Applying Data Science to Software Engineering: A summary on how data science helps software development

Shrenuj Gandhi
Computer Science
North Carolina State University
sgandhi4@ncsu.edu

Neng Jiang
Computer Science
North Carolina State University
njiang@ncsu.edu

***Abstract*— Data Science can be widely applied in Software Engineering. We take the paper "Information Needs for Software Development Analysis[1]" as our baseline study. We are reviewing the relative papers to get a thorough understanding of the methodology and techniques in how data science helps software development.**

***Index Terms*— Data Science, Defect, Statistical Analysis**

I. INTRODUCTION

Data is everywhere and with the computing power increased in the past few years, data science and analytics have become important topics of investigation. source [May, T. The New Know: Innovation Powered by Analytics. Wiley, 2009.]

Amit Valia, in his talk, mentions that we in the Data 2.0 world where we perform data warehousing and analytics on the data to extract value from it. "The world we live in, is the world of data. 25-30 years ago, we were in Data 1.0 world, where the focus was on automating business and development processes via applications. Data was just the output of those applications and there was nothing strategic about it. A few years ago we phased into Data 2.0 world, where data warehousing and analytics came along and we now extract value from that data." source [https://www.youtube.com/watch?v=EIyxdG91XBM]

SE is all data rich activities. Aspects of development generating data. decisions are relied on data. Analytics is especially useful for helping users move from only answering questions of information like What happened? to also answering questions of insight like How did it happen and why? While there has been significant research into information needs of developers [6][7][8], the needs of managers, those who make the important decisions about the future of projects, are not well understood.[Buse, R.P.L. and Zimmermann, T. Information needs for software development analytics. In ICSE '12: Proceedings of 34th International Conference on Software Engineering (2012), 987-996.]

The rest of the paper is organized as follows. Section II discusses related work which summarizes research done in this area during the period 2008 - 2012. Section III describes the baseline study and results conducted in the paper titled "Information Needs for Software Development Analysis". source [Buse, R.P.L. and Zimmermann, T. Information needs for software development analytics. In ICSE '12: Proceedings of 34th International Conference on Software Engineering (2012), 987-996]. Section IV presents current work in this domain covering research from 2013 to 2016.

II. RELATED WORK

A. *C. Bird, A. Bachmann, Fair and Balanced? Bias in Bug-Fix Datasets, 2009[4]*

Software engineers have long been interested in where and why bugs occur in code and in predicting where they might turn up next. Researchers have used bug tracking systems and code version histories as source of bug data to test their hypothesis and build statistical bug prediction models. But Bird and Bachmann have questioned the integrity of the bug fixes records/datasets in 2009 and asked are they fair representation of the full population of bug fixes?

In their research, they include hypothesis testing and the following three biases:

- Sampling Bias: When the data gathered does not represent the full population the sampling bias is said to be introduced. Sampling bias makes truly random sample impossible.
- Commit Feature Bias: It is likely that certain types of commit features are systematically over-represented or under-represented among the linked bugs. This representation is termed as commit feature bias.
- Bug Feature Bias: It is likely that the properties of linked bugs look just like the properties of all fixed bugs. The authors define bug feature bias as over or under representing the linked bugs in the sample.

The authors state processes and humans are imperfect. Historical datasets are used to test hypothesis concerning processes of bug introduction and also to build statistical bug prediction models. The claim is that only a fraction of bugs are actually labelled in those data set and the same goes for code commits. And prediction made from samples can be wrong if the samples are not representative of the population. This threatens the validity and effectiveness of the derived hypothesis and models. The related work shows that bias is big consideration in hypothesis testing. Data quality also differs from project to project. Different projects have different methods of defect handling.

The authors have used Eclipse and AspectJ bug data set from the University of Saarland. Apart from this, they gathered data for five projects: Eclipse, NetBeans, the Apache Webserver, OpenOffice and GNOME. They also mined bud databases: Bugzilla and Issuezilla.

The authors have described patterns for data retrieval and preprocessing. They extracted change histories from source code management system (SCM) commit logs. Regarding bugs, they extracted information related to opening, closing, reopening, assignment, severity, comments, time and initiator of every event. The authors also adopted Fisher et al.s technique of finding links between commit and bug report. The technique involves searching the commit log messages for valid bug report references. But the authors made several changes to decrease the number of false negative links. (see steps mentioned on page 5) To validate the data retrieved, the authors manually scanned a random sample for false positives and false negatives.

Result and Conclusion:

- Experiment on BugCache, a defect prediction model, shows how a biased training set can affect the prediction. As a baseline the authors performed training and evaluating w.r.t. bug severity on the entire set of linked bugs. This gave them a recall of around 90% for all categories. But when they train BugCache on a baised training set the evaluations (recall value) are skewed.
- Thus severity bug feature bias is affecting the performance of BugCache in this case.

As a conclusion of the paper, predictions made from samples can be wrong, if the samples are not representative of the population. Data generation process, in this case defect handling, varies among different projects.

B. V. Basili, M. Lindvall, Linking Software Development and Business Strategy through Measurement, 2010[3]

The author mainly talks about using the GQM+ (goals, question, metric) paradigm to measure the success of goals and strategies on all organizational levels.

This strategy helps to decide when and how to transform goals into operations and how to evaluate the success of strategies with respect to those goals.

