Software Engineering Measurement Report

Abstract

This report considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available and the ethical concerns surrounding this kind of analytics.

Introduction To Software Engineering

Software engineering is the process of analysing user needs and then proceeding to design, construct and test end user applications that will satisfy these needs through the use of software programming languages. The NATO Software Engineering Conference took place in 1968 and was attended by industry experts on computer software. At this event, they came up with a novel concept that software was in crisis. Software projects commonly were completed late, came in over-budget, were of low quality and didn't work as intended. Software engineering was the discipline devised to rectify the software crisis and established to ensure that software applications were built correctly, consistently, on time, on budget and within requirements.

Although, the discipline of software engineering has developed and been established for decades now, there are problem areas that exist. How do we approach the task of measuring this discipline? How can we deduce the productivity of a software engineer? How can we judge the product of the software engineering process? How do we assess the process itself? This report will endeavour to answer these queries by i) utilizing measurable data ii) providing an overview of the computational platforms available iii) detailing the algorithmic approaches available and lastly iv) the ethics surrounding such analytics.

Measurable Data

Source Lines Of Code

Counting the number of lines of code written had long been the standard for measuring the software engineering process. Software engineering is a discipline that promotes efficient and easy-to-understand code and thus productivity does not equate to producing excess code. In an academic paper entitled 'Analysis Of Source Lines Of Code (SLOC) Metric' by Bhatt, Tarey and Patel, the authors outline nine flaws in utilizing this software engineering metric:

1) Lack of Accountability

The counting of source lines of code only factors in the coding phase of the project which usually accounts for only 30% to 35% of the overall effort.

2) Lack of Cohesion with Functionality

Through experimentation it has been repeatedly confirmed that effort is highly correlated with source lines of code, however functionality is less well correlated with source lines of code. Skilled developers may be able to produce a program with the same functionality in far less code.

3) Adverse Impact on Estimation

Estimates based on lines of code can go wrong, in all possibilities.

4) Developer's Experience

As mentioned above, skilled developers may be able to produce a program with the same functionality in far less code than a less experienced software developer. Hence, implementation of a specific logic differs based on the level of experience of the developer.

5) Difference in Languages

When evaluating two applications that provide the same functionality but are written in different languages, the lines of code required to develop each respective application will not be the same.

6) Advent of GUI Tools

With the advent of GUI-based programming languages and tools such as Visual Basic, a programmer can write relatively little code whilst achieving high levels of functionality. Code that is automatically generated by a GUI tool is not usually taken into consideration when using source lines of code as a measurement tool. Thus, the same task may be possible to achieve with no code at all, but may require several lines of code in an alternative programming language.

7) Problems with Multiple Languages

Nowadays, software is often developed in more than one programming language and the number of languages employed commonly depends on the complexity and requirements on the application. The tracking and reporting of productivity and defect rates pose a serious problem in this case since defects cannot be attributed to a particular language subsequent to integration of the system.

8) Lack of Counting Standards

There is no standard definition of what a line of code is. Do comments within code count? Are data declarations included? What happens if a statement extends over several lines? Organisations, namely the Software Engineering Institute and IEEE Software, have published some guidelines in an attempt to standardise the counting of source lines of code, but it is difficult to put these into practice, especially with the introduction of more and more programming languages every year.

9) Psychology of a Programmer

A programmer whose productivity is being measured by source lines of code will have an incentive to write unnecessarily verbose code. The more management is focusing on lines of code, the more incentive the programmer has to expand his code with unneeded complexity. This is undesirable since increased complexity can lead to increased cost of maintenance and increased effort required for bug fixing.

As clearly outlined in this academic paper, using source lines of code is not an adequate measure of the productivity of a software engineer. The writing of code is only one task in the role of a software engineer, different programming languages can perform the same functionality in less lines of code than alternative languages and using source lines of code as a measure of the productivity of a software engineer encourages the addition of unnecessary and possibly inefficient code. Having software applications with higher code density can be harmful from a technical debt standpoint as complexity within the structure of the code can make it more difficult to add functionality further on in the development cycle.

Number of Commits

Another metric for measuring the software engineering process is based upon the number of commits performed. Commits are arbitrary changes captured in a single moment in time. Their size and frequency do not correlate with the work needed to achieve that change.

An important aspect of the role of a software engineer is that of being a collaborator and consistently making commits throughout their work assists with accomplishing that role of the job. The size and frequency of commits are not highly correlated with the completion of a project and thus commits between programmers should not be compared. What should be committed is often a matter of personal preference and is subject to debate.

