Measuring Software Engineering Report

Pavel Petrukhin

Introduction

Nowadays more and more things are getting automated in the IT industry. The main business reason for that is obviously reducing costs. Another reason is that developers can be generally described as “lazy-smart” people who would not be satisfied with doing repetitive tasks day after day, so they would naturally love to automate them. Some people are even joking that we might automate ourselves out of our jobs eventually.

Measuring productivity is not an exception, today there are a myriad of platforms available for measuring individual’s efficiency based on different metrics. However, automated measuring of software engineering may not be as good as it seems at the first glance. It is a controversial idea, which can produce negative results, if misused. For example, bad metrics can introduce wrong behaviours by making developers abuse them instead of focusing on actual work.

The aim of this report is to explore the metrics, associated algorithms and platforms available nowadays as well as to discuss the technical and ethical challenges of measuring software development.

Measurable data

Software engineering is definitely measurable, because developing software is a complex process, which produces a variety of metrics. There are multiple ways in which we can classify the metrics. For example, we could divide metrics into personal and collaborative, with ones focusing on individual’s contributions and the others on how the individual interacts with the rest of the team. Another way is to map the sub-processes of software development such as writing code, planning the sprints, testing the system and operating the production system to metric classes (Sealights, 2019). Hence, we can look at developer productivity measures (time spent on coding, debugging or collaboration), formal code metrics (lines of code, commits etc.), operational metrics (mean time to resolution, downtime per year etc.) and testing metrics (coverage, whether tests are hermetic etc.).

In addition, large scale version control systems both internal and external enable organizations to retrieve and store software engineering data seamlessly. Therefore, almost all the infrastructure required is already in place and all that has to be done is collecting the data and converting it into metrics, which can theoretically be analyzed by either a human, a computer or both.

Measuring software engineering is also possible due to the fact that performance tends to vary significantly within a single team and hence there is a good opportunity to compare members of the same team with one another. It is even argued that there are 10x developers, which are people who are naturally ten times more productive than an average developer. There is also a term “net negative producing programmer”, whose removal from the team will improve its productivity (Schulmeyer, G. G.). We could think of “net negative producing programmer” as a person whose code generates more trouble than good.

We briefly touched on an extremely important matter here. At the start of this section, we tried to classify the metrics by the part of the software engineering process they originated from, be it a simplistic personal vs collaborative perspective or a more complicated attempt to map sub-processes of the development and metric classes. A missing point of view here is that we do not attempt to define team and individual performance metrics in a distinct way.

“Team vs individual” performance discussion is crucial, because teams of extremely productive individuals are not necessarily a good thing. Consider a simple example: three average performance developers with exceptional communication skills versus three high performers who have trouble collaborating with each other. It is obvious that the first team will likely outperform the second one, even though individually a developer from team one is less productive than a developer from team two (Orosz, 2021). Given that software development is majorly about people, we should think in 10x team terms rather than in 10x developer terms (BlueOptima).

Since measuring a team’s performance as a whole is less targeted than looking at individuals, leaves less room for automation and is not as cost effective as measuring an individual, because more time and resources are required, it is tempting to come up with such individual metrics that can be combined to form a reasonably accurate team metric. This complicates the interpretation of metrics significantly, because it is not possible to consider different classes

of metrics on their own - we have to combine a person's ability to write clear, maintainable code with their ability to effectively function in a team environment.

Another issue is that even when obtained and combined correctly, metrics are still hard to interpret. What would you do with information that the user on average sends 100 messages a day in slack, writes 200 lines of code (LoC) of which 100 are new and 100 are refactoring and attends 2 hours of team meetings? Is 100 messages in slack too much? Is 200 LoC enough? Does the user waste their time in meetings or contribute to them meaningfully?

Luckily, the performance in teams tends to vary a lot as mentioned before and so a solution to this is comparing metrics for a specific group of developers in a relative rather than absolute manner. However, what can one do with situations when even the relative difference in metrics is ambiguous?