Steps to use this model:

- The starting point of GQM+ Strategies is a business goal. Along with this, it requires explicit documentation of relevant context factors and assumptions necessary for understanding and evaluating goals. Use the business goal template for documentation.
- The next step is to ask questions necessary to interpret the goal.
- The questions will yield metrics required to measure the goal and an interpretation that provides the information whether the goal is achieved or not.
- Finally, list all the possible strategies and select one for implementation.
- At the next level down in the model, a software development goal is derived from the strategy.
- Steps 2-4 are repeated for this level.
- Similarly, project level goal is derived and Steps 2-4 are followed

1) Keywords:

- **Business Alignment**: This phrase is used to talk about aligning the business goal with software development strategies.
- **Capability Maturity Model Integration (CMMI)**: CMMI is a process improvement training and appraisal program. CMMI models provide guidance for developing or improving processes that meet the business goals of an organization.
- **MoSCoW**: It is a prioritization approach for requirements and release planning. The term MoSCoW is derived from the first letter of each of four prioritization categories Must have, Should have, Could have, Would like but wont get.
- **COCOMO**: Cost Constructive Model is a procedural software cost estimation model which uses a basic regression formula with parameters that are derived from historic project data and current as well as future project characteristics. Basic COCOMO computes software development effort (and cost) as a function of program size (in KLOC).

2) Features:

- **Motivational Statement**: Organizations constantly need alignment of business goals with development strategies and justifying of cost and resources for software and system development. But we do not have effective methods for linking business goals and software related effort. GQM+ tries to bridge the gap between business strategies and their project level implementation.
- **Related Work**: The paper talks about related measurement programs like GQM, BSC, and PSM. GQM approach provides a method for an organization or a project to define goals, refine those goals into specifications of the data to be collected, and then analyze and interpret the resulting data in the light of the original goals. BSC (Balanced Scorecard) approach links strategic objectives and measures through a scorecard which consists of four perspectives: financial, customer, internal business processes and learning and growth. PSM (Practical Software Measurement) approach offers detailed guidance on software measurement, including providing a catalog of specific measures and information on how an organization can apply those measures in projects.
- **Informative Visualizations**: The paper has a list of figures and tables explaining GQM+ strategies and their ongoing applications. The following figure depicts the eight conceptual components, that form basis for constructing a consistent model, and how these components interrelate. The eight components are business goals, context factors, assumptions, strategies, level i goals, interpretation models, goal+ Strategies elements and GQM graph.
- **Tutorial Materials**: The paper takes on a sample application and teaches how to apply the GQM+ Strategy. The authors start with a business goal of increasing

profit from software service usage. They build assumptions, questions, metrics and interpretation models around it. They then derive lower level goals and perform the same task. Finally relate how the lower tasks will help determine the success of the higher level goals.

3) Improvements:

- **Absence of backup plan**: The users are supposed to make assumptions and select strategies that work best. But the authors did not mention what to do if at some point during the lifecycle, the assumption backfires or goals are not met completely.
- **Effectiveness of GQM+ approach**: The paper offers a list of industries that have adopted the GQM+ paradigm but it might make more impact on the readers if the authors had provided supporting data of the effectiveness of the method.
- **User lacking knowledge of linking goals and strategies**: The paper also fails to point out the importance of determining goals and strategies. Mentioning a list of heuristics or defining who should be responsible for coming up with the correct and effective goals and strategies would help new engineers to pick up this method. The model is more likely to fail, when a person taking charge has a weak understanding of the links between the goals and strategies.

C. An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution, 2011[2]

Refactoring is the process of changing a programs design structure without modifying its external functional behavior in order to improve program readability, maintainability and extensibility.

Relevant studies show different results between refactoring and following bug fixes.

The goal of this paper is to systematically investigate the role of refactorings during the software evolution by examining the relationships between refactorings, bug fixes, the time to resolve bugs and release cycles using fine-grained version histories.

1) Keywords:

- **API level Refactoring**: API-Level refactoring are rename refactorings at the level of packages/classes/methods, add/delete parameter refactorings, move refactoring at the level of packages/classes/methods and changes to the return type of a method.
- **Floss Refactoring**: There are two different occasions when programmers refactor. The first kind occurs interweaved with normal program development, arising whenever and wherever design problems arise. For example, if a programmer introduces (or is about to introduce) duplication when adding a feature, then the programmer removes that duplication. This kind of refactoring, done frequently to maintain healthy software, is called floss refactoring.
- **Change Distilling**: It is a tree differencing algorithm for fine-grained source code change extraction. It was published in IEEE transactions to Software Engineering.
- **Hunks**: List of consecutive source code lines that were added or deleted.

2) Features:

- **Motivational Statement**: In theory, refactoring improves software quality and programmers productivity. However, in practice there are trends that show increase in bug reports after refactoring. This motivates the authors to dig deeper and systematically investigate the role of refactoring. The idea is to study and establish a relationship between refactorings, bug fixes, the time to resolve bugs and release cycles.
- **Related Work**: Weissgerber and Diehl found that a high ratio of refactorings is sometimes followed by an increasing ratio of bug reports. In contradiction, Ratzinger et al. found that the number of defects decreases if the number of refactorings increases in the preceding time period. Dig et al. found that 80% of the changes that break client applications are API-level refactorings. Another study showed that large commits, which tend to include refactorings, have a higher chance of including bugs. Moreover, studies have also shown that there are many bugs in refactoring tools and these tools do a poor job of communicating errors. All these relate to the outcome of the empirical investigation of the authors.
- **Study Instruments and Commentary**: The authors applied M. Kim et al.s refactorings reconstruction technique to find revisions that underwent refactoring and S. Kim et al.s bug history extraction technique to identify bug fix revisions. The refactorings are used for this study are from change logs of Eclipse JDT, jEdit and Columba.
- **Results**: Results of relationship between refactorings and bug reports include in-depth knowledge regarding the following questions
  - Are there more bug fixes after API-level refactorings? Yes, there is a short-term increase in the number of bug fixes after refactorings.
  - Do API-level refactorings improve developer productivity? Yes, when it comes to fixing bugs introduced near the time of refactorings, the average fix time tends to decrease after refactorings.
  - Do API-level refactoring facilitate bug fixes? Yes, Fixes and refactoring often appear in the same revision. Furthermore, it is more likely for a refactoring revision to be followed by related bug fixes than for a non-refactoring revision.
  - Are there relatively fewer API-level refactorings before major releases? No, there are more refactorings and bug fixes prior to major version release.

