CSU33012 Measuring Software Engineering Report

Conn Breathnach 19333813

December 2021

1 Introduction

Ever since software engineering became a career path, people have attempted to measure how productive the people working in this area are. Many other professions have metrics that are used to measure productivity of workers, and these metrics are then used to make decisions regarding hiring, wages, allocating work, taking on new projects, and much more. Therefore, businesses and managers are incentivised to find which developers contribute the most to the development process, so that they can streamline the process by hiring the best developers. Defining a metric that accurately measures productivity us a "silver bullet", meaning that we have no singular metric that encapsulates all of software engineering productivity. In this report I will discuss many of the aspects involved in measuring software engineering, from different measures that can (or cannot) be used, how developer activity can be tracked, computations to be done over data collected from developers as they work, and ethics concerns involved with measuring engineering performance and collecting data on engineers. The insights that can be gained by researching this topic are certainly interesting, and exploring how to measure such a complex set of variables with no ground truth metric can lead to new developments in data analysis, clustering, natural language processing, and much more.

2 How to measure software engineering activity

There are many ways that one can "measure" software engineering activity, though just because one can measure activity in a way does not mean that they should. Having a quantifiable method to express the productivity of an engineer sounds like an incredibly powerful tool, but this tool only works if the measure used is an accurate representation of productivity. Here I will look into some of the commonly used metrics on how one can measure software engineering activity, along with the pros and cons of using these methods.

2.1 Hours worked

Perhaps the least informative, yet also seen by many managers as a golden standard, the hours worked by an employee has long been used as a metric of how productive a worker is. When viewed from a high level, it seems reasonable to equate work hours with productivity, after all if I can write one module in an hour, I should be able to write ten modules in ten hours. This way to define performance sounds fine on paper, but refuses to acknowledge that software engineering is about much more than simply churning out code. Of course, this metric is attractive to business as you can easily calculate a developer's hourly rate by dividing their salary by yearly hours worked. You can then make business decisions based on economics and money, a quantifiable measure. This once again does not provide a valuable or useful metric to define developer productivity, however. It has also been shown (Pencavel, 2014) [1] that hours worked and productivity are not linear. "The relationship is nonlinear: below an hours threshold, output is proportional to hours; above a threshold, output rises at a decreasing rate as hours increase." This further invalidates hours worked as a reliable and singular "silver bullet", and must be combined with other metrics to achieve any sort of robust understanding of your engineer productivity.

2.2 Lines of code (LOC)

One of the classic measures of software engineer productivity is to look at how much code a developer has written and to associate that metric with how productive that developer is. To someone who is not involved in the engineering process or whose only understanding of what a software engineer does is to write code, then this metric makes perfect sense. However, the process of building an application is far more complex and abstract than simply writing code, and even if LOC was a viable metric, the nature of code is that there is often more than one way to write a program. A "for loop" could easily be unpacked to make your contributions seem much larger than they actually are, but any developer who sees copy-pasted identical code will agree that this is bad code, and that the developer who wrote it is breaking many fundamental software engineering principles (Don't Repeat Yourself). Even the programming language or framework used will have an impact on the lines of code written "After all, an LOC in an assembly language is not comparable in effort, functionality, or complexity to an LOC in a high-level language." (Fenton, Neil, 1999) [2]. While LOC is an easy to gather and simple to understand metric, and code is how our systems are built, it is easy to see how LOC can easily be exploited to seem as though a developer is contributing more than they actually are. LOC also abstracts away from all the other work that goes into developing software. It looks at the end product, but does little to explain how a specific developer contributes to a project during its lifecycle. Architectural decisions, research into methods and technologies, mentoring team members, and many more important contributions a developer makes to a team are ignored when we use a metric such as LOC. LOC rewards the developers who write long modules, which may be badly structured, unmaintainable and unreadable, which are often signs of a bad developer.

