Measuring Software Engineering CSU33012 Report

Michael Makarenko (student no. 19333583)

Abstract

This report explores software engineering as an activity, how it can be appropriately and accurately measured, and the desirability, feasibility, viability, legality and ethics behind various forms of software engineering analysis. The focus is primarily on productivity and efficiency, how it can be identified, compared, and improved upon. This is an important and complex task that can yield great rewards for managers and companies hoping to better understand and improve the performance of their employees. It is also beneficial for teams and individual developers to keep track of themselves, since busy work is not the same as productive work.

Section 1 – Gathering, Processing, & Measuring

Introduction

As stated above, the primary goal of measuring the software engineering process is to assess and manipulate worker productivity in this specific field. However, there is a glaring difference between programming and other, traditional work. Not all code is created equal, and a few simple lines of code that are not complex nor large may be the result of a substantial amount of time and effort and may be crucial towards the success of a project. Thus, software engineering by its very nature is difficult to quantify.

In recognising this problem, we can now explore attempts to solve it.

What not to do

One possible method is to value code by its complexity. However, this is naïve and problematic, as it is often more difficult yet more desirable to write simple code that does the same job as a larger or more convoluted codebase.

Another naïve method is to quantify code by its size, also known as Source Lines of Code (SLoC), or by the number of code commits within a period of time in a Version Control System (VCS) such as Git. This shares the same problems as the above, as more concise code with the same end functionality is in fact more valuable. It is also of no use to count how many or how often commits are made to a VCS as commit sizes can vary wildly.

Having established that it is difficult to directly quantify the value of code, we must consider more out-of-the-box ways of measuring software engineering.

Code-centrism

Code, at the end of the day, is meant to *do* stuff. By splitting the development process into roughly equivalent tasks and subtasks that improve upon or add functionality, one can measure productivity and work efficiency by the rate at which individual tasks are completed, and by whom; this is measures quantity.

Leaning further into this line of thinking, one can measure the quality of work by how often errors or issues are found, how long they remain unresolved. In short, as long as the code's functionality is growing and improving at a desirable pace with few problems that are resolved promptly whenever they crop up, one can have confidence that the software engineering process is productive and efficient. This is echoed by the Agile Manifesto, which states that "Working software is the primary measure of progress"[1].

Listen to the pros

However, a code-centric stance is not the only appropriate perspective to take. Industry professionals such as Jonathan Roger from AndPlus LLC, state that the most important thing is customer satisfaction, and this too may be used as one of several useful metrics in order to measure software engineers and their work[2]. As long as the client is kept informed and is satisfied that sufficient progress is being made, one can deduce that all is well, and vice versa if the customer is unhappy.

Other valuable metrics mentioned include peer code reviews, as experienced software engineers can help maintain sufficient code quality and prevent future issues, as well as using historical logged data as a point of comparison to judge whether productivity is being kept steady or not on a per-team or even per-engineer basis. It is important to note here that, following the Eisenhower principle of "what is important is seldom urgent and what is urgent is seldom important", not only should we measure the amount of work completed, but also the importance of that work[3].

We can infer that, due to the inherent obscurity and complexity of the software engineering process, and the fact that any individual projects is to one extent or another, unique in the issues it may face; it is important to combine metrics from various views and sources. Be it clients, code, peers, or logs, all of these points of measurement are valuable and combining them allows one to peer into the answer behind the question of software engineer productivity and efficiency.

We can do better

Additionally, we can follow an alternative path, and look strictly at things that affect productivity either positively or negatively. For example, studies have shown that, especially in software engineering, remote work leads to increased productivity[4]. Research has also unveiled, that lone workers in private workspaces are more productive and less prone to distractions from co-workers[5]. Meanwhile, micro-management has been shown to reduce productivity and worker morale[6]. We can measure software engineering by investigating, implementing, and testing the effects on productivity of such phenomena.

My GitHub visualisation project aims to provide a tool to assist in investigating one such case, namely how many potential collaborators share the same or close timezone with a user. This is because wildly different timezones in the world of remote work do in fact impact productivity, as it is more difficult for teams to stay in sync and schedule important meetings, not to mention the inherent complexities of different timezones and working hours based on daylight times.

But wait, there's more

There is a number of industry-standard metrics, for measuring software engineer productivity.