3) Improvements:

- **Generalization of results**: The authors have closely studied API-level refactorings but in their results they have generalized these to normal refactorings. To a

reader perusing the results, it might mislead to believe that general refactorings have the same trend as API-level refactorings.

- **Results and type of refactorings**: The results generated talk about API-level refactorings and its trends with bug-fixes and release cycles. It would have been more interesting to know what type of refactorings, manual or automatic, conform to these results.
- **Excluding Software Life Cycle Activity**: The authors did not take into account the fact that their subject might practice the activity of refactoring the code before a major release. This fact might skew the last result.

III. Baseline Study

In the paper "Information Needs for Software Development Analysis[1]", authors Buse and Zimmermann have presented the data and analysis needs of professional software engineers. They conducted a survey among 110 Microsoft developers and managers, in which they asked about their decision making process, their needs for artifacts and indicators, and scenarios in which they would use analytics.

A. Methodology

The authors advertised a survey consisting of 28 questions to random, equal-sized pools of engineers and lead engineers(managers). It consisted questions from 4 categories [1]

- **Analytical**: What factors influence your decision making process?
- **Importance and Difficulty**: What questions are important or difficult to answer?
- **Artifact**: What artifacts are important to measure?
- **Indicator**: What indicators do you currently use? What would you like to use?
- **Decision Scenarios**: What decisions could analytics help with?

A total of 110 (57 managers and 53 developers) responded. The authors also recorded both the importance and difficulty of answering the questions in each domain. Importance-related questions were rated on a 4-point scale of Not-important; Somewhat-important; Important; Very-Important. Similarly difficulty-related questions were rated on a 4-point scale of Not-Difficult; Somewhat-Difficult; Difficult; Very-Difficult.

B. Survey Results

In case of analytics, managers rated Data and Metrics as the most important factor to their decision making. Developers, on the other hand, rated their personal experience as most important.

While rating importance and difficulty, participants responded, Figure 1, that questions of insight are generally more difficult to answer than of information, and furthermore that they become even more difficult if they pertain to the future as compared to the past or the present. Surprisingly, questions about the future were rated as progressively less important (though still important in absolute terms). In other words, both developers and managers find it more important to understand the past than try to predict the future;

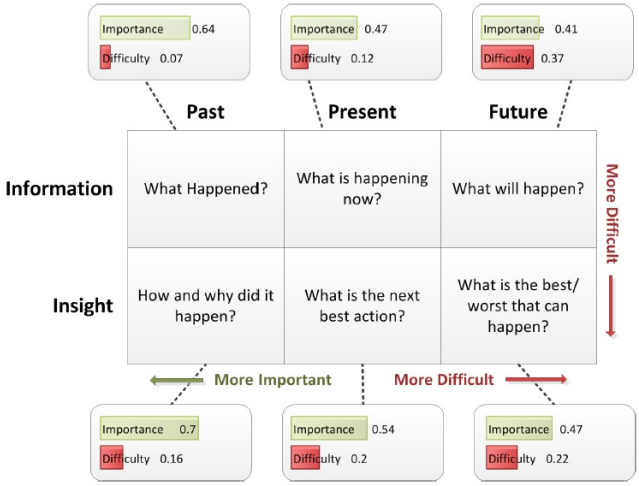

Fig. 1. Analytics Questions. The authors distinguish between questions of information which can be directly measured, from questions of insight which arise from a careful analytic analysis and provide managers with a basis for action.

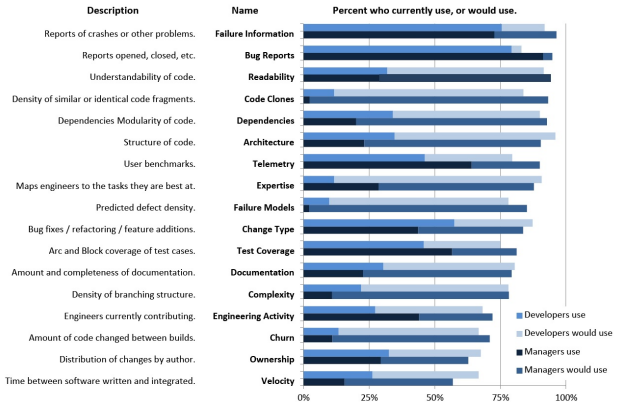

Fig. 2. Percent of managers and developers who reported that they either use or would use (if available) each of the given indicators in making decisions relevant to their engineering process.

Analyzing the individual features of a project was deemed most important artifact by both developers and managers as many important decision relate directly to features (e.g., release planning). Other important artifacts include components, entire products, and bug reports; each of these high-level artifacts are most important to managers. On the other hand, lower level constructs like classes, functions, and test cases are most important to developers.

In addition to artifacts, participants rated on the use of indicators. Figure 2, shows, for each indicator, the fraction of respondents who reported they use or would use the indicator. Failure information was ranked most important and bug reports second most overall.