For example, we have a developer that tends to receive a lot of comments on their PRs. Is that because they are working on a crucial part of the system, where a lot of input is required from the team members or is it simply because they are a new hire or an intern who is not very experienced with style guides? Hence, the context is crucial for us to be able to correctly interpret the metrics.

It is clear that assessing individual developer’s performance is a tempting thing to do and that it is possible to build measuring systems to achieve that. However, it is not always clear how to interpret the results of such measurement to come up with beneficial advice for managers and developers. The further sections of the report aim to examine what tools are available nowadays and to understand whether the interpretation is something that can be automated or will still require human reasoning.

Development analysis platforms

In the previous section, we mentioned that version control systems provide almost all infrastructure required for building measuring tools. However, once data is obtained it has to be analyzed, which can be a practical problem as there are a lot of metrics that can be collected and the collection may happen during a long period of time, yielding huge datasets.

In the past, the size of the data could be seen as a stumbling block, but with the increase in computational power and the variety of cloud services, providing infrastructure for big data analysis as a service, this is no longer a problem. What about the actual analysis though? Does the analysis software has to be written for each individual case?

The answer is no, because there are companies which provide analysis applications as a service. The increasing number of such companies makes measuring software engineering extremely easy for the managers. A typical workflow could be buying a subscription, allowing access to the codebase and waiting for a couple of days for the useful analytics to appear. This is a seamless process, which even a person with no technical experience can follow. Moreover, it is a very tempting thing to do for all the levels of organizations, as developer salaries are extremely high today, meaning that the organizations want to know if the output produced by a developer is worth the money. Let's have a look at what tools are available nowadays and briefly describe their functionality.

The first platform we would like to consider is Pluralsight "Flow" (Pluralsight). Based on the website, we see that the analysis happens for four types of metrics: code, review, collaboration and upskill, which is linked to their other platform called Skills targeted at helping developers improve their technical expertise. Let's look at the four types of metrics in more detail.

Pluralsight claims that "code" allows teams to see time spent on refactoring and new work, to recognize and eliminate project bottlenecks and get concrete data around commit risk, code churn (code that is rewritten or deleted shortly after being written) and code impact. It is also stated that code metrics for a particular programming language/technology can be linked with Skills platforms to suggest useful training for the developers. This is delivered in the form of a report which contains commit efficiency by language, Skill IQ data from the Skills platform, which is based on assessments that are completed by developers, and recommendations for team skill development. Another part of "code" is work log which helps the managers see commits and PRs in a single dashboard. It also provides information about the "day in life" of a particular engineer, by assessing their activities. Work log helps pull together artifacts produced by engineers. Finally, there are project timeline and retrospective features, which help assess the impact of organizational changes on the

development process and provide tools to evaluate successes of releases and sprints respectively.

“Review” is another set of metrics visualized by Pluralsight Flow, which help identify if code reviews are productive and positive, scan for disagreement in code reviews and observe team dynamics in code review process. Another feature of “review” is providing resolution insights based on metrics such as number of reviewers and time to resolution.

“Collaborate” aims to provide powerful visualizations of teamwork dynamics based on metrics such as percentage of PRs that get zero responses or if senior engineers provide adequate amount of feedback. “Collaborate” also features visualizing the code base knowledge distribution in a team, so that managers can eliminate the situation where everything is dependent on a small number of highly knowledgeable developers.

Pluralsight Flow integrates with Github, BitBucket and GitLab. More advanced plans also allow connecting Jira, which is an issue tracking system. Hence, the instrumentation can be automated and done remotely, if the team already uses some of the services above.

Pluralsight Flow is a powerful visualization tool, which also provides insights to the managers, however managers still need to decide how to act on the insights, which raises a lot of ethical concerns discussed in the further parts of the report.

