David Kocen
Professor Barrett
CSU33012
8 November 2019

Measuring Software Engineering Report

Introduction

Using metrics to measure employee productivity allows a company to better judge the strength of their employees at all levels. This in turn enables team managers to provide support for individuals struggling, higher level managers to improve productivity among teams, and business executives to know what company-wide changes need to be instituted. Of course, measurements can also be used to justify nefarious actions such as blindly cutting underperforming workers and instilling toxic surveillance that stifle individual creativity.

Software engineering is no exception to being able to reap the benefits of measurement. However, software engineering is inherently complex[1] and notoriously difficult to measure effectively. Many different approaches have been suggested with varying degrees of success from simply counting lines of code to complex machine learning models. This report will explore the current state of measuring and assessing software engineers. I will start with an overview of what data is currently being collected followed by the algorithms used to extract meaningful measurements from the data and the tools used to implement these algorithms providing companies with meaningful results. At the end of the report I will consider the ethical concerns surrounding the collection and measurement of software engineering data.

Measurable Data

In order to properly assess the effectiveness of software engineers specific data must be collected and measured. To be suitable for companies to use, these data points should be relatively quick to collect and capable of scaling with company growth. Generally, these measurements can be categorized as measurements of quantity, quality, and process. Quantity is the most surface level form of measurement and looks at how much or what rate something is

[1] Brooks, F. (1986) *No Silver Bullet—Essence and Accident in Software Engineering*

being done. A common quantitative measurement used is lines of code written. Simply put, a company could count how many lines of code each engineer writes over a certain time interval and compare it to some baseline to estimate engineer productivity.[2] This measurement is easy to gather but runs into several issues such as encouraging needlessly complex code to increase line count. A slightly more complex quantitative measurement would be looking at redundancy-free source lines of code that "ignore redundant parts of a system."[2]

Another quantitative measurement is to look at the number of commits. This measurement is nice because it is not language specific. Number of commits instead focuses on counting bits of progress made throughout a projects lifespan. Like lines of code, this measurement can also be gamed leading to an increase of useless commits. Other quantitative measurements that provide insight into the effectiveness of software engineering teams include the amount of hours someone is spending on one aspect of a project and number of bugs fixed.[3] Most quantitative aspects of software development can be calculated automatically. However, these measurements do not effectively represent the entire development process. Issues surrounding using these measurements will be discussed in the ethics section.

The second measurement category is quality. By focusing on code quality, companies go beyond simple counting by putting the code in context. One common qualitative measurement is defect removal effectiveness (DRE). DRE takes the number of defects found while working on a project and divides it by the total number of defects in the project. It is not possible to know the true total number of defects but it can be estimated by adding the number of defects found to the number of defects reported by users after the project's release[4]. The closer DRE is to 1 the more effective a team was at finding and removing bugs before release, suggesting better quality code is being written.

Another measurement of quality is code coverage. This measurement does not look directly at the source code but rather at unit tests. In its simplest form code coverage is a measurement of how much of the source code is ran during testing. High code coverage means more source code is being tested. Since more source code is being tested there should be fewer

---

[2] Mas y Parareda, B., Pizka, M. *Measuring Productivity Using the Infamous Lines of Code Metric*
[3] https://stackify.com/track-software-metrics/
[4] Westfall, L. (1996) *Defect Removal Effectiveness.*

undetected bugs.[5] Other qualitative metrics include measuring cyclomatic complexity,[6] runtime for specific processes, and application crash rate.