[1]The questions in the paper are edited for brevity. The original questions are more specific and listed in a technical report [R. Buse and T. Zimmermann, Information needs for software development analytics, Microsoft Research, Tech. Rep.MSR-TR-2011-8, 2010, Appendix 2].

For the decision scenarios, the survey participants described a total of 102 total scenarios. Some of the common themes emerged were: Targeting Testing, Targeting Refactoring, Release Planning, Understanding Customers, Judging Stability, Targeting Training, and Targeting Inspection.

C. Software Analytics Guidelines

The survey responses lead the authors to propose several guidelines for analytics tools in software development including

- Engineers do not necessarily have much expertise in data analysis. Thus tools should be easy to use, fast, and produce concise output
- Engineers have diverse analysis needs and consider most indicators to be important. Thus tools should at the same time support many different types of artifacts and many indicators
- Engineers want to drill down into data based on time, organizational structure, and system architecture

Additionally, the authors identified a set of analysis types that fit well with information needs. In Figure 3 they organize these analyses by what time frame they pertains to (i.e., Past, Present, Future) and their general category of technique (i.e., Exploration, Analysis, and Experimentation).

IV. Current Research

A. *A. Begel, T. Zimmermann, Analyze This! 145 Questions for Data Scientist in Software Engineering, 2014[5]*

The paper is about two surveys related to data science applied to software engineering. The first survey solicited questions that software engineers would like to ask data scientists to investigate about software, software processes and practices. This yielded 145 questions grouped in 12 categories. The second survey asked a different pool of software engineers to rate the 145 questions. Respondents favored questions that focus on how customers typically use their applications. The idea behind categorizing and cataloging 145 questions is to help researchers, practitioners, and educators to more easily focus on their efforts on topics that are important to the software industry.

1) Keywords:

- **Software Analytics**: It is a subfield of analytics with the focus on software data.
- **Affinity Chart**: The affinity diagram is a business tool used to organize ideas and data. The affinity diagram organizes ideas with following steps:
  - Record each idea on cards or notes.
  - Look for ideas that seem to be related.
  - Sort cards into groups until all cards have been used.
- **Churn**: Your churn rate is the amount of customers or subscribers who cut ties with your service or company during a given time period. These customers have churned. Business people say an acceptable churn rate is in the 5 7% range annually, depending upon whether it measures customers or revenue.
- **Telemetry**: Software Project Telemetry is a project management technique that uses software sensors to collect metrics automatically and unobtrusively. It then employs a domain-specific language to represent telemetry trends in software product and process metrics. Project management and process improvement decisions are made by detecting changes in telemetry trends and comparing trends between different periods of the same project.

2) Features:

- **Motivational Statement**: The data science stream is growing. All businesses want to use analytics to better reach their customers. As a result, there are keynote speeches and panel talks about how to make the software engineering community more data-driven. With this survey the authors have given the software developers a chance to express their doubts regarding to data-science.
- **Related Work**: With the big data boom, several research groups pushed for more use of data for decision making, and shared their experiences collaborating with industry on analytics projects. Zhang et al. emphasized the trinity oh software analytics in the form of three research topics (development process, system, users) as well as three technology pillars (information visualizations, analysis algorithms, large-scale computing). Buse and Zimmermann argued for a dedicated analyst role in software projects and presented an empirical survey on guidelines for analytics in software development.
- **Patterns**: The authors received 728 items in 203 responses in the first survey. In order to categorize they used the open card sort technique. Card sorting is a technique that is widely used to create mental models and derive taxonomies from data. In this case, card sorting also helps us to deduce a higher level of abstraction and identify common themes.
  Card sorting has three phases: in the preparation phase, cards for each question written by the respondents are created; in the execution phase cards are sorted into meaningful groups with a descriptive title; finally, in the analysis phase, abstract hierarchies are formed in order to deduce general categories and themes.

3) Improvements:

- **The survey could have been both internal and external**: The survey was internal to Microsoft. The authors could have reached out to a bigger audience and weighted the opinions of other software engineers around the world.
- **Data scientists view on the survey results**: Since the idea to help researchers, practitioners and educators improve their focus and efforts on topics that are important, the authors could have added a step where the data scientists explain how each of the top 24 questions could be analyzed.
- **Taking input from data scientists**: The motivation is to make software processes more data driven. Then

	Past	Present	Future
Exploration Find important conditions.	**Trends** Quantifies how an artifact is changing. Useful for understanding the direction of a project. ■ Regression analysis.	**Alerts** Reports unusual changes in artifacts when they happen. Helps users respond quickly to events. ■ Anomaly detection.	**Forecasting** Predicts events based on current trends. Helps users make pro-active decisions. ■ Extrapolation.
Analysis Explain conditions.	**Summarization** Succinctly characterizes key aspects of artifacts or groups of artifacts. Quickly maps artifacts to development activities or other project dimensions. ■ Topic analysis.	**Overlays** Compares artifacts or development histories interactively. Helps establish guidelines. ■ Correlation.	**Goals** Discovers how artifacts are changing with respect to goals. Provides assistance for planning. ■ Root-cause analysis.
Experimentation Compare alternative conditions.	**Modeling** Characterizes normal development behavior. Facilitates learning from previous work. ■ Machine learning.	**Benchmarking** Compares artifacts to established best practices. Helps with evaluation. ■ Significance testing.	**Simulation** Tests decisions before making them. Helps when choosing between decision alternatives. ■ What-if? analysis.