2.3 Examining commits

Similar to Lines of Code, looking at the commits a developer makes to the codebase is a metric that rewards exploiting the system and only measures specific aspects of the software engineering life-cycle. Examining commits can refer to the amount of commits being made by a developer, which once again leads to an easy way to game the system for developers. Making small, frequent commits is often best practice, though once again this can be exploited into making too many small commits that are unnecessary in order to boost metrics. This also disincentives engineers from taking on larger tickets, which may be more important to the project as a whole, as spending longer to write code and make commits will make it seem as though you are a less productive engineer compared to peers working on smaller changes to the codebase.
Examining commits at a deeper level, i.e actually reading the commit messages and inspecting the changes being made to the codebase will provide much more information into how productive a developer may be, though once again it does not provide any knowledge on work done by the developer outside of writing the code. While this method is able to provide a deeper understanding of the contributions a developer is making, it also creates more overhead in the process of observing engineer productivity. Reading through code and commit messages, and understanding how that code interacts and contributes to the entire project takes time. It is also difficult to use this metric to compare developers, as you can have two developers working on completely different aspects of the project (front end vs back end), though despite a solid metric to quantitatively measure which engineer is contributing more, someone who understands the codebase should be able to see whose code provides more value (though once again, code is only one part of what a software engineer provides to a project).

2.4 Bug reports

Faults is software are expensive to fix, both economically and in terms of time, so it's within our best interests to keep bugs to a minimum. Time spent patching bugs could be better spent improving software, adding important features, or working on new projects. I think that linking productivity levels to how little time is spent fixing bugs is an understandable, but I also see how this can lead to a toxic work environment, where experimentation is discouraged and bugs are punished. Writing code without any bugs is incredibly difficult, and as a codebase grows, so does its complexity. As the software written needs to interact with more and more modules in more convoluted ways, it is to be expected that more errors will naturally occur. Tying a developer's value in terms of productivity and then using bug reports to measure that productivity will discourage people to make mistakes. This sounds like a good plan, but I believe that this can also hinder a developer's growth. If making mistakes is a natural part of software engineering, then punishing bugs will only create developers who are not willing to work on difficult code, and who will be too slow and meticulous in their development process. This may seem like I think that bugs in code are ok, and that we should not worry about them, which is not my belief at all. I simply feel that it is near impossible to create a software product without having any bugs at all in the process, and creating a culture where people are worrying more about avoiding bugs than actually building software will hamper development more than the bugs themselves would. There is a healthy medium where people write clear, concise and working code that is not sloppy, but they are not overly cautious, and are willing to test new things.

2.5 Asking developers

Perhaps the most thorough way to understand a software engineer's productivity is to ask both the engineer, and their peers working on the system to provide insight and feedback on the contributions the developers made to the system. By doing so, we can get a deeper grasp of who made what decisions during the life-cycle of the project. The developers will have a better understanding of the entire codebase, and will be able to explain each individual's role and contributions in the project. An in-depth analysis of the project is truly required to understand exactly how efficient an engineer is, and simply examining the end product is not sufficient in understanding why and how certain design decisions were made. Software development is not done in isolation, and no engineer is an island (or at least they should not be). A software engineering team should work together, and team dynamics play an important role in how successful a project is. Collaboration is key to building reliable, large scale systems and a productive software engineer should be able to provide support to their peers, contribute to ideas and planning, and write code to implement these ideas. "Put together three, high-performing developers, and with poor team dynamics and certain projects, that team might work worse than the three average ones combined. Do the same with three average performing developers, with the right dynamics and the right project, and they might do wonderfully well." [3]. Of course, asking the development team is not a silver bullet in measuring software engineer productivity either. Once again, there is no exactly quantifiable metric to measure an engineer's productivity based on this data, and much like reading through code and commits it adds a large overhead in terms of time to interview developers and gather insights into how each member of the team works. One must also consider how human bias would influence the feedback received. Engineers may exaggerate their role and contributions to the project, or downplay the influence of others in order to boost their score. Someone could also intentionally sabotage another developer by greatly downplaying the influence of their work, especially if those two developers are not on good terms.