Waydev is another platform which provides functionality similar to Pluralsight Flow (Waydev, 2021). Waydev seems to be a bit more focused on the cost analysis than Pluralsight Flow. It also features DORA metrics, which are essentially Site Reliability or more generally DevOps metrics such as mean time to resolution and deployment frequency. In addition, Waydev analyzes code of individual developers by looking at metrics such as commit risk breakdown, adhering to agile practices, for example, incremental development. PRs are used to review collaboration and the metrics mentioned here are review coverage, responsiveness for PR authors and reaction time for PR reviewers. Agile processes are also assessed, this is done by looking at cycle time and resource planning, for instance. Finally, there is a work log which offers a high level overview of the project development.

Similarly to Pluralsight Flow, Waydev integrates with common git version control systems.

Code climate is a different platform, because it provides two products: Velocity and Quality (Code Climate). These are measuring systems which are focused on two opposite aspects of software development. Velocity helps organizations ship software faster while Quality assists in shipping better quality code by automating parts of the PR review process.

Velocity is based on Jira and DevOps tools configured for a specific product. It uses the data from there to come up with insights which should drive engineering processes. Quality involves automated code review messages for pull requests and automated code coverage assessment. It can correlate code quality metrics with churn to find hot spots in development.

There are many other platforms which have a slightly different focus from one another, but the key conclusion is that there are plenty of measuring platforms available on the market and that most of them provide visualization, advice and insights, but still leave the human judgement in charge, serving as a tool for engineering leaders.

One might think that eventually such platforms may automate the managerial and people operations work, but given the algorithms available nowadays this is far from possible. The next section of the report will examine the options available today and why they are not capable of abductive human-like inference.

Algorithmic approaches

The first thing that comes to everyone's mind, when a lot of different metrics are present, is machine learning or some other sort of statistical inference. Machine learning is definitely a good technique for working with well-defined data, for example tasks of classification. However, any statistical method essentially tries to average out the data set to find general facts about the structure of the data.

Let's consider an example of a common technique known as Principal Component Analysis (PCA). Given a sample data set, it tries to accurately represent the data in a space with fewer dimensions. For instance, we have 10 code metrics and 10 PR metrics, PCA would try to come up with 5 metrics based on the 20 we have, which would represent the data reasonably well. Suppose we have a lot of people, who quickly respond to comments from code reviews

and also quickly review the PRs assigned to them, in our organization. We could describe them as productive mid-level developers. Assume now that we have a senior developer, who probably does not have time to write a lot of code and hence is slow with their own PRs but reviews a lot of other developers' PRs and mentors junior engineers. If we use PCA on our data, the new metrics that we get would probably suggest that a person has to quickly respond to code review messages if they quickly review PRs assigned to them and vice versa. Such a metric will not work in favour of the senior engineer, even though they might be an extremely important person in the team.

Perhaps, if we increase the complexity of statistical methods by, for example, creating and training a neural network that would have a number of neuron connections similar to that of the human brain, the issues like above would not arise. However, given the technology available nowadays this does not seem feasible. It is estimated the human brain has 100 trillion connections between neurons (Tompa, 2019). Moreover, neural networks are just a model for the human brain, so we cannot be sure that even at this scale they would be able to mimic the judgement of a human being.

Another way is a simple counting and aggregation approach. The issue we see here is that counting and aggregation will inevitably produce results that are closer to raw data than to human-level insights. Counting and aggregation methods could be combined with some logical engine to make inferences based on the metrics, which may be able to mimic human reasoning. Unfortunately, logical engines or expert systems are not capable of that. Firstly, there are limitations as to what can be put into their knowledge bases as logical rules. Secondly, experts may hold opposing opinions, because measuring software engineering is generally a controversial topic, hence making it impossible to derive concrete rules. Finally, logic engines are limited by the rules present in them. If they encounter a situation that they were not programmed to analyze, they will be practically useless.

