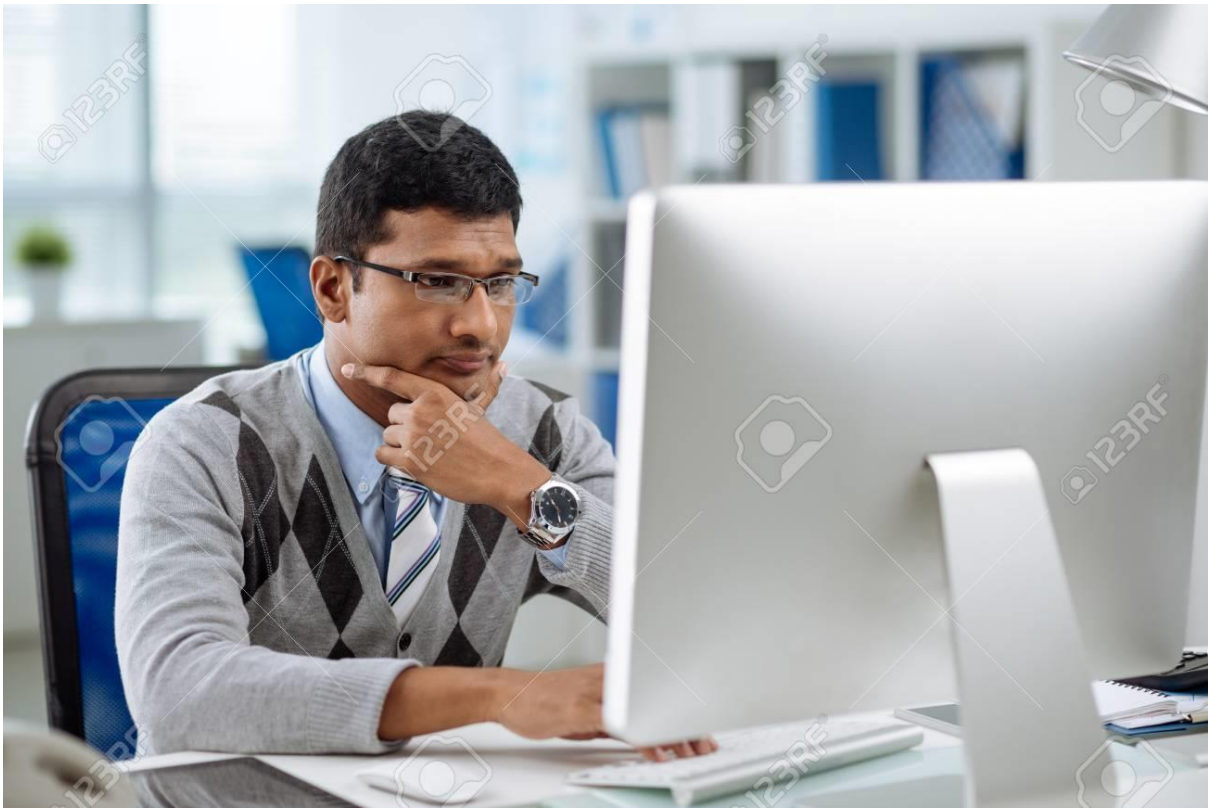
CSU33012 Software Engineering

Measuring Software Engineering



A report compiled by Tom Tye

18323900

Table of Contents:

- What ethical concerns may arise from this sort of analysis?
- Conclusion/Thoughts

Introduction

Software Engineering is an incredibly new and exciting skillset crucial to any aspiring computer scientist. As the name may suggest, the discipline is chiefly concerned with the process of designing and building purpose-built digital software, for a variety of applications; from gaming to online shopping, cybersecurity to communications, the only limit of what can be built is the human imagination (and, naturally, hardware and time limitations).

However, the overwhelming promise of this new field is tempered by some burning questions. How should the construction of such abstract systems be tackled? What are the best practices? What skills are aptly suited to tackling these challenges?