Sprint Burndown is an agile scrum graph that shows the rate at which work is completed and how much work remains to be done.[7]

Cycle Time is how long it takes for some task to be completed from beginning to end, such as bugs going from raised to being resolved, or a feature going from the drawing board to release[8].

Software Development Velocity, also known as Delivered Value, is the number of features, each with an associated value, completed in some period of time that are ready to deploy.

Keeping track of the number of open pull requests on a VCS and how long it takes for them to be merged allows one to see whether a team is working together properly[9].

All of the above methods and metrics can be used together to get a good picture of the effectiveness of a given software engineering process or workflow, and assess the productivity of teams or even individual developers. This can be implemented in a variety of ways and, of course, there is a number of platforms available to compute and process such data in order to quickly and efficiently provide an overview of the process being measured.

Section 2 – Hardware and Software Platforms

Introduction

Now that we know what kind of metrics and data may be collected for the purpose of measuring software engineering, we may look at how that data is processed. An array of computing infrastructure has emerged in order to handle the task of processing large datasets, in the form of servers and workstations, as well as high-bandwidth networking. There is also a diverse variety of software which enables one to gather the valuable data metrics mentioned above.

Outline

Software may come in the form of workflow planners and task tracking boards such as Jira[10],Trello[11] and Monday.com[12], or time trackers such as 7pace[13]. Metrics can also be collected from sources such as GitHub via an API. These software solutions set the stage for collection the necessary and desired data. It's then up to networking infrastructure to deliver those metrics to the necessary hardware for storage and processing.

The necessary networking depends on if one desires to host and process data on-site or on a remote server (cloud). In the former case, it’s a matter of ensuring that the software platforms and the developer PCs on which they run are configured and connected properly to the on-site workstation/server. Meanwhile cloud solutions are usually a question of picking the right provider for the job, among choices like Microsoft and Google. These large, established service providers have extensive hardware for storage and processing needs, and work best when proper integration is supported by the tracking software. As an example, 7pace offers tight integration with Microsoft’s Azure DevOps service[13].

The right software tool for the job

Some software solutions are rather self-contained and focused purely on delivering insight into a software engineering workflow. One example of this type of software platform is Pluralsight Flow[14], which takes a code-centric approach to measuring software engineering. It uses Git data to build visualisations of what is being worked on when, evaluating developer performance and efficiency, and identifying project bottlenecks. This strong Git integration allows for scalable pull request management too.

Further specialised forms of software platforms in this sphere of software engineering include Code Climate, which is made up of two products, Velocity[15] and Quality[16]. The former is a productivity tool for optimising meetings, detecting risks in Git workflows, displaying various important Key Performance Indicators (KPI) as outlined previously, such as pull request cycle time and measuring business impact and Return on Investment on a per employee basis. The latter is an automated code review and test coverage tool, which can be used to not only enhance the software engineering process but also gain insight on code quality and technical debt.

Section 3 – Computation & Evaluation of Data

Introduction

Over the years, a number of various computation models and techniques have been developed in order to assist in computing software engineering data.

The bad

There exist algorithms and formulae such as Halstead Complexity Measures (HCM), which count operators and operands in source code to get an idea of how complex and maintainable a given codebase may be. Another is Cyclomatic Complexity (CYC) which effectively measures the number of decisions made in a program and derives the effective testability and maintainability of the code.

Both were developed in the 70s and are not effective measures of code value, or not in the modern day at least. The primary reason is that both HCM and CYC are strongly correlated with raw SLoC[17], which has been pointed out previously to be a poor and naïve metric of code quality/value. Additionally, the specific definition of CYC is vague and differs in its implementation across software solutions[18], which is clearly problematic, and a piece of code with a high CYC may still be very readable and simple for humans, such as in the case of switch/case statements[18].

The good

There do exist genuinely useful techniques for data computation that are used by large enterprises both within and outside of the field of software engineering. What better way is there to profile the performance of a software engineer than by the real-world performance of their code? This is where Mean Time To Repair (MTTR) and Mean Time Between Failures (MTBF).[19]

MTTR is a metric of maintainability. It measures the time it takes to react to an incident and get everything back up and running. In other words, the cycle time between a critical issue being discovered or a failed component causing downtime, and it getting resolved. This metric reflects not only on whether code is maintainable enough to quickly repair when it fails, but also how well organised a software engineer or team is in order to be able to quickly diagnose and fix the issue at hand. This value should ideally be as short as possible.[20]

MTBF is a metric of reliability, as it measures the time between a first failure being repaired and a second failure occurring. Simply put, it's the uptime between two failure states, so it is sometimes expressed as Mean Uptime.[21]

While these discrete methods of measuring software engineering are clearly effective if they are being used in the industry and beyond, they still cannot give us the full picture, particularly as MTTR and MTBF only look at the end result of software engineers and their code, and not the creation process itself.