To conclude, measuring the software engineering process based on the number of commits made by developers is a flawed and arbitrary process. Instead, the number of commits should be viewed as an indication of activity.

Code Coverage

This metric aims to measure the number of lines within the code covered by the devised test cases. Code coverage can act as a quality assurance tool as the higher the code coverage, the lower the probability of having undetected software bugs. By utilizing this metric, management can obtain an insight into the thoroughness of the code from each developer. A possible shortcoming of employing this metric is that the tests devised must be sufficient in covering all eventualities, otherwise optimal code coverage does not translate into a bug free software application.

Cyclomatic Complexity

This metric is a quantifiable measurement tool used to indicate the complexity of a program. The objective whilst utilizing this tool is to limit code complexity and determine the number of test cases required.

Formula For Calculation

CYC = E – N + 2P

Where:	CYC = Cyclomatic complexity
	E = Number of edges
	N = Number of nodes
	P = Number of disconnected parts of the flow graph

This metric calculates the complexity based upon the number of linearly independent paths through a program's source code. The higher the value for the metric, the more complex the code is and this will result in the increase in probability of the code possessing defects, be difficult to test, be difficult to read and difficult to maintain.

In conclusion, complex code is unreliable, inefficient and low quality. By calculating the cyclomatic complexity, management can work on decreasing the value by simplifying the code and thus increasing the quality.

Code Churn

This software engineering metric assesses the editing of code by a developer. It does so by inspecting the number of source lines of code that were mortified, edited or deleted. The primary objective of measuring code churn is to allow management to take control of the software development process by utilizing the metric as an indicator. When code churn spikes, this could be interpreted as something being off with the development process and management need to evaluate that software engineering process.

Lead Time / Cycle Time

Lead time can be defined as the time elapsed between the identification of a requirement and its fulfilment. Similarly, cycle time can be defined as the time elapsed between the commencement and fulfilment of a project. To some extent, these definitions are currently fluid and may vary from one software development team to another.

The tracking of these metrics is undemanding as it requires just one member of the software development team, commonly the Scum Master, to take note of when the project has been received or commenced and when the project is completed.

Both these metrics can be used by management to assess efficiency of the software engineering process. They can also be used to evaluate the capability of an individual developer. When the software development team obtain a new project, they can evaluate the results of these metrics from previous similar jobs and provide a data-driven estimate for the time required for completion.

Computational Platforms

Personal Software Process

The Personal Software Process platform was developed by the Software Engineering Institute and in 1995, American software engineer Watts Humphrey published this technique that is intended to help software engineers better understand and improve their performance by tracking their predicted and actual development of code. It acts as a framework for self-evaluation and best practice in software engineering.

The Personal Software Process improvement process can be outlined by the following steps:

1) Define the quality goal
2) Measure product quality
3) Understand the process
4) Adjust the process
5) Use the adjusted process
6) Measure the result
7) Compare the result with the goal
8) Recycle and continue improving

It is up to the engineer to prioritise their own improvement but the Personal Software Process allows software engineers to measure and monitor their performance and adjust the process as issues or improvements are identified.

Several modifications of the process exist, however most require a lot of manual input from the software engineer. The Personal Software Process uses forms to collect information from developers, which includes a project plan summary, a defect-recording log, a design checklist and a code checklist amongst others. These forms are not automated to provide flexibility but ultimately this has resulted in data quality problems.

The LEAP toolkit

The University of Hawaii developed the LEAP toolkit to address the data quality issues of the Personal Software Process by automating and normalising data analysis. The software engineer must still manually input data, but unlike the Personal Software Process, the subsequent analysis is automated. It creates a repository of personal process data that developers keep with them as they shift from and to projects.

The LEAP toolkit resulted in being lightweight and portable whilst providing reasonably in-depth conclusions. However, it emerged as being far more inflexible than the Personal Software Process as a new toolkit would have to be designed and implemented for each new task as opposed to just filling out a form as was done for the Personal Software Process.

Hackystat

Hackystat is an open source framework that can be used in the analysis and evaluation of the software engineering process. It has four main design features:

1) Client and server-side data collection
2) Unobtrusive data collection
3) Fine-grained data collection
4) Personal and group based development

Hackystat has been designed to offer the user a visual representation of data. It is easily integratable with code repositories such as GitHub, Bitbucket and alternative options to provide a valuable insight to the developer with regards to the data

Hackystat is automated and collects the information automatically at regular intervals. It analyses much of the same information as the Personal Software Process namely the time spent on the project, the size of the project, the number of commits, as well as some more advanced data, such as code coverage and complexity.