Due to just how recently the field has emerged, there is no unified idea of what (or indeed who) the perfect software engineer is. In this report, I will be taking a look at how we can measure the performance of a given software engineer, how we may meaningfully interpret any such data we are able to glean, and what tools could be used to these ends, as well as any ethical concerns that may arise from this analysis.

I hope to be able to shed some light on these topics.

How can the software engineering process be measured in terms of quantifiable data?

In all software development workflows, the beginning and end of the process is static: it starts with a client approaching the software developer with a brief

of some sort, and ends with the delivery and maintenance of the software in question. The stages in between vary from methodology to methodology, however the most generally accepted practice is the Agile workflow.

In Agile, the software company/developer works closely with the client in an iterative process: the developer keeps the client in the loop as to what they're implementing, how they're implementing it, and in turn the client let's them know any changes in design or focus that need to be applied to the next iteration. This usually continues ad infinitum, as the software developer is responsible for the continued maintenance of their code, even until well after it is implemented.

With this in mind, we can begin to look at what factors we may want to isolate which may help or hinder this process. Firstly, and most importantly, speed. It's no secret that time is money, and the more you can shorten the time between receipt of the brief and delivery of the product, the higher your overall turnover will be. Secondly, quality of the product is also very important, not just to uphold the reputation of your service, but also to minimize the amount of maintenance that will need to be performed as time goes on.

So, we know speed is important, and we know quality of code is important- but this still doesn't tell us how we can measure the productivity of a given software engineer. To this end, we can divide the factors we may want to look at into three broad categories;

- Code-Related Factors
- Environmental Factors
- Personal Factors

Code-Related Factors:

Code Quantity

Simply put, how many lines of code did they write on a specific task. This is broadly measured using two metrics;

Source Lines of Code: The simplest, and seemingly most flawed way of measuring this metric. SLOC is only concerned with how many lines were written, and as such fails to consider important aspects such as code efficiency, and the ability of the programmer to simply write more lines for a given task, making it seem like they have contributed more than they actually have.

Logic Lines of Code: This metric is concerned with the number of logical statements embedded within the code. LLOC eliminates some of the issues with SLOC, such as two different ways of formatting the same statement counting towards the total differently. However, it can still be manipulated in several ways.

Bugs per Line

Bugs per line is generally measured, ironically, as how many bugs manifest per 1000 lines. This can be used to assess the quality of the work being produced, as well as the mastery of the engineer over the tools they use.

Leadtime

More of an organizational metric than an individual assessment metric, leadtime is solely concerned with how much time it takes to go from the briefing stage to the final delivery. This is a very useful metric for measuring the productivity of an organization, however it provides almost no insight into the performance of an individual engineer.

Consistency of Code

Concerned with an engineer's performance over a period of time, this metric can be measured in several ways (some of which are expanded upon below). One way would be to consider the quantity of code being written which can be done via SLOC or LLOC, however this fails to consider important factors, such as variable difficulty of work and seasonal changes in workflow.

'Churn Rate'

Churn rate is the percentage of a developer's code that is a change applied to their recent work. A high churn rate would indicate poor quality of code, whereas a low churn rate would indicate high-quality code. This also has flaws, however, as once again the difficulty of the task and also the clarity of the brief given to the developer can both be called into question.

Bug Fixes

Fixing bugs is often one of the trickier aspects of building software. While a programmer is bug fixing, their line output will naturally be extremely low, as an intense amount of thought must go into each line. Bugs can be challenging to understand at the best of times, and this metric can be intensely hard to measure, as once again, some bugs are a lot trickier than others.

Commit Rate

How often a programmer will push their changes to the build. While this usually provides helpful insight into the speed of a programmer, it can be cheated, and as before, certain work is much more difficult, and would therefore have a much lower rate of lines written with respect to time.