Fig. 3. A spectrum of analyses suitable for comprising the core of an analytics tool for development activities. We describe each technique and the insights it primarily pertains to. Additionally we bullet a related technique for each.

why not include the opinion of data scientists (since they are also part of the organization and possess knowledge about the processes). The authors could have collaborated with data scientists to rate the questions. Moreover, they could also have created a reverse survey where data scientists ask the questions regarding current software practices and then told the data scientists to analyze how these practices could be transformed into data driven practices.

4) Conclusion: Researchers can use such survey results to collaborate with industry and influence their software development processes, practices and tools. Professionals can get idea about which factors to analyze. Educators can get guidance on what analytical techniques to teach.

B. X. Ye, R Bunescu, Learning to Rank Relevant Files for Bug Reports using Domain Knowledge, 2014[6]

The paper is about two surveys related to data science applied to software engineering. The first survey solicited questions that software engineers would like to ask data scientists to investigate about software, software processes and practices. This yielded 145 questions grouped in 12 categories. The second survey asked a different pool of software engineers to rate the 145 questions. Respondents favored questions that focus on how customers typically use their applications. The idea behind categorizing and cataloging 145 questions is to help researchers, practitioners, and educators to more easily focus on their efforts on topics that are important to the software industry.

1) Keywords:

- **Mean Average Precision**: It is a standard metric that is widely used in information retrieval. It is defined as the mean of the Average Precision (AvgP) values obtained for all the evaluation queries.
- **Bug/defect**: A software bug or defect is a coding mistake that may cause unintended and unexpected behaviors of the software components.
- **Vector Space Model (VSM)**: It is an algebraic model for representing text documents as vectors of identifiers such as index terms. It is used in information filtering, information retrieval, indexing and relevancy rankings.
- **Term Frequency (tf) Inverse Document Frequency (idf)**: Term frequency factor tf(t,d) represents the number of occurrences of term t in document d. Document Frequency factor df(t) represents the number of documents in the repository that contain the term t. Inverse Document Frequency idf(t) is computed using a logarithm in order to dampen the effect of document frequency factor in the overall term weight.

2) Features:

- **Motivational Statement**: When a new bug report is received, developers usually need to reproduce the bug and perform code reviews to find the cause a process that can be tedious and time consuming. A large number of bug reports could be opened during a development lifecycle. For example, 3389 bug reports were created for the Eclipse Platform product in 2013 alone. A tool that helps the developer narrow down the search and lead to the source file containing the cause of the bug would lead to a substantial increase in productivity.
- **Commentary**: A ranking model is defined to compute a matching score for any bug report r and source code file s combination. The scoring function f(r, s) is defined as a weighted sum of k features (k = 6) , where

each feature sigma(r, s) measures a specific relationship between the source file s and the received bug report r

$$f(r,s) = w^T \Phi(r,s) = \sum_{i=1}^{k} \omega_i * \phi_i(r,s) \qquad (1)$$

The user is presented with a ranked list of files, with the expectation that files appearing higher in the list are more likely to be relevant for the bug report. The model parameters w are trained on previously solved bug reports using a learning to rank technique.
The following are the 6 features:

- Surface lexical similarity (equation 3 and 4) the input documents is tokenized using standard IR techniques (remove whitespaces, remove stop words, stem words using Porter stemmer). Since the source file is large, AST parser is used to segment the source code into methods in order to compute per-method similarities with the bug report.
- API enriched Lexical similarity (equation 5) This bridges the gap between the bug report (usually expressed in natural language) and source code file (written in programming language). This is achieved by the following: for each method, the authors extract a set of classes and interfaces names and using project API specification they obtain textual descriptions.
- Collaborative filtering score (equation 6) This accounts for the following observation: a file that has been fixed before may be responsible for similar bugs.
- Class name Similarity (equation 7) This accounts for the following: a bug report may directly mention a class name in the summary, which provides a useful signal that the corresponding source file implementing that class may be relevant for the bug report.
- Bug fixing recency (equation 8) This feature is based on the fact that change history of the source code provides information that can help predict fault-prone files.
- Bug fixing frequency (equation 9) This feature is based on the fact that a source file that has been frequently fixed may be a fault prone file.
- Feature Scaling (equation 10) This helps bring all the features to the same scale so that they become comparable with each other.

- **Data**: The training /benchmark datasets were taken from 6 open-source projects: AspectJ, Birt, Eclipse Platform UI, JDT, SWT, Tomcat. All these project use BugZilla as their issues tracking system and GIT as a version control system.

3) Improvements:

- The training set may be prone to sampling bias. The authors trained only on those data sets that had fixed files associated. This implies that the learner is not trained on certain bugs. The authors should develop a technique for predicting source files for such bug reports.
- The authors could have added a section listing points that pose a threat to their validity. Adding such certification envelope may help other researchers and practitioners provide usage guidelines.
- The authors could have mentioned techniques to further enhance the Accuracy@k metric. Current standards imply that a developer has 70% chance of locating the bug in 10 source files predicted by LR. This could still be time-consuming in certain situations threatening the initial goal of better productivity.

4) Results:

- The new method makes correct recommendations within the top 10 source files for over 70% of the bug reports in the Eclipse Platform and Tomcat projects.
- It performs better than BugScout because LR uses features that capture domain knowledge relationships between bug reports and source files. Also LR is trained directly to optimize ranking results. But BugScout is trained for classification into multiple topics.
- The system requires a max of 3643 sec (of the 6 projects) in indexing, max of 3.56 sec in training and a max of 1.59 in testing making it suitable for practical purposes.