In conclusion, it is not a good idea to allow algorithms of any sort to make decisions based on the data gathered from software engineers today. Perhaps, the decisions made by advanced modern AI systems are reasonable for most of the cases, but managers cannot simply follow what the tools dictate. This raises a number of ethical concerns on the degree to which measuring systems should be integrated in business and on the influence of such systems on

managers' decisions. The next section of the report aims to discuss various ethical challenges that arise and the policies that can be implemented to overcome these challenges.

Ethical Analysis

In this section we will further discuss how managers should make decisions based on the measuring tools available nowadays, whether we even need to measure software development at all and if we, as developers, are good with being constantly measured.

Previously, we clearly showed that important decisions cannot be made solely based on automation, because computer systems are not capable of handling context well. Similarly, to how we do not allow machine learning algorithms in the healthcare context to choose treatment for patients, we should not allow systems to make important decisions in businesses. In the end, who would like to be fired or moved to a different team because some system made that decision and the management blindly followed it.

Hence, we need clear policies that define measuring systems as tools and not decision making frameworks. The decisions have to be made solely by humans, while measuring tools can assist them.

Measuring software engineering is definitely something that has to be done. Due to the complexity of software development, it is hard for humans to gather and analyze all the different metrics available on their own. We definitely need a system to summarize and visualize the data well, so that we as human beings can make data-driven decisions. If we do not measure our work, we will never be able to understand if we are actually making progress.

Some people might argue that we should not measure software engineering because it leads to gamifying the workplace, making developers locally optimize their performance for certain metrics rather than trying to do actual work. We disagree with this statement because such a situation is only possible if the metrics are poorly chosen. Managers and engineering leaders have to be very careful in choosing the metrics to eliminate any chance of negative influences arising from the use of them.

We believe that developers would be comfortable with being measured, if concrete policies are in place to govern how their data is being used. Software engineering is a technical and highly competitive field, so developers naturally love to measure themselves. We can argue that almost every software engineer bragged about the amount of code they wrote in a certain period of time or the fact that their project's test coverage is extremely high at some point of their career.

Conclusion

In conclusion, we observe that measuring software engineering is a quite established practice that is used in the industry nowadays. We need to remember that any systems we create are just tools and not something that has to be blindly followed. Software engineers should be aware of what data is being collected about them and how that data is used.

Sources

1. The 10x developer myth: Why it fails to deliver meaningful gains. BlueOptima. Retrieved January 2, 2022, from https://www.blueoptima.com/blog/the-10x-developer-myth

2. Unlock the full potential of your engineering organization. Code Climate. Retrieved January 2, 2022, from https://codeclimate.com

3. Orosz, G. (2021, September 28). Can you really measure individual developer productivity? - ask the EM. The Pragmatic Engineer. Retrieved January 2, 2022, from https://blog.pragmaticengineer.com/can-you-measure-developer-productivity/

4. Schulmeyer, G. G. (n.d.). The net negative producing programmer - pyxisinc.net. Retrieved January 2, 2022, from http://pyxisinc.net/NNPP_Article.pdf

5. Tompa, R. (2019, March 14). 5 unsolved mysteries about the brain. Allen Institute. Retrieved January 2, 2022, from https://alleninstitute.org/what-we-do/brain-science/news-press/articles/5-unsolved-mysteries-about-brain#:~:text=We%20humans%20have%20approximately%2086,%2C%20our%20behavior%2C%20our%20consciousness

6. Top 5 software metrics to manage development projects effectively. Sealights. (2019, October 15). Retrieved January 2, 2022, from https://www.sealights.io/software-development-metrics/top-5-software-metrics-to-manage-development-projects-effectively/

7. Unlimited online developer, IT, and cyber security training. Pluralsight. (n.d.). Retrieved January 2, 2022, from https://www.pluralsight.com/

8. Waydev Info. Waydev. (2021, October 29). Retrieved January 2, 2022, from https://waydev.co/