Code Review

Reviewing the code of a developer is often the best way of truly assessing their efficiency and talent. The way a programmer writes code often will say a lot about how they work in a team, how much they are actually contributing to a given project, and many other useful observations. Code review, however, has two glaring faults; it takes time checking and understanding the code (which could be used more beneficially writing code or bug fixing), and it is also very hard to quantify.

Environmental Factors:

Working Climate

In order to fairly assess an engineer's efficiency, the environment in which they work must also be taken into account. Working from home vs in an office

can make a marked impact, as well as the quality of their surroundings in each of those respective areas. The human body is a sensitive machine, and requires very specific temperatures and light levels to operate at peak efficiency.

Communication

Disregarding the case of the solo developer, any good development team must have strong and efficient communication lines at all levels. Bad communication leads to wasted time trying to interpret obtuse code, unnecessary tension between colleagues, and all sorts of other unnecessary problems which could impact the productivity of an engineer.

Personal Factors:

Easily the hardest factor to measure, personal factors can play a large role in an engineer's productivity. Family grievances, personal relationships, and health concerns will of course have an impact on the performance of an engineer. However, without going too far into ethical concerns (this will be addressed later), there is only so much a company can and should be able to access this kind of data about their employee.

What computational platforms are available to gather/assess this data?

Pluralsight Flow

Pluralsight Flow is a cloud service that allows the user to gather data from whatever version control system they may be using (GitHub, BitBucket, etc.).

PF allows you to visualize how much time is spent refactoring legacy code vs. new work, recognise project bottlenecks and remove them, and get concrete data about commit risk and code churn.

It also allows you to see the pull requests of the team, and see data such as which PR's are reviewed and unreviewed, and scan for uncertainty and disagreement triggers.

As well as these features, it allows visualization of many of the metrics discussed above, although for the problems discussed earlier, these findings must be taken with a grain of salt.

CodeClimate

CodeClimate allows the user to import a repository from version control software and analyse the productivity of the contributors. By evaluating a number of metrics, CodeClimate is able to issue a 'score' whereby management teams can directly compare their teams efficiency both before and after key workflow changes to best optimize their team's efficiency.

Hackystat

Hackystat is an open source framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development processes. Hackystat allows the user to attach 'sensors' to their development tools, which feed information into a centralized server. However, Hackystat has no limits on what data it is allowed to collect, which raises some ethical eyebrows. This will be touched on more later.

What algorithmic approaches are available to interpret this data?

Machine Learning

AI (or Machine Learning) is an increasingly popular approach to tackle large amounts of data. There are 3 main methods of ML training;

Supervised Learning

By providing a large labelled dataset, an ML algorithm can be trained from it to label an unlabelled dataset. The more data provided to the algorithm, the more accurately it can label the unlabelled data.

Unsupervised Learning

Unsupervised learning involves providing the algorithm with a large amount of unlabelled data and allows it to discover correlations (given certain parameters) on its own. This method of ML learning is primarily used for purposes such as online shopping and advertising.

Reinforcement Learning

Reinforced learning is similar to unsupervised learning in that the algorithm is provided with a large amount of unlabelled data, however the key difference is that the algorithm is provided with either positive or negative feedback on its output given by an external 'agent'. The most common application of this method is in Machine Learning in games, where there is a clear preferred outcome and feedback is not subjective. This method allows the algorithm to search for the best way to solve the problem on its own.

What ethical concerns may arise from this sort of analysis

While this report has shed some light on the possible metrics and approaches that may be beneficial to the software engineering process, it is clear that there are many concerns that need to be addressed in regards to the morality of these methods.

Collecting/Analysing Data

Amidst the wave of pushes for more data security in light of events such as the creation of the GDPR (Global Data Protection Regulation) and the Cambridge Analytica scandal, it should be immediately apparent that the collection of an engineer's data is a touchy subject.

While code-based metrics are based on data the engineer has willingly given (in essence, they wrote their code knowing it would be available if not publicly, at least to their colleagues and management),the collection of other forms of data that we speculated could be of use such as data on an engineer's health and mental wellbeing are, in my opinion, crossing a line.

