Measuring Software Engineering

Introduction

This report discusses one of the most controversial topics related to the Computer Science industry - Whether or not the quality and value of the software engineering process can be measured by collecting and analysing raw data. Why is this an important question in today's world? It's no secret that companies and firms look for the top engineers, to have the best of teams working for them so as to get their work done efficiently and successfully. Moreover, by assessing and measuring a team's or an individual engineer's quality of work, the downsides and inefficient features of the engineers can be revealed and this gives the programmers a chance to improve their efficiency by striving to minimise the causes of their inefficient work.

Just to stop and think for a moment, could we really look into data to gauge the quality of a software engineer's contributions? Would raw data be meaningful enough to assess which programmers are really "better" than others? Firstly, are some software engineers really "better" than others or does every software engineer have their strengths and weaknesses in different aspects of Computer Science, and their own beauty in the way they work, which make them incomparable to their colleagues? Before we jump ahead and delve straight into the topic of whether or not we can measure software engineering, we need to look at what kind of raw data we can gather from engineers, so as to measure the quality of engineering in the first place.

What data can be gathered?

Many companies in the world today are already gathering data about software engineer's in order to estimate the quality of their work and their value. The types of raw data that can be gathered about a programmer are plenty. An obvious type of raw data we can gather about workers of a software engineering team could be the amount of hours worked by an employee. Data about the amount of lines of code each developer contributes to the program could also be collected, perhaps in order to check which developer has had the hardest impact on the product or

has contributed the most to the product or project. The time taken for a project to be completed could be a good measurement as it could imply the quality of a flatware engineering team as a whole. To measure each individual's quality of work, we could measure the time taken by an individual to complete a small part of component of the project that they've been assigned to, but we also have to carefully associate that with the difficulty of tasks been given to each individual accurately enough. The number of times a programmer gets distracted, can be measured by the use of sensors. The amount of resources used by a team, such as costs or the number of programmers needed in the team could be recorded as well.

Moving on to more promising and programming-specific types of measurements, companies are also able to measure the number of bugs introduced into or removed from the code by a particular programmer in a team. The quality of code could be analysed as well. Quality, as a single component, is too subjective and abstract, and could be described by more objective attributes, like the depth of the inheritance tree, length of the user manual, fan-in or fan-out and cyclomatic complexity of the product. However, code reuse and compactness isn't the only thing that measures the quality of a software product, as maintainability is an important and necessary factor to take into consideration. To a judge whether code written is maintainable, we can collect data about the average length of identifiers, depth of condition or loop nesting and the length or amount of comments accompanying the code.

There are a variety of different measurements that can be made about a developer, or their code, out of which some are introduced later in this report. Surely, there's no argument that data about a software engineer or his/her work cannot be gathered, as there is so much data to look at that would have a direct or indirect connection to the quality of their product. But before we come up with more ways to measure features of a code or the qualities of a software engineer, we need to find a way to use some of the data that we have already mentioned to properly assess and compare different software engineers. Can the types of data mentioned be analysed to determine the value of a software engineer? Can the high-level work and design of a system by a software engineer be measured by gathering such low level data?

Can this data really be put to use?

The work that a software engineer does is too sophisticated complex and high-level to be boiled down to analysing raw data. Software engineers are designers of complex systems and code, and trivial data like hours worked can surely not correctly determine the quality of a developer's work. Hours worked is a could measure of how dedicated a person is, not how good one is at his/her job. Two different developers could do the same amount of work but take a different amount of time to do it, and so hours worked surely mustn't have anything to do with evaluating a software engineer.

Similarly, the concept of measuring lines of code could also be discarded, as different programmers have different ways of writing code. If one developer of a team contributes more lines of code to a project, and another developer contributes less lines of code to achieve a more difficult task, we cannot conclude that the first contributor had a greater impact on the product. What's worse, is that the first developer could be a bad coder, who introduced more unnecessary lines of code. Using sizes of code-contributions as a measure could encourage developers to write bad and inefficient code only to contribute more lines to the project in order to be recognised as one of the top members of the team, and this type of measurement could not only be useless, but also damage the work done by engineers instead of improving their work ethic in the first place. Sensors to check whether a software engineer is distracted or absent from his desk could result in a incorrect deduction of data, as the engineer could be helping someone else in his team in that time which is more of a priceless quality, and being penalised for that seems like an unfair response to that.