Code Climate

Code Climate is a software product that offers static program analysis tools for developers and can be used to assist them in improving the quality of their source code. It performs automated analysis of the code to identify and flag potential code errors, security vulnerability and instances of flawed coding methodology. It can collect and present quantitative and qualitative metrics describing and summarising various aspects of the code.

The software product is available to the public through a free open-source license and can also be used in conjunction with code repositories such as GitHub and alternative options to provide the user with an insight into areas such as their code coverage, technical debt and a progress report.

GitHub

GitHub is a web-based version-control and collaboration platform for software developers with over 40 million users worldwide. It is an open-source platform where managers can evaluate data from developers and track the software engineering process. The REST API on GitHub can be downloaded to obtain metrics such as the number of commits made to a project, the number of developers contributing to a project and the frequency of an individual contributor’s input. Platforms such as GitPrime, Screenful Metrics and many more are integratable with GitHub and allow management to create visual dashboards and automated reports based upon the metrics obtained.

Algorithmic Approaches

Bayesian Belief Network

In 1999, Fenton & Neil published their paper 'Software Metrics and Risk' where they stipulated that software metrics are misleading indicators in describing all of the different activities undertaken during the software engineering platform and fail to act as reliable indicators of future code dependability. They highlighted the fact that utilizing the Bayesian Belief Network model as a substitute for the issue of imperfect metrics results in the most success.

The model applies well-established statistical theory known as Bayesian statistics to software engineering, made possible due to the advancements in algorithms and software tools. One key advantage of the Bayesian Belief Network model is that it intentionally accounts for uncertainty, incomplete and subjective information while making explicit assumptions that have previously been hidden from traditional statistical approaches.

The Bayesian Belief Network model operates as a graphical network with an associated set of probability tables. A probabilistic value, computed through empirical analysis, is given to each variable within the model and will be computed for every variable regardless of the amount of evidence for the variable. The model collects data such as the defect counts at different stages of testing, the size of complexity metrics at each development phase, and the approximate development and testing effort. This algorithmic approach was been utilized for a wide variety of applications, from empowering the Microsoft Office wizard to decision-making on the Space Shuttle.

Computational Intelligence

Computer Intelligence can be utilized to assist in better understanding the software engineering process. It refers to the ability of a computer to learn a specific task from data or experimental observations. It is particularly useful where mathematical approaches are inadequate in solving a problem.

There are three main areas under Computational Intelligence:

1) Neural Networks

They are not algorithms themselves, but they provide a framework for machine learning algorithms to work together and process data inputs. Neural Networks can be compared to the human brain learning from examples and past events as artificial intelligence and machine learning can provide to us an insight into trend and commonalities in the investment process. They are interconnected groups of nodes that use examples to generate identifying characteristics from the learning material they process.

2) Fuzzy Systems

They can be used to model uncertain problems such as linguistic imprecision by altering the use of traditional logic to include things that may be partially true. Thus, we can use this output to perform approximate reasoning.

3) Evolutionary Computation

This is used to solve optimization problems by generating, evaluating and modifying a population of possible solutions. Evolutionary Computation include genetic algorithms, genetic programming, evolutionary programming, evolution strategies, multi-objective optimization and more.

Multivariate Analysis

Multivariate Analysis deals with the statistical analysis of data collected on more than one dependent variable. There are two families of techniques that one can use under Multivariate Analysis – supervised and unsupervised learning.

Supervised learning methods teach the mapping function through trial and error. The mapping function is an algorithm that maps an input to an output. After teaching the algorithm sufficiently, the function should be able to predict the output for new inputs. Some examples of supervised learning methods that can be used to measure the software engineering process are linear discriminant analysis, k-nearest neighbours and logistic regression.

Unsupervised learning methods do not have a teaching process and so we must discover the structure of the data by only using the data we have obtained. Some examples of unsupervised learning methods include cluster analysis, hierarchical analysis and k-means clustering.

Ethics

Data Collection

There are numerous ethical concerns with regards to data collection. Many people, including developers, are aware that data used appropriately can become a massive tool and consequently feel uncomfortable with their data being collected without their knowledge or consent. When management implement platforms to measure the software engineering process it is common that developers feel under constant scrutiny. To avoid negative repercussions and conflict due to implementing software engineering measuring tools, it is imperative that there is transparency with regards to the data that management are collecting from the developers.

In addition, another important aspect of data collection is the type of data management are collecting. It would be regarded as unethical for management to collect data which is not linked to the work they are conducting. The main objective for the collection of data should be to improve decision-making, aid in identifying problem areas to be rectified and improve internal operations. Developers should be in favour of the data collection to improve the overall software engineering process of the organisation.