The final category of measurement is process. Many software development teams use agile and Scrum methodologies to quickly create a working product and effectively respond to changing demands. With these methodologies process is often as important as the product being developed. One of the most important process metrics is team velocity. Using Scrum terminology, "velocity represents the amount of work accomplished in each sprint expressed in story points."[7] We can think of velocity as a specific measurement of how much work is getting done over time. Velocity is particularly useful for looking at team performance. If velocity is increasing after each Scrum sprint it would suggest to management that the team is learning to work more effectively together. Alternatively, poor velocity over time suggests management intervention is necessary. Actual velocity can be compared to previous estimates to help teams better understand their actual performance.[7]

Another important measurement of process is code churn. Churn is looking at the amount of code that is being modified over time. If requirements are clear then a good project should only expect significant churn at the start of a task when an engineer is prototyping. It should decrease during actual development once an effective prototype is found. Churn can increase slightly near the end of a task when troubleshooting any remaining problems but a large increase may be a sign that significant problems still exist.[8] If churn remains consistent throughout a project's lifespan it may be suggestive that an engineer is struggling to complete a task or that requirements are too vague. Of course, churn rate is going to vary based on the specific task with harder ones always having higher churn. As with all the metrics mentioned, code churn must be put into context.[8]

Other important measurements of process include release burndown, which explores the amount of work remaining at the start of each Scrum sprint, and cycle time, which looks at how long it takes a team to create a new iteration of a product.[7] One can also get more creative with process measurements by looking at non-code related measurements such as team sentiment over

---

[5] https://en.wikipedia.org/wiki/Code_coverage
[6] https://en.wikipedia.org/wiki/Cyclomatic_complexity
[7] Mahnic, V., Zabkar, N. (2012) *Measuring Progress of Scrum-based Software Projects.*
[8] https://blog.gitprime.com/6-causes-of-code-churn-and-what-you-should-do-about-them/

time or comparing time spent in meetings versus working getting done. Though often more difficult to measure, process metrics are important for determining if an engineer or team is improving over time or stagnating.

It is worth mentioning briefly that employee data can be used that is not specific to software engineering. This includes things such as communication metadata, sentiment, time spent in the office, and even proximity to one another.[9] Because software engineers are workers just like everyone else this data can provide insight beyond engineering tasks. Looking beyond engineering tasks allow for the measurement of employee well-being and company culture, all of which impact the overall performance of engineers.

Algorithmic Approaches

Once data is collected, it must be processed using some algorithm to gain insight into developer effectiveness. These algorithms can have a wide range of complexity from simply counting data values to leveraging machine learning. Though most measurement tools will use a combination of algorithms to produce a final metric of engineering efficiency, I will be splitting the algorithmic approaches into two categories: basic comparisons and machine learning.

The most rudimentary algorithmic approach that can be used is simply taking the results of some measurement and comparing them directly to one another. For instance, a manager can see how many lines of code each member of a team is writing and create a sorted list to determine which engineers are working the most. Of course, there are a number of reasons why one engineer may be writing fewer lines of code besides being a less productive worker. Using direct comparison of one measurement at one time rarely provides meaningful insight into an engineer's actual productivity.

Adding a bit more complexity, we can look at change over time. This would involve collecting data points over a set span of time and plotting them to get a long-term understanding of engineer performance. We could then use simple linear regression to see if an engineer's productivity is improving overtime, stagnating, or decreasing. The resulting regression model could then be compared to historical trends in the company to understand the engineer's

[9] https://www.techworld.com/startups/us-startup-humanyze-is-bringing-its-employee-tracking-badges-uk-3653590/

performance in relationship to other people at the company. This simple linear regression approach can also be applied to other metrics such as velocity to see how a project overall is fairing compared to prior projects.

While the above algorithms are fine to get a basic look at how software engineering is going, they are limited to looking at metrics in isolation of one another. Software development is a complex task. It is unreasonable then that the analysis of one metric will provide a full picture. To tackle this, managers can use algorithms that combine two or more metrics. One example of this is an efficiency measurement. This approach plots data on churn rate and compares it to some measurement of throughput, such as amount of code written. By plotting these two measurements we get a more in-depth understanding of individual effectiveness. A high code throughput with little churn could mean a lot of good code is being written by the engineer while high churn and low throughput is a sign that the engineer might be stuck on something.[10] This can then be combined with the change over time algorithm discussed above to gain more insight.