The work ethic of each programmer may be utterly useless, as all any customer, client or company cares about is the quality of the end product, which is what matters much more than how a software engineer accomplished the work needed to be done. Checking the quality of the product and code by looking at the depth of the inheritance tree, length of the user manual, nesting of conditional statements or loops and cyclomatic complexity all seem like a potential way of measuring software engineering, but we have to keep in mind that these will vary not only because of programmers, but because of the nature of the project that needs to be built and therefore, the context of the project is a huge factor when it comes to these sort of measurements, which itself is very subjective and makes it harder to measure the value of software engineers.

This is the reason behind why many people conclude that the task of software engineering is too high-levelled to be boiled to to the analysation of raw data and to be measured. Some of them think measuring happiness instead, is key, because if it's well thought about, success and well written code is usually accompanies with a sense of satisfaction in engineers, while badly written/hard to understand code, or failure to complete a task would bring in disappointment in any developer. This is believed so much so that a tool like NPS (Net Promoter Score) to measure customer satisfaction has also been used for the employees of a software company, namely Redfin.

According to a few software companies and engineers, the closest we can do to measuring software engineering is by measuring how well the company is doing, as good quality products and it's good maintenance is bound to attract customers, thus making the company more successful. So the value of a software engineering team does have a direct correlation to the profit or gross earnings of a firm or company. On the other hand, many software engineers believe that software engineering needs to stop being measured in order to achieve performance.

So is measuring Software Engineering more of a Danger to Productivity than a catalyst?

Measuring lines of code, as stated before, could lead to a decrease in the quality and performance of a system, rather than a tool used for improvement in the work ethic of engineers, and the other types of data seem to be ineffective and almost completely useless. Is measuring software engineering nothing more than just a waste of time and effort then? I strongly disagree with this.

It has been established that data can be gathered, and some results derived from them are meaningless, but just because these results are meaningless, it doesn't prove that there is no way of analysing data to arrive at meaningful measuring of any software engineer. That would be a classic example of falling into the traps of formal fallacy, one of the most common errors made in arguments and formal logic.

There are surely other ways of collecting different types of data and analysing data to measure software engineering appropriately. This is well known globally, so the question is, why are so many software engineers and companies absolutely blind to this? In my opinion, software engineers and developers want their job to seem like something special and artistic in a sense, rather than letting people boil in

down to facts and measurements. More importantly, it is the fear of people holding them back from believing that some software engineers, are in fact, “better” and more productive than others, whether this means smarter or more intelligent, and it angers or scares them that the theory of the top engineers being 10x more productive than average engineers is (very likely) true, which is why these people try to deny that the measurement of software engineering is useful in the first place. There are many different ways in data can be collected and analysed in order to truly display the value of a software engineer (or a team).

Ways to measure software engineering correctly

We covered the fact that counting lines of code per developer could be a terrible metric for software engineering performance evaluation. However, this doesn’t mean we should complete throw away the idea of “size” of a program for engineering measurement. The smaller and more compact a program is, makes it more likely for it to be better written, thought-through, and well designed. Do not lead yourself to believe that this measures software productivity! My colleague and I can write a program for the same purpose in the same duration, with my program being twice as large as his. This makes us equally productive, but it means my program is much more poorly written than his, and reveals that his value as a software engineer is greater than mine.

Apart from the size of the program, checking the amount of code coverage from unit tests could be a competitive metric for measuring software engineering. Naturally, a lack of code coverage is less desired in any area of programming, regardless of the context. However, depending on the context of the product trying to be designed, achieving complete code coverage from unit tests might be more or less challenging. In particular, firms trying to create something that’s never been achieved before usually have no comparisons to test if their results are accurate or not, and code coverage could be a poor metric in that context. A smaller program which has half of the functionalities and covers half of the test cases as compared to a bigger program is not of great use, and so, in most cases, code coverage from unit tests is a way stronger and accurate metric than the size of the program.

Absolute measurement is not the only way to measure software engineering, relative measurement is what’s important. As a boss or founder of a software team or company, the best thing one can do is check the frequency of commits per individual on a site like Github, where they’ll presumably be posting their code. Number of commits vary from person to person and I strongly believe that it is

nonsensical to compare different employees by the number of commits they have. However, comparing an employee's activity compared to their past activity might lead to meaningful results. If an employee's performance decreases, it might imply that he/she is dealing with some issues or is overworked. Relative software measurement can reveal the cause of any inefficiencies, which when dealt with, would help a team or an individual to be much more productive.

