Measuring Software Engineering

Since the term 'Software Engineering' was coined at a NATO conference in 1968, we have been attempting to uncover what distinguishes a great software engineer from an average one. It has become the industry norm to 'measure' employees in an attempt to determine their added value to the firm. Although it may be easy to list the qualities of this idealistic 'great' software engineer – somebody who is productive, writes code that is simplistic yet of good quality, meets deadlines, and collaborates and communicates well as a member of a team – these qualities can be subjective, and are not easy to measure. Consider for example meeting deadlines. If this was the sole basis upon which we measured programmers, development projects are destined to fail before they have even begun. I say this because anybody can meet a deadline, but often this is at the expense of the quality of the work. In this report I will look at data selection and limitations, the systems where software engineering metrics are computed both historically and currently, the algorithms and methods that are being used to measure developers, and the ethical problems associated with measurement and analysis.

What data?

Although often overlooked, data collection is a vital step in any analysis process. By looking at past data we can predict future performance and discover trends and qualities that will help us better understand an individual's development process. Prior to analyzing and measuring progress of developers, the correct data must be selected.

Have a specific Purpose

Before one can begin to dissect results and offer solutions, the question of what we are looking to measure must be asked. The following list offers some of the key attributes I believe a great software engineer should have. As can be seen, each attribute requires the collection of unique data before we can attempt to measure it.

1. ***Productivity***: How well a developer uses their time. Can they achieve results with minimum time wastage? To test this attribute one might look at the percentage of time a developer is idle within a given time period, or the output delivered within a given time period.
2. ***Effectiveness:*** The extent to which a developer successfully achieves their objective. For example, an effective programmer who is asked to eliminate bugs might not write the most eloquent code, but they will find and fix the most bugs. To measure effectiveness one might look at an engineer's success rates of testing and debugging code.
3. ***Skill:*** How well a developer does a certain task or action. We might consider a particularly difficult technical challenge and look at the time required to complete this, or the creativity of the solution offered.
4. ***Reliability:*** How likely a developer is to complete tasks on a consistent basis. We might consider how often the developer in question successfully meets deadlines, and the quality of the work they deliver.

5. ***Communication:*** How strong is a developer at communicating with colleagues and how well do they collaborate with others while working on a project. When attempting to measure this attribute we could look at how many times an engineer contacts a team member within a given time period, and the relationship they have with their peers and management.

Defining the above attributes highlights an important rule one should follow when measuring software engineering: The first step in data collection is to define a **specific purpose** for the data. It is not enough to simply want to measure 'a great developer', as this is subjective and can mean a number of things. As a result, the data collected will likely by obscure and lead to poor or misleading analysis.

Types of data

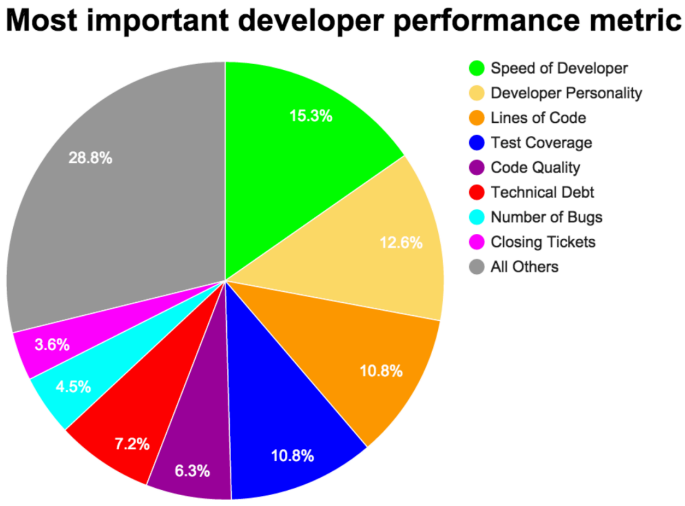
Once we have answered the question of why we are looking to collect data - with a specific purpose in mind – we can move to the next question in data collection. What kind of data is potentially relevant to our objective? In practice, there are number of different types of data collected from developers. The most prominent category of data collected is code related.