C. J. Cito, P. Leitner, The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud, 2015[7]

This paper reports on a systematic study on how software developers build applications for the cloud. The primary focus of the study was to examine what makes software development for the cloud unique, i.e. what differs in terms of processes, tools, and implementation choices, to other development projects. From the study, the authors extracted a set of guidelines for cloud development and identified challenges for researchers and tool vendors.

1) Keywords:

- **DevOps**: DevOps describe the convergence of previously separated tasks of developing an application and, its deployment and operation. In DevOps, software development and operation activities are often handled by the same team, or even by the same engineer.
- **Infrastructure-as-a-Service (IaaS)**: In an IaaS cloud, resources like computing power, storage, networking are acquired and released dynamically, typically in the form of virtual machines. IaaS customers do not need to operate physical servers but are still required to administer their virtual servers.
- **Platform-as-a-Service (PaaS)**: PaaS cloud provide entire application runtimes as a service. The PaaS provider manages the hosting environment and customers only submit their application bundles. They typically do not need the access to physical or virtual servers that the applications are running on.
- **Software-as-a-Service (SaaS)**: In SaaS, complete applications are provided as cloud services to end cus-

tomers. The provider handles the entire stack, including the application. The client is only the user of the service.

2) Features:

- **Motivational Statement**: The cloud is rapidly growing area of interest. Several platforms such as Amazons EC2, Microsoft Azure, Google App Engine or IBMs Bluemix are already gaining mainstream adoption. However, so far, there is little systematic research on the consumer side of cloud computing, i.e. how software developers actually develop applications in and for cloud. This user-driven approach became the motivation for this paper.
- **Related Work**: There has been a multitude of empirical research on the development of general software applications. So far, research on cloud computing has mainly focused on provider-side issues like server-management or performance measurement. But there is a lack of client-side research. Barker et al. named user-driven research as one of the major opportunities for high impact cloud research. Khajeh-Hosseini et al. stated that the organizational and process-oriented changes implied by adopting the cloud is currently not sufficiently researched. Minor work on cloud programming models was carried out but it did not cover the professional software development environments.
- **Study Instruments**: The authors conducted the study based on the techniques found in Grounded Theory. They used a mixed methodology consisting of three steps of data collection and iterative phases of data analysis. First they defined a set of 23 open-ended questions and conducted qualitative semi-structured interviews with 16 participants. Second, they ran a quantitative survey with 32 questions which were primarily formulated as statements, asking participants to state their agreement on a five-point Likert scale. They gathered responses from 294 professional software developers, and using open coding they identified 4 topics of high interest: fault localization, monitoring and performance troubleshooting, cost of operation, and design for scalability. Lastly, to gain a better understanding and more details on those topics they conducted a second round of qualitative deep-dive interviews with 9 professional developers.

3) Improvements:

- The 25 cloud developers, that were interviewed, raises a question that to what extent do they represent to the overall population of cloud developers. This can be resolved by re-producing similar results on a different set of cloud developers.
- Since the participants for the survey were chosen vis GitHub, there is room for sampling bias. This bias seemed to attracted only software engineers who are actively interested in open source development and who were following progress of at-least one big open source cloud project. Selecting participants from different mindsets could have enhanced the results.

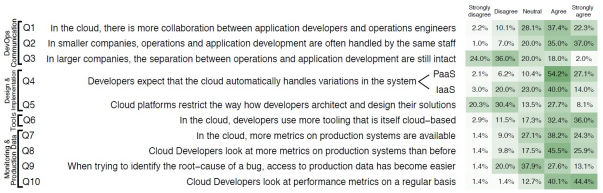

Figure 3: Results from our quantitative survey

Fig. 4. Results from our quantitative survey

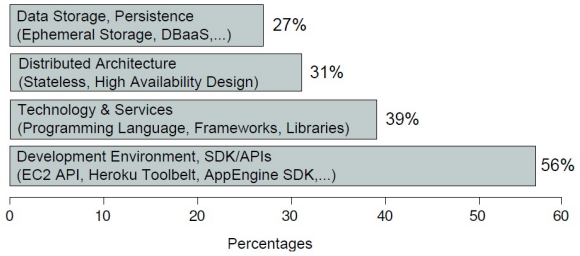

Figure 4: Most significant restrictions developers have encountered on cloud platforms

Fig. 5. Most significant restrictions developers have encountered on cloud platforms

- The authors could have dived deep into metrics. They could have asked the participants (developers) which metrics they would like to user and why. Analysis on the results would have helped the community decide on which metrics to focus.

4) Results:

- The majority of developers thinks about cloud mostly as a deployment and hosting technology following either the IaaS or PaaS model. For these developers, the ability to easily scale applications and the ease of infrastructure maintenance is what makes cloud unique.
- The interviews also revealed that there is a large mindset gap between the developers who think of cloud as either IaaS or PaaS, and those who think of it mostly in terms of SaaS.
- The quantitative results for the Likert-type scale questions from the survey are summarized as Fig 4 shows.

- The interviews show that elasticity, ease of infrastructure maintenance and automation drive most of the changes of software development in the cloud. These can be broken down into two aspects i.e. API-driven infrastructure at scale and volatility of cloud instances.
- As part of the survey, the participants were asked whether they faced limitations in application architecture and design specific to the cloud. The results were as Fig 5 shows

5) Conclusion

1) **Cloud developers design for failure.** Well known cloud users have adopted this mindset. Netflix, for instance, has stated that they use an application called

Chaos Money to randomly terminate cloud instances in production, to force the application design that can tolerate such failures when they happen unintentionally.