Technical debt is one of the most effective metrics of software engineering measurement in my eyes. It is the implied cost of remodelling and modifying software in the future caused by taking a simpler approach for the solution in the present, to achieve functionalities in the product quickly, rather than taking the approach that is harder and would take more time, but which would prove beneficial in the long run. Reducing the amount of technical debt in any software firm, company, or team can lead to a higher quality of code being produced, which would elevate the value of the final product, reduce costs and effort spent into the product, and in turn, this would mean the overall value of the software engineering team would be higher. On the other hand, ignoring and not responding to increasing technical debt is bound to increase software entropy, which would complicate things in the long run, and lead to a higher amount of costs and effort, thus relatively reducing the value of the software engineering team.

I like one tedious, but compelling style to measure software engineering productivity, in which measurement and data collection is completely up to the boss or "leader" of the software team. The best employees may spend most of their time helping other developers close tickets, or sharing experiences and knowledge, which would increase the potential of several other developers in the team. This "true value" doesn't appear in the github commit history or isn't recorded as data, and that's what the problem is. The idea of this approach is realising that the employee that's being evaluated is the best source of information and data for his/her own evaluation. Working closely with employees, asking them directly what value they have contributed to the firm, and helping them find evidence for it, not confined to what's in the codebase, but contributions in terms of mentorship, code design decisions, and even documentation would be an eye-opener for any leader to understand how much each person contributes to the project and who's doing what. Following this up with a methodology of asking a few randomly selected engineers to evaluate their peers, would be a good way of establishing the quality and value of software engineers. Repeatably applying this methodology would give a deep understanding of how valuable each software engineer in the team really is, and could also reveal if the productivity of any software engineer is rising or dropping. This practice of measuring software engineering value, although doesn't consist of numerical values or actual quantity,

is according to me, the single best way to evaluate software engineering productivity. However, innumerable companies are terrible at this because this technique could be challenging to apply in large companies with massive teams.

Cyclomatic complexity is a measurement of how complex the code written really is. To calculate cyclomatic complexity, one must construct a flow control graph of indivisible parts of the program as the nodes and directed edges as an indicator that one process is followed by the next. This would give a measure of the linearly independent paths in through the program. Intuitively, if there are more edges in the graph than the number of nodes, it would mean a program is more complex because more edges means more interactions between many parts of the program which would increase the “mess" and complexity of the code. The goal to set here is to reduce the cyclomatic complexity of a program as much as possible. Any high value of cyclomatic complexity of the product would indicate a lower quality of software engineering, and minimal cyclomatic complexity would be desirable in order to make the code easier to understand, maintain or even add functionalities and features to the product, thus indicating a considerably higher value of software engineering.

According to many software engineers (those who believe in measuring software engineering), the open/close rate of issues or bugs or the MTTR (mean time to repair bugs) is a good estimate for productivity measurement. I partially disagree. Bugs can be of different levels of difficulty and could be either extremely vital to the functionality of the product, or extremely insignificant. Moreover, some bugs are rarer and harder to find than others, and treating each bug as the same “unit” to calculate the open/close rate seems like an unattractive practice to me. I believe that a partial modification of this, however, by classifying bugs into their levels of difficulty and importance, and counting each important bug fix 3 times or so as much as an insignificant bug fix, and then applying the practice of calculating open/close rate of issues would be a more competent and numerically objective way of measuring the quality of the software engineering process.

The open/close rate of bugs is a good estimate of the continuing process of software engineering, but what about the overall value of the system? Bug “volumes” still present in the existing product could be recorded, and this data would not have to be fetched, because if the product is being used, the issues will be reported by the customers themselves. The “severity” of bugs in these scenarios is key, which indicates how many users are reporting a particular bug, and so, how many users are affected by the bug. An example of a bad bug would be reported by many users, whereas a few bug reports out of millions of users

would be close to perfection. This approach of measuring software engineering seems apt for estimating the value of the final entire product after it is released.