3 Tools for measuring software engineering

Over the years many tools have emerged to gather data from engineers and aggregate this data in order to process it in a meaningful way and to attempt to provide insight into the workflow of a project. The insights generated by this kind of analysis can then be used to make informed business decisions, to understand the timeline of a project and make forecasts on when certain aspects can be expected to be completed, and to see how developer teams work together and the contributions individuals make to the codebase. These tools are built for human readability, and presenting data in a pleasant dashboard so that informed decisions can be made by managers and analysts. Machine learning is not used

3.1 HackyStat

HackyStat is a popular, though no longer under active development, tool for collecting data on developer workflow through plugins, or "sensors". These sensors collect raw data from a developer's workflow to a web server (named the sensorbase), upon which custom analysis can be performed, to provide insights into whatever aspect of developer productivity you wish to better understand. Being able to gather large volumes of data in real time and having a historic view over all this data means that managers, researchers, and analysts can reliably keep track of a projects workflow and gain an understanding as to how particular developers are performing throughout the project, and whether certain areas of the project allow for developers to be more or less productive in their workflow.

3.2 Process Mining Tools (ProM Tools)

ProM is a open-source framework for gathering analytics and data from developer workflows through the use of plugins. Developed by the Eindhoven University of Technology, this toolkit provides frameworks for both academic researchers and end users who wish to perform process mining. ProM models its data as objects, which can be filtered, enhanced, checked and have various other computations performed over them before finally providing visualization tools to display the data. Toolkits such as ProM are popular as they give managers who may not be as involved in the development process a look into how their team is creating software and how developers work within the team environment. Of course, this tool is only as powerful as the person using it allows it to be, and such having an understanding of what your data represents and being able to interpret how certain objects define a developers productivity is vital when using any tool such as ProM.

3.3 Pluralsight Flow (Flow)

Pluralsight Flow is a set of tools to manage code, repositories, and collaborative processes from within a team. Flow provides easy dashboards which aggregate information on repositories, and displays statistics on the work performed on code, the ways in which developers interact with the codebase, and the different collaborative processes happening within a team. Looking deeper into a team's dynamics, by examining interactions between developers when making pull requests or the ways a team communicates issues to one another can allow us to create a more dynamic and better environment for our engineers. By examining different aspects of data from git, rather than simply using it as a version control platform, we can gain a deeper understanding of a software engineer's performance. Flow is an especially powerful tool as it leverages the information contained in a git repository, which is the full history of a codebase. Having a centralized solution to store code (git) allows for easy access to this data and the structure of most source control solutions, which contains information on code, contributors, commit messages, etc means that tools like Flow can leverage this information to create powerful analytical engines.

4 Computation over engineer productivity data

We are now entering the age of big data, where algorithms make use of large volumes of information to make data-driven decisions and perform more accurate analyses than ever before. Using HackyStat or ProM to gather engineering data has no meaning if we cannot perform calculations and create metrics based on the data generated. Tools such as Pluralsight Flow and Code Climate Velocity provide a platform to both gather data regarding how code is written, and also create dashboards based on analysis of the data, though there are many other computations we can apply and algorithms we can implement over data to generate insights we wish to see regarding our developer productivity. Over the years many statistics-based models have shown promising results in other fields regarding the interpretation of data, and some of these methods can certainly be applied to suit our needs in understanding software engineer productivity.

4.1 Bayesian Belief Networks (BBN)

A Bayesian Belief Network is a graph network which makes use of Bayesian probabilities and the graph data type. It provides a probabilistic model rather than a deterministic one, meaning that it gives probabilities over outputs for the algorithm, rather than ground-based truths. Bayesian statistics rely on prior beliefs, meaning that we can leverage subjective beliefs we may hold to build a more accurate model. With the rise in power of computation and the sheer amount of data generated, it has become possible to create larger and larger models that make use of this data to provide more accurate results and that perform better calculations over our data. Studies have shown (Fenton et al, 1998) [4] that BNNs can be used to assess system quality when building software, which both makes use of development quality and can then be used to interpret software developer productivity, based on the influence particular developers may have over the network. This kind of network was also used (Fenton, Neil,

1999) [2] to find a cause-and-effect relationship between software modules and bugs found within these modules, i.e understanding what contributes to faults in software as it is being developed. This metric is of course important when trying to understand developer productivity, as any bugs introduced into software must later be fixed, which takes away from time developing software further.