The phenomenal

Beyond what has been outlined above, there is the rising field of Artificial Intelligence (AI) and particularly Machine Learning (ML).
"AI solves tasks that require human intelligence while ML is a subset of artificial intelligence that solves specific tasks by learning from data and making predictions."[22] succinctly describes what we are talking about here. Since we are dealing with data and attempting to assess software engineering processes, we want to look at ML specifically.

In recent times, ML has seen and explosive growth in popularity and capability. The most impressive recent use case would have to be GitHub's CodeQL[23] as seen on Semmle's lgtm.com[24]. A code analysis platform that uses Neural Networks (probabilistic graphs and state machines that attempt to model human brain's learning patterns and behaviours) to run automated code review and security analysis, and labelling codebases with code quality grades. Considering that this specific application of ML in assessing software engineering is trusted by security teams and NASA, Google, and Microsoft, we can confidently conclude that this field has great current and potential strengths.

This is further backed up by a paper by some of researchers at Semmle where they make use of ML as well as the large dataset of code quality and quantity on their lgtm.com to measure developer productivity, going so far as probabilities to each minute between commits that a developer is working. The researchers state "we aim to predict the 'average' coding time of the population of developers." As part of their ML model validation.[25]

As this is a constantly evolving field, I personally expect Machine Learning to continue to further overshadow traditional algorithmic approaches to measuring software engineering.

Section 4 – Ethics & The Law

Introduction

The modern world has seen a meteoric rise in the efficiency and efficacy of data acquisition. While this data hoarding opens up many possibilities and benefits, it is not without its downsides. We must question the moral and legal consequences of the activity of collecting data, as well as concerning the data itself.

The morality

A human being's right to personal privacy is sacred, most people would agree to this. After all, everyone should be well within their rights to close the door to the toilet. Digital privacy is no different. And yet, this brings us to an impasse where we desire to gather as much information as possible to gain proper insights into the software engineering process, with good intentions. But this data, which may be personally identifying, can also be used for malicious purposes to target individuals. This information is also valuable, and data gathered from many people may be sold, or otherwise used to exploit them.

Tracking software may be installed on employee computers in order to monitor their productivity. This can be perceived as problematic for several reasons. It displays a lack of trust[27], an employer acting distrustful towards employees impacts morale which in turn actually worsens productivity and overall business financial performance[26]. Overly aggressive monitoring is also a breach of privacy[27], which, especially for privacy-minded individuals, is a cause for stress; and stress leads to worsened performance[28]. We can see that the ethical problems affect both employer and employee negatively when privacy is invaded with overbearing tracking software. Creators of measurement software and companies/managers who use it must tread carefully, especially in the modern world of increasing remote work where software engineers often use their personal computers for work. For example, I made sure to be careful in anonymising collected GitHub user data in my API visualisation, even though all of the gathered data is publicly available on GitHub. Such is the importance of privacy, at least, in my opinion.

The legality

Several laws and regulations have been passed in order to protect users from being tracked unreasonably, or without knowledge/consent. The most notable being the European Union's (EU) General Data Protection Regulation (GDPR). Harsh fines are levied on organisations that fail to responsibly and transparently handle user data. We live in a world where reckless logging and telemetry is becoming more legally dangerous, and handling personally identifiable data appropriately is becoming ever more important. Yet we cannot simply collect less data, that impacts the efficacy of our goal, assessing and measuring the software engineering process.

This brings us back to the balancing act between collecting enough valuable data for our purposes without violating privacy, both ethically and legally. This is especially true as legal mechanisms exist to protect employees specifically from the kind of overzealous tracking mentioned earlier, in the form of Invasion of Privacy Claims. Personally, I strongly advocate for privacy and give my full support to any defence of this right.