Though not strictly an algorithm, managers can take a holistic approach looking at software development telemetry to create a full picture of each engineer. “Software project telemetry enables an incremental, distributed, visible, and experiential approach to project decision making.”[11] It is the responsibility of the manager to look at all these different measurements for each engineer and team to come to a final understanding of productivity. While this provides the most complete assessment for each engineer, it is difficult for humans to simultaneously keep track of many different measurements. As a result managers may resort to cognitive shortcuts such as only focusing on one or two measurements they deem most important.[12] This leads to large amounts of potentially useful data being ignored and so algorithms that leverage machine learning and big data should be implemented.

One simple approach leveraging big data is multiple linear regression. This is a generalization of the previously mentioned simple linear regression. It takes into account multiple variables and assigns weights to each one to get an overall understanding of engineer

---

[10] https://blog.gitprime.com/5-developer-metrics-every-software-manager-should-care-about/

[11] Johnson, Philip M., et al. "Improving software development management through software project telemetry." IEEE software 22.4 (2005): 76-85.

[12] Gruszka, A., & Nęcka, E. (2016) *Limitations of working memory capacity: The cognitive and social consequences*

performance.[13] For instance, a large data set could be made containing several metrics believed to impact engineer performance. Each time data is collected it will be coupled with a subjective measurement of engineer performance by a manager. Once enough data is collected multiple linear regression is run on the data set to create a model that predicts what metrics have the highest impact on engineer performance. A simple example model would be

$$Performance = 0.6(amount\ of\ churn) + 0.3(total\ commits) + 0.1(lines\ of\ code)$$

This model can then be used on new data to provide an estimate of performance.

Multiple linear regression allows a manager to look at many different measurements without having to personally balance all the numbers themselves, removing a source of bias. Bias is still present though when first creating the model because a subjective measurement of performance is needed for the sample data. Multiple linear regression has several statistical requirements that must be met, such as data independence, outliers, and linear relationships.[14] It is important to consider these requirements in order to produce a valid model.

Going beyond multiple linear regression we could utilize neural networks and large datasets by taking a supervised learning approach. Like with multiple linear regression, data points about an engineer will be collected over time and coupled with a subjective measurement of engineer performance. However, this time the data points should not be limited to a few measurements thought to impact performance. Instead it should be as much data as can be reasonably collected. The data set and performance labels can then be used to train a neural network via supervised learning. Once trained, the neural network would be used to estimate the performance of an engineer based on the variables it deems most important.[15] By taking this route performance measurements would take into account all the data collected, not just those thought to be important by a manager. Additionally, if the data is collected across projects the most important metrics regardless of the project will take center stage providing for a general measure of performance no matter what is being specifically worked on. Like multiple linear

[13] https://blog.bigml.com/2019/03/27/linear-regression-technical-overview/
[14] https://sciencing.com/disadvantages-linear-regression-8562780.html
[15] https://www.guru99.com/supervised-machine-learning.html

regression however, this approach is not void of bias because the training data still contains a subjective measure of performance for each engineer.

All of the algorithmic approaches mentioned above can be used to measure the overall performance of individual engineers and teams. While simple approaches such as one metric rankings are fast and easy to implement, the more complex approaches such as using supervised learning allow for a fuller understanding of what has the highest impact on engineer performance and would result in better managerial decisions.

Available Tools

Most of the algorithms and data collection discussed above are not difficult for individual companies to implement. Despite this, there is a growing industry creating tools to measure software engineering performance with both paid and open-source software readily available.

One of the leaders in this industry is GitPrime whose product "connects to existing code repositories and ticketing systems to provide a quantitative view of engineering workflows, collaboration patterns, and productivity trends."[16] By automatically collecting development information from project repositories, GitPrime handles all the tedious work of gathering the types of data discussed above such as commits per day and code efficiency. It also collects more Git specific measurements such as responsiveness to comments and receptiveness to input. Data is presented in clear development snapshots using reports that look at both individual engineers as well as entire teams. Coupled with nice visualization, GitPrime argues that their product will let you "say goodbye to pointless meetings" and "build a better engineering culture."[17] GitPrime starts at $749 a month for up to 25 engineers using their basic visibility package supporting up to 50 repositories and 12 months of data availability. At $899 a month companies can have up to 25 engineers using their more advanced velocity package supporting unlimited repositories and 36 months of data availability. GitPrime also offers custom plans for larger enterprises.