2) **Cloud developers design for scalability.** The Twelve-Factor app design manifesto is the de-facto standard when it comes to best practices when building cloud applications.
3) **How does the development and operation of applications change in a cloud environment?** In the cloud, servers are volatile. They are regularly terminated and re-created, often without direct influence of the cloud developer. Our study has shown that the concept of API-driven infrastructure-at-scale and the cloud instance volatility have ripple effects throughout the entire application development and operations process. They restrict the design of cloud applications and force developers to heavily rely on infrastructure automation, log management, and metrics centralization. While these concepts are also useful in non-cloud environments, they are mandatory for successful application development and operation in the cloud.
4) **What kind of tools and data do developers utilize for building cloud software?** Based on our research, more data, and more types of data, are utilized in the cloud, for instance business metrics (e.g., conversion rates) in addition to system-level data (e.g., CPU utilization). However, developers struggle to directly interpret and make use of this additional data, as current metrics are often not actionable for them. Similarly, cloud developers are in the abstract aware that their design and implementation decisions have monetary consequences, but in their daily work, they do not currently think much about the costs of operating their application in the cloud.

*D. **E. Kupiainen, M. Mantyla, Using metrics in Agile and Lean Software Development - A Systematic literature review of industrial studies, 2015**[8]*

This paper makes a systematic literature review to investigate what metrics are used in industrial projects. The authors analyze the reported motivations i.e. benefits of applying the metrics, as well as the effects of using metrics in Agile teams.

1) Keywords:

- **Agile**: In Agile software development methods, the focus is in lightweight working practices, constant deliveries and customer collaboration over long planning periods, heavy documentation and inflexible development phases. Characteristics of Agile are team in control, customer focus, simplicity, following a plan, fast feedback, responding to change.
- **Kaban**: Kaban is neither a software development lifecycle methodology nor a project management approach. Instead, it can be applied to incrementally applied to change and improve some underlying, existing process.
- **Lean**: It is a software development process translated from lean manufacturing. It was adopted from Toyota Production systems. It shares similar values and principles with Agile Methods.
- **Systematic Literature Review**: It is a study methodology that aggregates and synthesizes existing knowledge, identifies the gap in earlier research and provides background information to start investigating a new research topic.

2) Features:

- **Motivational Statement**: Software metrics have been studied for decades and several literature reviews have been published. However, there are no systematic literature reviews on the reasons for and effects of using metrics in Agile software development context.
- **Checklists**: The paper lays down the following research questions to achieve their goal
  - Research Question 1: What metrics are used in industrial Lean and Agile Software Development?
  - Research Question 2: What are the reasons for and effects of using metrics in industrial Lean and Agile software development?
  - Research Question 3: What metrics have high influence in industrial Lean and Agile software development?
- **Patterns**:The authors chose to perform a systematic literature review (SLR).
  The guidelines provided by Kitchenham were used as a basis to develop the SLR protocol. Moreover, the protocol was developed iteratively, performing first a small pilot study and iterating the details of the protocol in weekly meeting among the researchers.
  The strategy for finding primary studies was to (1) automate search (gave 774 papers), (2) select based on title and extract (reduced to 163 papers), and (3) select based on full text (reduced to 30 papers). The pilot study was then carried out by selecting 15 papers (5 by relevance, 5 by number of citations and 5 by random selection). Based on the pilot study, a few improvements to the SLR protocol were made. Finally, data extraction was performed by reading the complete text of all the selected papers and coding the relevant excerpts. Integrated coding was selected as data extraction strategy. The coding started by reading the full text and marking relevant quotes with a temporary code. After reading the full text, the first author checked each quote and coded again with an appropriate code based on the built understanding. In weekly meetings, all authors iteratively built a rule set for collecting metrics and discussed borderline cases.
  The final rule set was as follows:
  - Collect metric if team or company uses it.
  - Collect metric only if something is said about why it is used, what effects it causes, or if it is described as important.
  - Do not collect metrics that are only used for the comparison and selection of development methods.
  - Do not collect metrics that are primarily used to

compare teams. (There were cases where a researcher or management uses a metric to compare teams. The authors wanted to find metrics a team could use.)

- **Data**: Scopus, which contains IEEE and ACM database, was used as a search engine. The search string, hits and dates for the search process of the papers can be found in Appendix A. The criteria for the selection process of the searched papers can be found in Appendix B. The authors used the capture-recapture method, where authors evaluated each others findings to establish repeatability. To do this they adopted a quality assessment technique details of the list can be found in Appendix C. Finally a list of definitions of metrics can be found in Appendix D.

3) Improvements:

- Losing valuable metrics because of stringent criteria. The authors accepted only those studies that talked about a metric and had reasons and effects of using it. This implied that if the study talked about a valid metric with compelling reason but no evidence of effect, then it would not be selected. This could yield to sampling bias which might affect the results.
- Another source of sampling bias is the selection of 30 studies. 7 of them were from enterprise information systems domain and 10 of them large telecom industries. Moreover 8 of them were from the same company Ericsson. This over-representation of large companies in the studies could have affected the results. The authors could have targeted small or medium sized company to equally represent the study set.
- The manual process of searching and selecting studies, understanding which metric is involved and what reasons are explicitly stated in the study, and understanding its impact could have caused incorrect interpretations. To mitigate threats from such sources, the authors could have reached out for outside council or validated its primary result from survey.