But it's not all doom and gloom, in the software engineering world, software solutions for measuring the software engineering process are built primarily by, and for, developers. The specific examples of software platforms I mentioned in previous sections are most useful for software engineers to better evaluate themselves and their peers.

Conclusion

I have stated that the task of measuring software engineering is important for developers and managers/employers alike. It is worth noting that this assumes that the managers in question are technically proficient in this field to an appropriate extent, ideally having experience as software engineers themselves. I made this assumption when looking at industry leaders in section 1, but this is not a given and depends on proper corporate organisation and structure in the workplace.

As expected, this is a complex topic and a task to which there are multiple evolving solutions. I have found that it is best to combine many sources of information and metrics to paint as accurate a picture as possible about the given process. There are many tools out there that cater to various niches in the field of measuring software engineering, with various approaches to solving the task.

Beyond the mere acquisition of data, I explored the computation and evaluation of these metrics, and found there are many approaches here too; new and old, with varying levels of effectiveness. We can only expect there to be more breakthroughs here as the march of technological progress continues ever steadily forward. I also looked into the moral and legal implications of gathering data, especially in terms of personal privacy. I gave my own personal opinion on the matter of privacy, but also made sure to point out that in practice, privacy need not, and mostly is not, invaded for the purpose of strictly and objectively measuring software engineering.

If there is one thing I believe we can take away from these findings, it's that there is no perfect way to measure software engineering. Processes in this field can vary greatly and methods of measurement constantly shift and evolve. It is important to always look for alternatives to what is being used at present, as moral and legal frameworks change and new, better solutions are developed to tackle this task.

I would like to add that the importance of accurate measurement cannot be overstated, as we are well past the days of relying on gut feeling on whether a certain process was effective or not. Engineers rely on empirical data to drive their processes, and software engineers must be no exception, if we are to hold software quality on the high pedestal it deserves in this world that is coming to rely ever increasingly on computers for everything to run smoothly.

Sources

[1] https://agilemanifesto.org/principles.html
[2] https://stackify.com/measuring-software-development-productivity/
[3] https://www.7pace.com/blog/how-to-measure-developer-productivity
[4] https://www.indexcode.io/post/why-your-software-team-should-be-remote
[5] https://www.sciencedaily.com/releases/2008/02/080220110323.htm
[6] https://www.eleapsoftware.com/7-ways-micromanagement-negatively-affects-employees/
[7] https://www.scruminc.com/sprint-burndown-chart/
[8] https://www.wrike.com/blog/what-is-cycle-time-formula/#What-is-cycle-time
[9] https://www.indexcode.io/post/best-kpis-to-measure-performance-success-of-software-developers
[10] https://www.atlassian.com/software/jira
[11] https://trello.com/en-US
[12] https://monday.com/
[13] https://www.7pace.com/
[14] https://www.pluralsight.com/product/flow
[15] https://codeclimate.com/velocity/
[16] https://codeclimate.com/quality/
[17] https://books.google.ie/books?id=DxuGi5h2-HEC&pg=PA137&lpg=PA137&dq=halstead+software+science+sloc+correlation&source=bl&ots=0UqysjT5pS&sig=R9RZaCdHCaNz_T3bzBSEk5wK78g&hl=en&ei=gQhwTrSHH4-htwfvg9GOCg&sa=X&oi=book_result&ct=result&sqi=2&redir_esc=y#v=onepage&q=halstead%20software%20science%20sloc%20correlation&f=false
[18] https://www.cqse.eu/en/news/blog/mccabe-cyclomatic-complexity/
[19] https://www.plutora.com/blog/failure-metrics-mttr-vs-mtbf-vs-mttf
[20] https://cio-wiki.org/wiki/Mean_Time_to_Repair_(MTTR)
[21] https://en.wikipedia.org/wiki/Mean_time_between_failures
[22] https://www.freecodecamp.org/news/ai-vs-ml-whats-the-difference/
[23] https://codeql.github.com/
[24] https://lgtm.com/
[25] https://codeql.github.com/publications/measuring-software-development.pdf
[26] https://hbr.org/2016/07/the-connection-between-employee-trust-and-financial-performance
[27] https://blog.sage.hr/invasion-of-privacy-at-workplaces/
[28] https://www.corporatewellnessmagazine.com/article/workplace-stress-silent-killer-employee-health-productivity