- ***Lines of code:*** A straightforward method – simply count the lines of code that a software engineer produces over one unit of time. Personally, I am not a fan of this metric. It is common knowledge that a good practice when writing software is to KISS – Keep it Simple Stupid. This means making your code as accessible, straightforward, and short as possible while also maintaining a high quality. The LOC metric contradicts this, and encourages spreading code over a number lines when it could have been written just as effectively and more clearly on one line.
- ***Commit count:*** Counting the number of times a developer commits their work to a repository within a given time period. Like the LOC measurement method, I have my reservations about this as an effective metric. Of course, it is good practice to commit code regularly so you are not at risk of losing work from your hard drive. However, it is obvious that fifteen near empty commits in a day is far inferior to 4 worthwhile commits, where code has been improved and updated each time. So while this metric encourages good practice, it is not a good outlook on the performance of a developer.
- ***Bugs*:** This metric involves measuring how much time an engineer spend fixing bugs within a week. In practice, firms do not look at how many bugs, as there will always be some and this number will vary greatly from project to project. This includes both fixing issues once you've identified them or troubleshooting issues when they come up. In my opinion this metric is much less misleading than the previous two, and an effective way to quantify the performance of a software engineer.
- ***Test Coverage:*** Measuring the amount of code (usually as a percentage) covered by a set of tests. In practice, a key part of any developer's job is to ensure their code is robust under all circumstances. As a result, testing is hugely important and this metric offers a good reflection of the extent to which testing has been carried out. A good, thorough developer should have consistently high test coverage.
- ***Technical debt:*** Technical debt is a concept in programming that reflects the extra development work that arises when code that is easy to implement in the short run

is used instead of applying the best overall solution. This metric is expressed as a percentage, with a higher figure indicating lower quality of code. Of all of the metrics I have studied, I believe this is perhaps the most worthwhile as it rewards work of good quality. In the industry, a rule of thumb is that codebases with a technical debt ratio over 10% should be considered candidates of poor value.

It is clear that there is no one superior type of data or metric when attempting to measure the performance of a software engineer. Although code based metrics are a good place to start and less expensive to collect, each has a unique set of pros and cons which must be taken into account.

Limitations to data collection:
It should be noted that not all attributes are easily measured. Qualitative aspects are much harder to rank and compare than those that are quantifiable. Consider diligence, how does one compare developers on this aspect? Can we really say that one person is harder working than another, or map progress in an engineer's diligence over time? One could find more examples of this when looking at aspects such as team morale. Most would agree that a strong team morale has a positive impact on how the team performs. But there is no scale used to represent the morale of a team, and I believe it is impossible to collect and record this aspect accurately.



300 developers were asked what they believed was the most important metric when measuring a developer's performance. Their results are depicted in the graph above. I found these results to be very surprising – with lines of code ranked higher than technical debt. From this survey, I came to the conclusion that software engineers might not be well enough informed about the data and metrics their work can be measured on.

Where to Compute?

Once firms have decided on the software engineering metrics they will use and have collected the relevant data, they must decide how – and where – they will compute a meaningful result. Historically, metrics and analysis were computed using the Personal Software Process (PSP) and later using automated analytics programs deployed locally such as PRO Metrics. Although firms can develop an internal analytics hardware system, in the industry today there are more firms than ever before tapping into the data analytics market.