An alternative to GitPrime is Gitential which offers a similar product but using a freemium model. Like GitPrime, Gitential collects data from repositories to provide insights on development at both the project and individual developer level. By using "unique, language

[16] https://www.gitprime.com/pricing/
[17] https://www.gitprime.com/engineers/

agnostic algorithms" Gitential makes measuring the software development process transparent and thus optimizable.[18] Gitential's offers a free version for individuals. Companies using their services are priced on a per developer basis with $20 per developer per month for up to 30 developers or $15 per developer per month for more than 30 developers. Custom enterprise plans are also available.

There are a number of other companies out there that offer some variation on the services mentioned above. These include WayDev,[19] which offer a free and paid option, as well as CodeClimate[20] which starts at $1,759 per month for their essentials package or $2,111 a month for their professional package.

A third, and entirely open source option is Hackystat. Meant for the "collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data," Hackystat works by attaching software sensors to development tools which collect telemetry data and send it to a central database to be stored.[21] This data can be queried using online tools to gain insights. One of the main benefits that Hackystat has over the other products mentioned is that it works at the developer environment level rather than the repository level. This allows for unique measurements, such as active time editing code, that cannot be seen at the repository level. Being completely open sourced, Hackystat is free to use but requires considerably more initial setup than paid services.

As previously mentioned, there has been a general trend across all industries to collect and analyze more employee data to improve company productivity. Though not specific to software engineering, tools that process general employee data are also valuable when measuring software engineers. One example of this is StatusToday's Isaak, which is branded as AI powered people analytics. By collecting metadata about employees such as email headers, logins, calendar activity, and file activity, Isaak provides visualization for employee interaction, workplace wellbeing, and industry benchmarks.[22] This data can be used to answer important questions about software engineers such as are they actively communicating with one another and is an engineer

[18] https://gitential.com/
[19] https://waydev.co/
[20] https://codeclimate.com/
[21] http://csdl.ics.hawaii.edu/Research/Hackystat/
[22] https://statustoday.com/features/

being over or underworked. Isaak offers a free trial and then a starting price of $99 a month. There are also custom plans for enterprises.

Ethical Considerations

When taking an algorithmic approach to the measurement of software engineering, or for that matter any human process, it is important to consider possible ethical issues. These include questioning how effective an approach is, if the approach respects individual differences among humans, and whether the approach is an invasion of privacy and dignity opening the door for potential abuse.

The first consideration is questioning how effective an approach really is. The reason that there are so many different ways to measure software engineering in the first place is because of its inherent complexity. Simplistic measurements like lines of code or bugs fixed can all be gamed in order to appear productive.[23] However, the more complex a metric is the harder it is to actually calculate and the more data you need to determine it. A company could easily end up wasting time trying to measure things instead of creating good software.

I personally believe that the only way to properly measure software engineering is to take a holistic approach. Managers can start by looking at the measurements discussed above to get a general idea of where the team is but this does not mean that engineers, or any person, should be reduced to numbers. It is important that managers are consistently engaged with every member of the team. Consistent engagement through checking in, answering questions, and performing in-person reviews involving employee feedback allows for a richer, more human understanding of the impact an engineer has on a team. Humans are bad at running numbers but fantastic at recognizing social situations and group dynamics, two crucial components for engineering teams to be able to work effectively. Couple this personal approach with collected data and only then is a complete picture of productivity revealed.

The second crucial ethical issue is acknowledging individual differences among engineers. Several studies have confirmed the general belief of "order-of-magnitude differences among programmers."[24] This has led to the phrase 10xers referring to the fact that the best

[23] https://redfin.engineering/measure-job-satisfaction-instead-of-software-engineering-productivity-418779ce3451
[24] https://www.construx.com/blog/productivity-variations-among-software-developers-and-teams-the-origin-of-10x/

software engineers can do 10 times the work of an average engineer. The same has also been found for differences between teams. Because of the massive variation among engineers, forcing specific numbers onto engineers could result in inaccurate labelling of engineers.