Data Regulations

In May 2018, the European Union released the General Data Protection Regulation (GDPR) to replace the previous Data Protection Act (DPA). GDPR has greatly strengthened the protections surrounding data collection, in particular due to the new regulation that all organisations now require consent to hold one’s data and must hold a record of their consent in addition. Due to the new European Union law, organisations are more transparent with regards to the data they hold, how it is stored and how it is utilized. Harsh fines are given to organisations found to have breached elements of GDPR, and so it is essential that management abide by the regulations when measuring the software engineering process.

Data Usage

Management must ensure that the analysis of the data is fair and impartial whilst avoiding the promotion of unwanted working practises. There is responsibility placed on management to utilize the data in an ethical and non-discriminatory manner. Decision-making based upon unethical practice is not only morally unjust but does not utilize the data and derive the benefits of doing so correctly and appropriately.
It is imperative that data is utilized in such a manner that improves the software engineering process and does not simply allow an organisation to track it more accurately. Poor utilization of data can negatively impact the work of developers.

Conclusion

In this report, I have discussed four aspects of measuring the software engineering process – the metrics, platforms and algorithmic approaches to both measure and best utilize data whilst also addressing the ethical issues surrounding such data analysis. There were a multitude of challenges identified that challenge the consistent and accurate analysis of the software engineering process. Through advancements in machine and deep learning technologies, we can expect an improvement in the ability of measuring the software engineering process in years to come. One insight that can most certainly be derived from this report is the complexity and intricacies involved with the process at the moment. To derive the benefits of utilizing data attributed to the software engineering process, management must be informed and astute in deciding what metrics to assess, platforms and algorithmic approaches to use whilst ensuring such measures and action taken based upon the data is ethical.

Bibliography

5 Developer Metrics Every Software Manager Should Care About Article:

https://blog.gitprime.com/5-developer-metrics-every-software-manager-should-care-about/

Bayesian Belief Network:

https://machinelearningmastery.com/introduction-to-bayesian-belief-networks/

Brief History of Software Engineering Video:

https://www.youtube.com/watch?time_continue=11&v=9IPn5Gk_OiM

Code Coverage vs Test Coverage Article:

https://www.sealights.io/test-metrics/code-coverage-vs-test-coverage-pros-and-cons/

Code Coverage Benefits Article:

https://www.ibm.com/support/knowledgecenter/en/SSB23S_1.1.0.15/gtpd3/d3ccodecovben.html

Code Climate:

https://docs.codeclimate.com/docs/overview

Commits Do Not Equal Productivity Article:

https://about.gitlab.com/blog/2016/03/08/commits-do-not-equal-productivity/

Computational Intelligence:

https://cis.ieee.org/about/what-is-ci

Cycle Time:

https://screenful.com/blog/software-development-metrics-cycle-time

Cyclomatic Complexity:

https://www.perforce.com/blog/qac/what-cyclomatic-complexity

https://www.youtube.com/watch?v=rIz1-imryqk

Definition of Software Engineering:

https://www.techopedia.com/definition/13296/software-engineering

https://economictimes.indiatimes.com/definition/Software-engineering

General Data Protection Regulation (GDPR)

https://eugdpr.org/

GitHub:

https://www.howtogeek.com/180167/htg-explains-what-is-github-and-what-do-geeks-use-it-for/

Hackystat:

https://hackystat.github.io/

Lead Time:

https://www.agilealliance.org/glossary/lead-time/

LEAP Toolkit:

https://creator.magicleap.com/learn/guides/magic-leap-toolkit-overview

Multivariate Analysis:

https://www.sciencedirect.com/topics/medicine-and-dentistry/multivariate-analysis

Personal Software Process:

https://www.win.tue.nl/~wstomv/quotes/humphrey-psp.html

https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5283

https://explainagile.com/agile/personal-software-process/

Software Engineering Institute:

https://www.sei.cmu.edu/about/what-we-do/index.cfm

Source Lines of Code:

https://www.viva64.com/en/t/0086/

https://www.researchgate.net/publication/281840565_Analysis_Of_Source_Lines_Of_Code
SLOC_Metric

Supervised & Unsupervised Learning Methods:

https://blogs.nvidia.com/blog/2018/08/02/supervised-unsupervised-learning/

https://towardsdatascience.com/supervised-vs-unsupervised-learning-14f68e32ea8d

Why Code Churn Matters Article:

https://blog.gitprime.com/why-code-churn-matters/