While it's simple to draw a correlation between an engineer's daily habits and their productivity, an employer should not be able to have the ability to influences the engineer's lifestyle outside of their work. I fail to see a future where employers have near-total control over the habits of their employees as anything but a bleak one.

Moreover, the storage of this sort of data presents a huge security risk, with even huge tech firms capable of hiring competent cybersecurity specialists falling victim to massive data leaks. The potential for abuse given data on a person's lifestyle and habits is absolutely enormous, and for this reason, I object to the collection of this data in the first place.

Beyond the collection of this data, the methods used to analyse it are also far from infallible. Machine Learning techniques are far from perfect, and their use has resulted in major scandals such as Google's image recognition algorithm incorrectly identifying people of colour as gorillas, and a self-driving car failing to detect and subsequently striking a pedestrian. Where this data is used to monitor an employee's performance, it is not so far-fetched to imagine people losing their jobs (and financial stability) due to misinterpretation of the data by the software.

Interpreting the Collected/Analysed Data

Regardless of the metrics and methods used to produce this data, it will be viewed by a human at some point and used to make decisions. It is therefore imperative that anyone with this responsibility must endeavour to fully understand the context in which the data is presented. While features such as CodeClimate's 'productivity score' may seem useful and enticing, they by definition cannot represent more human factors that may be impacting the

performance of an engineer, such as emotional state and home/social situation.

Beyond the lack of encapsulating data, it is clear that the same factors cannot be applied to all work equally- naturally, the programmer performing complicated bug fixes will be writing far less new lines than the programmer implementing boilerplate code.

For all of these reasons, while our attempts to measure the field of software engineering may indeed bear useful data, it is clear that modern methods of analysis impose significant moral questions.

Conclusion/Thoughts

In summary, although there are certainly metrics which when combined could give us an indication of overall productivity, and tools to analyse data by these metrics, we must be extremely careful and thoughtful when attempting to assign a number to the worth of a given engineer.

Even within the code metrics, significant issues and considerations arise as to how much these metrics should be used and trusted to evaluate the skill and productivity of an employee.

Moreover, the collection and analysis of this data must be carefully considered, with the risk of personal data leaks and loss of livelihood due to misinterpretation of data being very real.

In my view, there must be a clear line on how much a company should be able to monitor their employees. Allowing employers to pick and choose how they gather and assess this data presents a clear risk not just to the engineer involved, but also to working culture at large. As the pioneers of a relatively new field, it is our responsibility to ensure we create a safe and healthy working culture for future generations of software engineers.

References

hrmagazine.co.uk/article-details/the-ethics-of-gathering-employee-data

malwarebytes.com/data-breach/

en.wikipedia.org/wiki/Source_lines_of_code

projectcodemeter.com/cost_estimation/help/GL_lloc.htm

mayerdan.com/ruby/2012/11/11/bugs-per-line-of-code-ratio

stackify.com/measuring-software-development-productivity

indoor.lbl.gov/sites/all/files/lbnl-60946.pdf

pluralsight.com/product/flow

codeclimate.com

csdl.ics.hawaii.edu/Research/Hackystat

dummies.com/programming/big-data/data-science/3-types-machine-learning

en.wikipedia.org/wiki/Supervised_learning

en.wikipedia.org/wiki/Unsupervised_learning

en.wikipedia.org/wiki/Reinforcement_learning

theguardian.com/technology/2018/mar/22/video-released-of-uber-self-driving-crash-that-killed-woman-in-arizone

theguardian.com/technology/2018/jan/12/google-racism-ban-gorilla-black-people

techbeacon.com/app-dev-testing/why-metrics-don’t-matter-software-development-unless-you-pair-them-business-goals

techbeacon.com/app-dev-testing/9-metrics-can-make-difference-todays-software-development-teams