Personal Software Process (PSP)

A significant development in the measurement of software engineering was the PSP. The Personal Software Process was a concept first introduced by Watts Humphrey in the 1990s. Humphrey was one of the first in a long list of academics and scientists who undertook to better understand the personal characteristics and behaviours of software engineers that lead to higher quality software. Humphrey believed that by following a structured development process and constantly tracking work and progress, a developer could better understand and improve his/her output. A large component of the Personal Software Process was self-assessment and management. By setting personal goals and time commitments, developers would be better equipped to meet deadlines, improve planning skills, manage the quality of their projects, and reduce the number of defects in their work. Using the PSP model, data collection and analysis were carried out manually. This required a substantial amount of time to be spent recording defect logs, code checklists, project plans and time estimates. When put into practice, the Personal Software Process proved too time-consuming for most developers to implement efficiently. However, it planted the seed of measurement in the minds of many engineers. The concept of measuring and analyzing software engineers with the hope of improving the quality of their output continued to grow, as the PSP was improved upon by the development of automated analysis tools such as PROM.

PRO Metrics

The PROM (PRO Metrics) tool was introduced in the 2003 article titled '*Collecting, Integrating and Analyzing Software Metrics and Personal Software Process Data'.* PROM was an automated tool, used to collect and perform analysis on the data produced by a developers Personal Software Process. This differed from Humphrey's original model in that instead of the engineer having to manually record the data and analysis, it would be done for them by the software. In many ways, this was an improvement on the previous implementation of the PSP as it saved a huge amount of time and effort for the developer. However, one drawback with the introduction of tools such as PROM was that it put a limit on the type of metrics that could be produced. A select amount of data was collected,

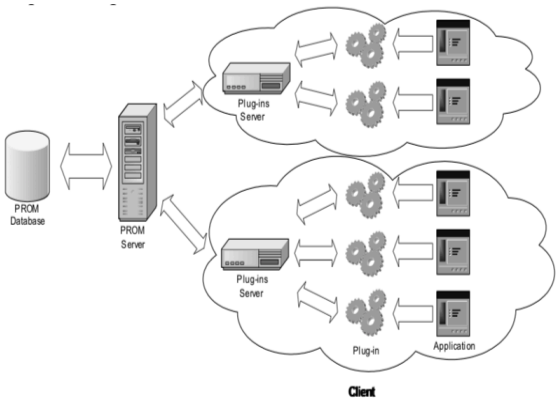


Figure 1: PROM data acquisition architecture

limiting individual developers to standard measurements. The infrastructure of PROM is shown in the diagram. The program was installed on individual machines which would collect the data and send it to the firm's server, where compute-intensive analytics were processed and the resulting analysis were sent to a database storing all user and project information.

Analytics as a Service (AaaS)

AaaS refers to the provision of analytics software and operations through web-delivered technologies. These types of solutions offer businesses an alternative to developing internal hardware setups just to perform business analytics. The idea of Analytics as a Service stems from the ideas of Software as a Service, Infrastructure as a Service, and Platform as a Service. With the amount of data the world produced increasing exponentially every year, firms were struggling to find the capacity to host such a large volume of information. Innovative companies such as Amazon introduced the idea of cloud computing – offering cloud-based services including storage and computation which eased the strain placed on local machines. Analytics as a Service is an attractive option for many software firms today. The task of not just storing data but also providing infrastructure to perform analytics on it is not only resource intensive, but costly. By outsourcing this task to other firms who specialise in it, software companies can focus on their own development projects and deadlines. There are many firms offering services for data analytics, some of the most notable being 'Google Analytics', and 'Amazon Web Services'. There are also a number of firms who specialise in providing analytics services for software engineers. This list includes Code Climate, Source Insight, Scrutinizer, and Codacy.