Surely, this proves that software engineering can be measured, whether it's to measure the overall value of the final product, the quality of the software engineering team or company, or even the value of each software engineer in a single team. The fact that there is a PSP (Personal Software Process) course shows that measuring software productivity is a vital and possible process in the software industry today. One obvious way to increase the productivity of the software engineering process is to find the defects or downsides using the above methods, and improve on those causes of inefficiency. But how can a bad design be prevented? What if the value of the final product delivered by a software team is evaluated and it turns out to be buggy and pathetic? How can one avoid such situations? There are numerous ways to work well and improve the process of software engineering, before it's too late.

Techniques to Improve the Software Engineering Process

There are many ways, other than responding to the measurements of software engineering discussed above, to increase productivity in the process of software engineering.

Fan-in/Fan-out is an analytical way of measuring how tightly bound the code is. Fan-in is the measure of the number of functions that call other functions in the program, whereas fan-out is the function count that are called by other functions. Keeping a track of the fan-in/fan-out metric could be considerably rewarding in the process of modifying code, to greatly reduce the number of bugs that will have to be dealt with later. A large number of fan-out functions suggests that the program is substantially dependent on the functions that are being called, and any modification to the code in those functions would tend to affect the entire code as a whole, so one should understand that any modification to these types of functions should be done with utmost care and consideration with respect to all affected parts of the program. I can guarantee that if this method is applied perfectly, the number of bugs you have to face in the future is bound to be less than the number of bugs you would have to face if the fan-in.fan-out metric was completely ignored. This is one good way to increase productivity.

Another metric that could be measured and improved upon during the course of the software engineering process is the average length of identifiers, fog index in documents and the depth of conditional nesting. Generally, a greater average length of identifiers tends to give the identifiers a more specific and understanding meaning, which makes the program more understandable and increases the quality of the code produced. Therefore, giving variables specific, but not extensively long names, would be a good way to avoid bugs and maintain code easily.

Fog index, on the other hand, increases the readability and complexity of documents. This is a measure of the number of sentences or words in documents, and a greater fog index would suggest a higher difficulty in understanding the documents due to the number of words and sentences in them. This should generally be kept low.

The depth of conditional nesting is self-explanatory, and one could easily deduce that the higher the depth of conditional nesting is in a program, the harder it will be to understand, and therefore, this depth should be minimal.

Other than these code-feature-specific metrics to increase engineering productivity, a few tools could be of great help to ensure safe, well-designed production of code and could also increase collaboration. The use of Github, which is extremely well known as a space for collaboration of tasks could be used by teams as numerous branches of other branches could be created and merged, and any small merge conflicts could be resolved manually, ensuring the safe production of code.

Trello is another tool which is of great use the it comes to any sort of teams. Using Trello, an entire team can collaborate on the same “board” of separated to-do lists, maybe one for each person or just separated by different types of functionalities, whichever is better suited depending on the context of the product. This enables teams to check what’s done, who’s responsible for what, what tasks are still left to be allotted and to format the work done or to be done in an organised and evadable fashion. I believe that this tool is extremely useful in improving the collaboration of a team.

Docker is another outstanding tool for testing software. Different users have many different platforms, and compatibility issues can be one of the most annoying troubles to deal with. Using docker, completely removes that issue, as whatever platform you want to test your code on can be virtually set up and the code can be

built and run on docker, providing results exactly as it would be if run on actual environments specified in the docker session. This would ensure that the product being designed works for all users on all types of platforms and machines.

Moving on from software tools, the last important factor which could help in increasing software productivity is behaviour in the workplace. In any software team, transparency is key. If a person in a team writes absolutely horrendous code, the owner of that code must be informed about the code s/he has written. Even though this procedure might seem counter-productive at first, as it could potentially increase the hostility in the work environment, I believe that transparency between employees' mindsets could help them understand each other lot better, and it would be much simpler to communicate to each other effectively and understand what exactly needs to be done, and how efficient or reusable the code needs to be. Transparency in the work place could assist in achieving what needs to be achieved with quality and accuracy, which would result in the creation of successful products.

Some people argue that measuring and improving the productivity, efficiency and quality of software engineers is a cyclic process that needs to be done continuously and is never-ending, with the scope for improvements reducing as the process is repeated more frequently. This may or may not be true, but by now, it should seem substantially clear that software engineering can be measured (assuming the metrics are collected and analysed accurately) and measuring it reveals flaws in workers and their causes of inefficiency, which when worked on, helps to increase the productivity of those individuals or the entire team. Therefore, software engineering measurement is possible, with a variety of techniques, and increases the productivity of a team, but is collecting data and analysing data, to boil down software engineers to a number or value continuously as they work, an ethical thing to do?