For instance, one engineer could be very fast at prototyping a project but absolutely awful at creating polished, robust code. The engineer might simply be unable to think creatively about all the different ways code can go wrong. Another engineer however may not have the creativity to come up with a clever solution to a problem but be fantastic at bug fixing and considering edge cases. While the first engineer may write more lines of code to get the ball rolling, it is the second engineer that is ensuring the team's work reflects the company's ability to deliver quality products. Both engineers are critical to a team but their true worth will be revealed only if several different measurements are considered. To pigeonhole humans into a few specific numbers ignores individual variation. Focusing on just a few metrics will result in devaluing certain engineers and eventually homogenous engineering teams which are less effective than diverse ones.[25]

A final ethical consideration is the issue of individual privacy and dignity. With so much data being collected about us all the time this has become an increasingly contentious topic. When choosing to work for a company it is reasonable for a person to sacrifice some level of privacy. Information such as use of company accounts, contribution on projects, and time spent in the office can and should be logged to ensure an employee is not taking advantage of their employer. However, there is a certain point in which data collection is no longer beneficial leading to a toxic work environment. As humans, we all need to feel a certain level of autonomy over our actions. Studies have found that a sense of autonomy in the workplace is "associated with higher job satisfaction, commitment, involvement, performance, and motivation" while reducing "physical symptoms, emotional distress, role stress, absenteeism, turnover intentions, and actual turnover."[26] Clearly autonomy is important. As monitoring increases though, sense of autonomy decreases. It is up to the company then to discover what is an acceptable level of monitoring.

---

[25] https://hbr.org/2016/11/why-diverse-teams-are-smarterhttps://hbr.org/2016/11/why-diverse-teams-are-smarter

[26] Gagné, M., Bhave, D. (2011) *Autonomy in the Workplace: An Essential Ingredient to Employee Engagement and Well-Being in Every Culture*.

In addition to decreasing engineers' sense of autonomy, increased monitoring could lead to unconstructive workplace behavior. Consider a situation where all communications are being monitored to gain an understanding of employee sentiment because happy workers are often regarded as more productive. Engineers may begin to believe that if they do not appear happy in their communications they run the risk of being labelled unproductive. Even if management says that this data will not be used for individual performance assessment, engineers may not completely trust their leadership and begin censoring how they communicate with others. They may choose to not bring up issues for fear of appearing like they are complaining. This could lead to problems not getting properly addressed and team productivity plummeting or the creation of an unsafe work environment. There are other, less obvious examples where data collection may negatively alter workplace behavior. It has already been discussed how measuring lines of code leads to inefficient code. Measuring time spent away from the desk could result in unhealthy amounts of sitting even when no work is being accomplished. Even measuring time spent in the office may increase burnout if company expectations on expected hours of work are not clear.

As more and more data is collected to measure performance, management teams need to ask themselves what data is worth collecting and how much data is too much data. They must ensure that all the information collected is put into the context of the individual engineer. By necessity of the complexity of software engineering and the significant variation among humans, not putting data points into context puts companies at risk of losing valuable workers and generating a toxic work environment. Data is good but when it comes to measuring people it cannot be used in isolation.

Conclusion

Throughout this report I have repeatedly said that measuring software engineering is a fundamentally hard task. This means that no one piece of data will provide a clear picture on an engineer's productivity. Rather, it is up to management to collect the relevant data to try and tease out a valid estimate. This must be done without invading privacy or creating an unhealthy work environment. Any good company would consider a combination of quantitative,

qualitative, and process-oriented measurements. They will go beyond simple algorithms for comparing individuals and use a holistic approach. This could involve some form of machine learning. It definitely should involve looking beyond just numbers and having managers engage with those being measured directly to create a richer picture of performance. There are several different tools to help with this such as GitPrime and HackyStat. Which tool is used will depend on company resources.

Regardless of what approach is eventually implemented, it is crucial that companies keep in mind that software engineers are people performing a semi-creative task. There is no one correct way to solve a software engineering problem. To assume, then, that there is one correct way to measure all software engineers is simply foolish.