Codacy

A technology start-up founded in 2014, Codacy offers an automated code review platform. Although only a few years old, the Lisbon, Portugal based firm have worked with hundreds of companies, boasting an impressive repertoire which includes names like PayPal, Adobe, Deliveroo and Cancer Research UK. Codacy software can be installed on premises or accessed in the cloud. It is used by developers to check the quality of code, and implement code quality standards. Codacy offer a free open source service, as well as 'Pro' and 'Enterprise' services, which can be paid for on a monthly basis. Their product offers metrics such as the Churn, complexity, duplication and number of lines of code. It also calculates code coverage, offers a project tracking and evaluation system, and gives developers feedback on security concerns and code style violations. Codacy analysis is flexible, and metrics can be customized to compliment certain projects. Codacy co-founder Jamie Jorge recently claimed "With Codacy, we estimate that we help developers optimise around 30 per cent of their code review time". On a more tangible basis, this corresponds to delivering software two weeks ahead of schedule. I believe firms like Codacy – who offer Analytics as a Service - are the future of measuring and improving software engineering. As the amount of data generated and collected continues to increase, it is more efficient and realistic for firms to pay for this service than to exhaust their own resources doing these compute intensive tasks.

What Algorithms?

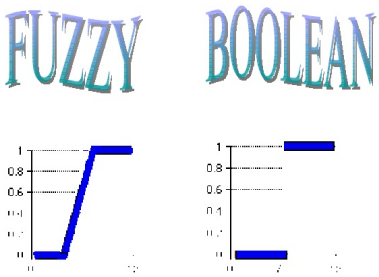
Over the past 10 years, software firms have become more advanced and more capable than ever before. Advanced machine learning algorithms that allow machines to make decisions are at the cutting edge of technology. What once began as a manual process, before becoming automated; the measurement of software engineering is changing. The question I – and the world – ask is to what extent can algorithms be developed to build an intelligent measurement process?

Computational Intelligence

Many people will have heard of the concept of Artificial Intelligence; building computers and other machinery that will simulate human intelligence. Artificial Intelligence is the idea of *smart* devices – machines being able to follow logic to come to a decision like a person would. Although AI is beginning to become a reality and not just a concept – think of self-driving cars which are on the horizon of general release – the world does not yet know how powerful this tool is, and to what extent it can be implemented. Another concept that has had much less media exposure is Computational Intelligence. This is defined as an offshoot of AI in which the emphasis is placed on *heuristic algorithms*. Heuristic algorithms are designed to solve problems faster and more efficiently than a traditional method might, by sacrificing optimality. They will come to a solution quicker than a traditional algorithm, but the tradeoff for this speed comes in the accuracy or optimality of the solution. Some examples of heuristic algorithms are fuzzy systems and neural networks. Unlike Artificial Intelligence, Computational Intelligence does not aim to implement *humanity* into machines. Although it focuses on computers following a logical decision making process like we do, it does not attempt to give machines emotional intelligence.

Heuristic Algorithms

I will briefly examine two common heuristics that are essential in the development of computational intelligence.



A *fuzzy system* is a mathematical system based on fuzzy logic. Computers traditionally operate using binary logic, where variables can take only the discrete values of 0 or 1, equating to true or false. Fuzzy logic is a mathematical idea that allows variables to take continuous values between 0 and 1 – so the actual value is 'fuzzy', partially true and partially false. This system more accurately represents the way we think, and brings mechanical computation a step closer to human logic. For this reason, many of the heuristic algorithms developed when researching the idea of computational intelligence are based on a fuzzy system.

Neural networks are computer systems modelled on the human brain and nervous system. A neural network comprises a set of algorithms that attempt to identify underlying relationships in a dataset. A neural network is made up of learning algorithms; the more data it analyzes the more accurate the system becomes at successfully recognizing and identifying future data. Like people, a NN learns by example, and improves with practice.

Neural Networks and conventional computers take different approaches to problem solving. Unless the specific steps that a conventional computer needs to follow are known, the computer cannot solve the problem. That restricts the problem solving capability of conventional computers to problems that we already understand and know how to solve. The idea of neural networks is key to Computational Intelligence – computers being able to come to decisions based on previous, similar situations they have encountered. This concept can be implemented into software engineering performance metrics in the future.

What does this have to do with measuring Software Engineering?