Should it be done?

Does it really matter that data can be gathered and analysed to measure the quality of software engineering so effectively? Sure, it can be very useful, but I think the Computer Science community thought about whether the process of measuring software engineering could be done or not, so hard that they forgot to stop and ask if it should really be done in the first place. In my verdict, the process of measuring software engineering should be abandoned or reduced as much as

possible, especially in big companies where a lot of employees working might be uncomfortable with this.

Boiling down employees to numbers or values is a breach of fundamental respect to employees, and belittles them to something unimportant. This would definitely affect the sentiments of innumerable employees, as it has been established that they would turn out to be 10 times less productive than the top software engineers, and this is turn would lead to a rise of hostility in the work place. The most important thing for any company is the morale of it's employees. The moment you decide to belittle them and lose their respect, you've lost them and the whole company along with it. Having the admiration and respect from your employees is a must for any successful firm, and measuring their software engineering could turn out to be counter-productive for the company.

Quality improvement and efficiency are not the only things that matter, but we have to acknowledge the ethical boundaries in performing such a process. In an agile development team, or a small software engineering team, where all the team members are extremely focussed on achieving their goal as efficiently and sophisticatedly as possible, measuring engineering productivity could be done and I would be absolutely fine with those kind of situations, because you have the consent of each member. However, in big companies, software engineers are bound to have a different skill levels due to the diversity and size of teams, and measuring software engineering would automatically imply comparisons between different software engineers, which is why I oppose measuring software productivity in big companies in particular, where employees are bound to oppose this kind of breach to their respect as well as privacy.

I believe that Ethics comes before achievements and productivity, and once we've lost that, we'll eventually lose out on everything. Moreover, collecting data by using sensors in the work place or engineers knowing that whatever work their contributing is being analysed to boil them down to a value, will cause them to be self-conscious at work, and may make them unable to focus well, or even change the way they work, maybe even for the worse, because they might try to make their contributions look larger even if that includes duplicate code instead of reusable code.

Using Github, or Trello, or methodologies that don't include measuring software engineering is alright in my opinion, because that just increases the productivity of software engineers without measuring or comparing them to one another. What's

important is that they feel valued, and so if it were up to me I'd dismiss all the practices which include measuring software engineers in big companies.

Moreover, companies, firms and teams who exercise poor methodologies in today's world do exist. They tend to arrive an incorrect and inaccurate conclusions, which work against productivity, rather than towards, and also risk the chance of losing, penalising or demotivating the top software engineers in the group. I was a part of an Android project, a team with the 4 of us and only the two of us happened to make actual contributions all the way. However, we arrived at an incorrect measurement of the value of software engineers in the team, which was done by another team member, who was not included in the programming aspect of our group. He happened to completely neglect the difficulties of the tasks each of us were assigned to, and it seemed like we all did the same amount of work in the report, which gave a very twisted view of the engineering process.

The danger of using measuring-tools incorrectly and assessing the productivity of software engineers poorly is more of a danger than anything else. For me, done right or wrong, measuring software engineering is a definite no as it directly opposes ethics, and unless and until you have everyone's consent, it is wrong to measure the value of any software engineer.

Bibliography

1. Ian Sommerville, "Software Engineering 10th Edition"
2. https://medium.com/javascript-scene/assessing-employee-performance-1a8bdee45c1a
3. https://dev9.com/blog-posts/2015/1/the-myth-of-developer-productivity
4. https://redfin.engineering/measure-job-satisfaction-instead-of-software-engineering-productivity-418779ce3451
5. http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf
6. http://theworkspacetoday.com/2017/01/13/for-engineering-performance-stop-measuring-productivity/
7. https://www.johndcook.com/blog/2011/01/10/some-programmers-really-are-10x-more-productive/

8. https://medium.com/@yupyork/the-best-developer-performance-metrics-6295ea8d87c0

9. https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams

10. http://engineering.kapost.com/2015/08/you-can-and-should-measure-software-engineering-performance/

11. https://martinfowler.com/bliki/CannotMeasureProductivity.html

12. https://cacm.acm.org/blogs/blog-cacm/180512-is-there-a-10x-gap-between-best-and-average-programmers-and-how-did-it-get-there/fulltext