4) Results:

- The study identified 30 primary studies with 3 cases in total. The primary studies were published in 12 different journals, conferences or workshops. A large share of the primary studies were published in the Agile conferences.
- RQ1 - The authors found 102 metrics in the primary studies. Further analysis was performed on this set and it was classified based on two categorizations. One of the categorization was by Fenton and Pfleeger (see table 7 in the paper) because their work on software metrics is widely known in the community. Second, it was categorized and compared with the metrics suggested in the original works on Agile methods (see table 8 in the paper) because this would allow us to see whether practitioners follow the metrics suggested in the agile methods or not.
  The Fenton and Pfleeger classification showed that the metrics were largely applied to products, test plans, code, builds, features, requirements, and defects.
  The other classification showed that most popular metrics suggested by the Agile literature are Effort estimate and Velocity. Also it showed that many metrics (39%) found in the primary studies were not suggested in the original works on Agile methods.
- RQ2 - the authors divided the reasons and effects of using the metrics to five categories: Sprint and project planning, Sprint and project progress tracking, Understanding and Improving quality, Fixing Software process problems and Motivating People.
- RQ3 - the authors have highlighted the most influential metrics found in the study. The performed quantitative analysis be assessing the perceived importance of each metric from 1(low) to 3(high). The metrics are considered important if the author of the primary study or case employee praised it, if there were signs of continuous use, and if metrics had positive correlation to important output measures.

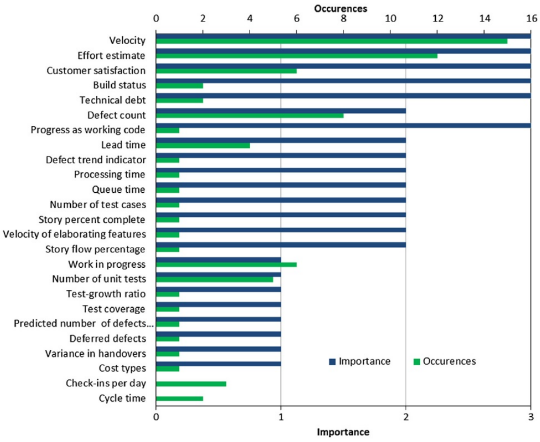

Fig. 4. High influence metrics based on number of occurrences and perceived importance factor.

Fig. 6.

6) Conclusion

1) Common Agile software development methods in industry are Scrum, Extreme Programming, Lean Software Development and Kanban.
2) Agile methods value individuals and interactions, working software, customer collaboration, and responding to change over processes, documentation, contracts, and plans. Agile development emphasizes short development cycles, frequent deliveries, continuous face-to-face communication and learning.
3) According to Pfleeger, we use metrics every day to understand, control, improve what we do and how we do it.
4) In Agile, practitioners can add and invent new metrics according to their needs.

V. COMMENTARY

1) In the paper[5], The authors in this paper tried to understand the questions that software engineers would like to ask to data scientists about software. To me, this paper seems to be inspired from the paper, where the authors have surveyed 110 software developers to present a spectrum of information needs and the corresponding insights gained from them. Moreover, the authors categorized those information needs into 9 categories. Interestingly, Thomas Zimmermann, co-author of that paper.
2) In the paper[6], The authors of this paper have concentrated on using bug reports to narrow down the source files containing the source. A bug report is an important artifact in the software development lifecycle. In the survey, conducted by authors Buse and Zimmermann in paper 1, it is evident that bug reports are extensively used by both managers and developers in their daily development process. Thus the authors have extended this result and tried to make the process of locating the bug efficient.
3) In the paper[7], The authors of this paper were interested in the role of data and operational metrics in cloud development. This was the basis of paper 1, where the authors pointed out that software development decisions should be largely based on metrics. By the extending this rule, the authors of this paper show that metrics indeed plays a major role in cloud development.
4) In the paper[8], The authors in this paper are focused on metrics in Agile software development. This is similar to paper 1, where authors Buse and Zimmermann surveyed information needs of software managers and developers at Microsoft and found that the most used information is related to quality. The information needs for metrics are related to supporting communication and decision making. This paper can be viewed as an extension of paper 1, where the focus is on what are metrics, what are the reasons for and effects of the metrics in Agile software development.

VI. REFERENCES

References are important to the reader; therefore, each citation must be complete and correct. If at all possible, references should be commonly available publications.

REFERENCES

[1] R. Buse, T. Zimmermann, Information Needs for Software Development Analytics. 34th International Conference on Software Engineering (ICSE), 2012.

[2] M. Kim, D. Cai, S. Kim, An Empirical Investigation into the Role of API-Level Refactorings during Software Evolution. 33rd International Conference on Software Engineering (ICSE), 2011.

[3] V. Basili, M. Lindavall, Linking Software Development and Business Strategy through Measurement. 43rd volume IEEE Computer Journal in 2010

[4] C. Bird, A. Bachmann, Fair and Balanced? Bias in Bug-Fix Datasets, proceedings of the 7th meeting of European Software Engineering Conference and the ACM SIGSOFT symposium on the Foundations of Software Engineering 2009.

[5] A. Begel, T. Zimmermann, Analyze This! 145 Questions for Data Scientist in Software Engineering, proceedings of the 36th International Conference on Software Engineering 2014

[6] X. Ye, R. Bunescu, C. Liu, Learning to rank relevant files for bug reports using domain knowledge, proceedings of the 22nd ACM SIGSOFT International Symposium on Foundation of Software Engineering, 2014

[7] J. Cito, P.Leitner, T. Fritz, The Making of Cloud Applications: An Empirical Study on Software Development for the Cloud, proceedings of the 10th Joint Meeting on Foundation of Software Engineering, 2015.

[8] E. Kupiainen, M. Mantyla, J. Itkonen, Using metrics in Agile and Lean Software Development - A Systematic literature review of industrial studies. Journal of Information and Software Technology Volume 62 Issue C (pages 143-163), 2015