I believe that Computational Intelligence could be the future of measuring software engineering. In the past, the most complete analysis occurred when developers had flexibility over the metrics they choose to represent their performance. The original Personal Software Process offered engineers complete freedom in selecting and customizing their analysis. The drawback of this was of course the manual nature of it – and the amount of time spent recording, calculating and studying metrics. Current tools solve this problem by eliminating the developer's effort and automating the process, however this also restricts the data collected and the metrics calculated to those the tool offers. What if CI could solve this issue? A typical analytics tool is composed of traditional algorithms – which follow a set of instructions to come to a solution. What if we had an analytics tool that could adapt and improve the solution offered based on the data it took in? The pattern matching element of Computational Intelligence would be hugely informative here. The ideas of fuzzy logic and neural networks would combine to offer a developer unique metrics that matched their unique style. The world has come to accept that there is no one way of performing a task. Different people have different methods, quirks and traits to get the job done. Methods that work well for one may not work for another. I believe the measurement process should respect and model this as much as possible. The most important aspects of measuring software engineers should be self-comparison and self-improvement. Developers should be able to track their own progress in an effort to understand what will make **them** as an individual perform better. To do this we need tools that can cater for each individual and learn from them. The data collected for an engineer should be used primarily to measure that engineer, and not to compare them to their peers. By developing and incorporating Computational Intelligence, I believe we can make tools that will be able to achieve this and ensure developers perform to the best of their ability.

Ethics

Measurement and comparison are essential in all industries today. How is a firm able to compete if they cannot measure their performance and map their progress? In the world today this is a problem all companies face. To have any sort of competitive advantage they must avail of in-depth analytics and performance metrics. The antithesis to this is the welfare of employees. Surely being constantly watched and evaluated must have some impact on the mental health of developers and – as a result – the quality of their work. In this section I will be researching the ethics and legality of measuring software engineers and will conclude with my opinion on this practice.

Data Ownership and Monitoring

A topic that is often glossed over and misunderstood is the idea of ownership of employee data in a firm. The concept of a company car is easy for people to grasp – as a perk the firm is lending you a car but at the end of the day they still own it. Although data follows a similar trajectory, people are often misinformed and do not realize the extent of this policy. The fact of the matter is: firms have access to emails, screen content, even an employee's *keystrokes.* Many are blissfully unaware of this, and continue to spend time using a work device engaging in their private life. I find it incredibly frightening that one day I could be held accountable for something I may type, delete and forget about. What I find even more horrifying is a massive proportion of workers are unaware that the IM they sent a colleague about their boss could someday come back to haunt them! Of course, the chances of this occurring are negligible, but my problem is the lack of clear legislation/policy both on behalf of the firm and governing boards. I imagine the issue of data ownership is addressed somewhere within the pages and pages of an employee's contract, or firm policy documents. Yet, one could search long and hard for this information and finish up frustrated and as unsure as they were beforehand. Within the EU there exists *Article 29 of the Data Protection Working Party as written by the European Commission* – which deals with the surveillance of electronic communications in the workplace. This document states that employer monitoring activity should be transparent, necessary, fair to employees, and proportionate to the concerns that it tries to ally. In general, the law implies that employees should not be monitored and their right to privacy should not be limited unless there is a justified reason – such as a suspected criminal offence. I believe this is not upheld within the software industry. Constant measurement, evaluation and critiquing tools may be transparent and some may argue that they are necessary, but I don't believe they are fair to employees or proportionate to the concerns they attempt to address.