4.2 Natural Language Processing (NLP) techniques

Over the past decade great leaps and bounds have been made in the field of deep learning. Natural language processing is a technique in machine learning where we attempt to inject linguistic knowledge into computers. Thanks to the rise of big data, new architectures, and the aforementioned leaps and bounds in deep learning, the subfield of NLP has become incredibly powerful and the programs created using these techniques have many incredible uses. In 2020 OpenAI released their most powerful text generation model, GPT-3, and then in 2021 they released Codex, an API based on similar language generation techniques but trained on large volumes of computer code. Microsoft invested heavily in OpenAI, so Codex was trained on a large corpus of public Github repositories, and thus Github Copilot was released. Copilot is an extension which can be used to generate code and speed up the development process. Aside from generating code based on context or comments, Copilot can also generate comments based on code written, or ”tell” the developer what the code in question does. This flexibility and range of applications from a single model gives hope that we can measure productivity to a better degree when examining code. While models such as this should not be trusted to build entire codebases, due to their use cases being primarily for speeding up code completion on smaller functions, developer productivity could certainly receive a boost all around in terms of writing code and documentation once these tools are robust enough and their output is properly examined and verified before deploying code written with them. It must be remembered that metrics based on analysing developer data will only understand data captured from code or tools, and ignores any other metrics, such as pair programming, mentoring, brainstorming or discussing solutions and decisions.

5 Ethics in measuring software engineering

While it may seem to make sense for managers and companies to track and measure data on their engineers in order to perform analysis on this data, it is important to discuss the ethical concerns that arise from such monitoring. Personally, I feel that only information relative to what you are looking to know should be collected, and that it is important to both make your engineers aware of the data you are collecting, and allow for them to opt out of the process without repercussions. Collecting data on employees may start with pure intentions in regards to understanding developer productivity, it can easily be led to excessive monitoring of developer workflow, and pressure can be placed on developers to never take breaks out of fear of being called out by their managers. This micromanagement will only harm developers in the long run, as stress and burnout can easily occur due to the pressure of knowing that they are constantly being monitored. I feel that while monitoring developer productivity is an interesting academic research field, we cannot trust for-profit corporations with handling this data in a way that will not be exploited. Exploiting this data could easily lead to overworked developers, unhealthy competition amongst coworkers and eventually lead to lower engineer productivity. Aside from the concerns involved in constant monitoring, I believe that an engineer's contributions and skill set cannot be defined in any easy metric, and there is certainly no objective way to measure these things. I also see no reason to force a developer to work for a specified number of hours, especially if a good amount of work has been done already in a day. If I finish my assigned tasks, adding more to my workload for the day so that I can fill out my full 8 hours of work will simply lead to worse code in the latter part of the day, and will contribute to quicker burnout as a developer and overall less productivity. Treating engineers as machines is a terrible idea, and engineering should be a creative process. We do not assign productivity to artists or musicians. Of course, when an engineer gets paid hourly for their work, we must set some standards for minimum work done, but trying to squeeze every ounce of productivity out of an engineer shows a mindset that is more focused on exploiting employees to make the most money from them rather than creating a sustainable and welcoming business environment.

6 Conclusion

Throughout this essay we have discussed many of the different aspects involved in measuring the performance of software engineers, from the different metrics we can use, to the tools that are available to collect and aggregate data from engineers. We discussed some of the various algorithms that can be used now and in future to properly understand this data. The ethics of performing such analysis over your engineers is an important yet often ignored discussion point, and I showed my concerns regarding issues arising from such monitoring. The research into this field is certainly interesting, as it attempts to define a metric that does not exist and one I do not believe can be objectively defined, since productivity has a different meaning to different people, and the hat of "software engineer" encompasses many different areas in the field, so even two engineers with the same title may have vastly different responsibilities and experiences.

7 References

1. Pencavel, 2014, *The Productivity of Working Hours* https://ftp.iza.org/dp8129.pdf
2. Fenton, N. E., and Martin, N. (1999) *Software metrics: successes, failures and new directions.* Journal of Systems and Software 47.2 pp. 149-157.
3. https://blog.pragmaticengineer.com/can-you-measure-developer-productivity/

4. Fenton et al, 1998, *Assesing dependability of safety critical systems using diverse evidence*