2013/3/26/the-data-science-venn-diagram
4. GitHub, "EDISONCommunity/EDSF," 2019. [Online]. Available: https://github.com/EDISONcommunity/EDSF
5. P. Chapman et al. "CRISP-DM 1.0: Step-by-step data mining guide," SPSS, 2000. [Online]. Available: https://the-modeling-agency.com/crisp-dm.pdf.
6. S. Martínez-Fernández et al. "Continuously assessing and improving software quality with software analytics tools: A case study," *IEEE Access*, vol. 7, pp. 68219–68239, 2019.
7. A. Aghabayli, D. Pfahl, S. Martínez-Fernández, and A. Trendowicz, "Integrating runtime data with development data to monitor external quality: Challenges from practice," in *Proc. 2nd Int. Workshop Software Qualities and Their Dependencies*, 2019.
8. University of Ottawa, "Promise software engineering repository." Accessed on: July 31, 2019. [Online]. Available: http://promise.site.uottawa.ca/SERepository/