Impartiality of Measurement

Another massive issue with the measurement of developers is the idea of impartiality or fairness. How can we assess bias within this system? Standardized metrics are often used to evaluate a developer's performance; pitting one engineer against another. As I have touched upon earlier, everybody is different and goes about their work with a unique approach. The typical, standard metrics might not favor an individual's work style. Consider a developer who takes frequent breaks while working. Maybe they need 10 minutes after every hour of work to stretch their legs, refresh and refocus. The developer in question may find they work much better by following this practice, but it will equate to a negative

productivity metric in an analysis tool. If this worker's performance was evaluated based on a metric which looked at the number of distractions they had during a day, it would portray him/her as somebody who is easily distracted and not particularly productive. Although this is just one example, I could find many more to back up the idea of bias in the measurement of developers. I think in general a good software engineer would have little issue with constant measurement. The problem lies within the accuracy of this measurement – are developers being punished because they are different? I believe this is an ethical concern for employees working in the software industry. Software engineering is not just a logical, step-by-step procedure; it is a profession where developers can express themselves and have a creative outlet. If engineers' differences are being seen as a disadvantage, then self-expression and creativity is discouraged. In an ideal world, the metrics used to measure the quality of a software engineer would be unbiased. Unfortunately, this is not currently the case, and the tools we use today are a long way from a system that supports a level playing field.

My Opinion

The more I have explored the ethics of measuring software engineers, the more I seem to dislike this process. Although measuring employee's gives firms a competitive advantage, I don't believe it takes the health of individual employee's into account. For one thing, knowing you are constantly under the microscope of analysis, being evaluated and compared to your colleagues is daunting and can put people under a lot of pressure. Another issue I have is with the huge volumes and types of data being collected. Is this invasion of privacy really necessary, and where does the boundary between work and private life lie? As the data explosion continues, I feel this boundary is constantly being blurred and pushed. My biggest concern ethically however, is I think performance metrics discourage creativity and an open work environment. An appropriate axiom of management I read while researching this report was *"What gets measured gets managed".* Often, micro-management is not a good thing. By instilling performance metrics into a work place you are suggesting certain developer characteristics are right and others are wrong. By managing software engineers to this extent I think we are eliminating individuality and creativity, or dehumanizing the workplace. I can't help but be reminded of *Taylorism* – a management model presented by Fredrick Taylor. To ensure the greatest efficiency in a workplace, every job was split into small, specific segments and workers were measured using a stopwatch as they performed the same task over and over again. Although this process resulted in a huge increase in production, Taylor's method was criticized for its unethical treatment of workers; who were viewed as mindless, emotionless, factors of production. I am of the opinion that the measurement of developers in software firms has similar ethical issues. If the current trend of constant measurement continues, engineers will be left devoid of the ability to express themselves, and freedom in how they go about their work. Despite my previous points, I do understand the importance of developer performance metrics. There are certainly advantages to measuring how often an engineer commits their work to a repository for example. By tracking this metric, the company is promoting good practices and ensuring they are met. However, I think it is futile to use this as a basis for performance and code quality.

Conclusion

I have learnt a huge amount about the attempts made to measure software engineering over the course of writing this report. This is a process that is constantly being developed and reinvented. Hopefully by being specific with the data selection process, moving to an Analytics as a Service(AaaS) platform, using the technological advances of Computational Intelligence, and addressing some of the ethical concerns of measurement, we can enhance our current methods, and gain a better insight into what makes somebody a great software engineer. Although there are many issues with collecting developer performance metrics – namely it is costly, inaccurate, and leads to ethical concerns – I believe that measurement offers a firm competitive advantage and helps developers improve their skills.

Bibliography

1. https://www.dagstuhl.de/Reports/96/9635.pdf
2. https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
3. http://thinkapps.com/blog/development/technical-debt-calculation/
4. https://www.techopedia.com/definition/29893/analytics-as-a-service-aaas
5. http://kaner.com/pdfs/metrics2004.pdf
6. http://www.citeulike.org/group/3370/article/12458067
7. http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.138.6806&rep=rep1&type=pdf
8. https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0
9. https://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/cs11/report.html
10. http://ec.europa.eu/justice/policies/privacy/docs/wpdocs/2002/wp55_en.pdf
11. https://www.britannica.com/topic/Taylorism