Measuring Software Engineering

Stephen Byrne [17324263]
CS3012

“this is not about **data** privacy, it is about **people** privacy.” - Snowden 2019.

Contents

Introduction

Measuring and assessing complex tasks is a difficult process. This particularly reigns true in the software engineering field where modern day “knowledge workers” are often measured based on historic numerical data, which may not be effective.
This report will explore the data that can be measured and assessed, an overview of computational platforms to perform this work, algorithmic approaches available and the ethics surrounding this kind of analytics.

Measurable Data

Since the profession of software engineering was born we have tried to measure it’s process. Some measurements such as lines of code (LOC) have stood the test of time and are still being used today as they were in the 1960’s. However, the assessment of data is ultimately what’s changing due to combination of multiple data sources and new interpretations of results. This section will first touch on historically used (generally numerical) data which is still relevant today, before exploring how we combine it with new more complex and subjective measurements.

Lines of Code (LOC)

Lines of code is a commonly used metric (often Kilo Lines of Code - KLOC) however, unless it is assessed in conjunction with another data type, it is effectively useless and only a loosely based productivity measure. This has led to many metrics being developed that are still used today such as time per KLOC, errors per KLOC and defects per KLOC. Although these are easy to measure they all come with their own downfalls when being assessed. A fast programmer (low time per KLOC) may make many errors and cause many defects is one example of such downfalls.

Today a commonly used metric is *effective* lines of code, or *source* lines of code, in other words, how many lines are added/removed from the production version of the product. This is a considerable improvement on before as it quantifies the *meaningful* work done by the software engineer.

Although in assessment there are still discrepancies as one developers effective lines may cause more bugs or defects than a developer with fewer effective lines, it is still an improvement on standard LOC measurements.

Effective LOC does have a problem, it is platform specific. A task that could take one line of code in a high level language could take hundreds in a lower level one, therefore their effective lines of code could not be compared. The solution here would be to only compared developers/teams working on similar languages or projects.

When we have a measure of effective KLOC we can then assess other factors based off these, such as cost per effective KLOC. This measure may provide a good general basis to cost a project if we can estimate how many KLOC the finished product should take and how much each developer will cost to produce these lines of code. Of course, this is still an estimate and there is a lot of room for error, however as manager experiences increases a knowledgeable managers estimates will become more informed leading to projections being more accurate.

Bugs and Defects

Measurement of bugs and defects is also complicated. Numerical reasoning would suggest the developer with less bugs/defects per effective KLOC would be preferable however the severity of the bugs must also be considered, which again would lie in the judgement of management. For example, if one developer introduced 100 visual bugs into a visual display, it might cause temporary unease from it's users, however it is likely the company would be able to recover. Now take a situation where a developer introduces a single bug to a different area of the same system. You may initially prefer the developer introducing the single bug when reasoning numerically, but what if this is a life support machine and the machine fails in operation in the second scenario, possibly leading to fatalities? Here we see that measurement of number of bugs or defects is not an effective assessment of the software engineering process and these must be assessed on a case by case basis. However, there is one metric which can be deduced from this problem that could be a good predictor of "reliability." We first distinguish between defects discovered at different life cycle phases, most importantly in operation. These may lead to *failures*. We can classify defects in development as less severe than failures and measure these separately per KLOC, however the situation described above are both considered failures in

operation, so a case by case basis is still needed along with the deduced metric (Fenton and Neil, 1999).

Platform Specific Measurements

Platform specific measurements could also be used and assessed, such as commits on version control systems, however here it would be wise to learn from mistakes of the past and use *meaningful* commits rather than numerical ones. Combined with other metrics (such as bugs created/removed) per meaningful commit, this can be useful.

Other

Ignoring ethical issues, we could look towards contrasting our effective numerical measurements with other measurable data. Measuring data such as age of developer, gender and race are all easily quantifiable. It is also easy to draw conclusions here when analysing contrasted data. An example of this is the older the developer the less errors per KLOC is to be expected. Based off this, in a project where failures could cause fatalities, should we hire based on age? Taking this a step further we could contrast female against male developer's and see which costs less per effective KLOC? There are numerous combinations here, some of which are largely unethical, however this data is easily measured *and* easily assessed, which could theoretically make it an effective measurement.

Testing

The testing of software is a process that is commonly measured. The common metric being code coverage (the % being covered by automated tests). Code coverage is a good metric for helping prevent failures occurring. Often developers must reach a certain coverage score before their code is permitted to be pushed into production as a safety net.

Documentation

Documentation of software is essential for the ease of future maintenance by other developers. It is also easy to measure, using metrics such as pages produced, word count or features covered. Flaws again exist when assessing the data with documentation quality being poor, however

some documentation always trumps none. Although concise quality documentation would be preferred this is again subjective to an observer of how the measured data is assessed.

Other Measurable Data

Other data that can be measured and assessed including more subjective areas such as impact on the company or value created. These can be used for both teams and personal developers, although these are extremely difficult to quantify. Other external factors which affect the software development cycle could also be measured, such as sleep. People are more likely to make errors when sleep deprived and are more productive when not. This is actually shown when we adjust daylight saving hours and the heart attack rate jumps about 25% the next day (Walker, 2017). It is evident here sleep is a crucial factor to not only our productivity but also our health. Here we could explore a plethora of personal factors affecting the software development process but some are harder to quantify and assessment changes on a case by case basis.

Computational Platforms

Monitoring employee activity during a development process is not a new phenomenon. In slavery to scientific management (Rosenthal, 2014) the author details how the variables that affected the productivity of slaves was measured. This section will cover the modern computational platforms we use to monitor software engineers and the software engineering development process.

Personal Software Process (PSP)

The personal software process is about monitoring individuals or an individual monitoring himself throughout the software development process (Johnson et Al, 2003). There are many variations available but most involve manual data collection which is analysed and continuously monitored. This is a substantial effort to collect data so we have developed platforms to aid us and make the process more efficient. One example is an alysis called PROBE,

which uses the estimated size of a system to find an estimated time of development based on past projects.

Team Software Process (TSP)

The team software process is about measuring and monitoring a team as a whole throughout the software development process. It aggregates data and allows us to take an overview of a team throughout the software process. We have developed tools for this also.

Pre Platforms

Initially manual data collection was the preferred method, as it was the only method. It required a lot of effort for collection and analysis, used mainly simple metrics and was not very private (Johnson et Al, 2003). We moved from manual collection to building tools to integrate with our development process which are detailed below.

Platforms Available

Numerous computational platforms exist to monitor both the personal software process and team software process, some of which are dedicated tools while others are for purposes such as source control but the data collected can be easily analysed and used to measure the software development process.

LEAP

Initially, the excessive overheads of collecting and analysing data was an issue. To fix the problem, “a toolkit for PSP-style metrics collection and analysis called Leap” (Jonson et Al, 2003) was developed. Adoption was low as developers struggled to switch between their regular software and the new toolkit. This lead to further systems being developed to integrate better with normal workflows.

Hackystat

Hackystat is a framework that allows developers to collect and analyse PSP data automatically, solving the problems LEAP had. It uses sensors attached to development tools that communicate with a centralised server using the SOAP protocol. Hackystat mainly focuses on individual data and its uses. This structure emphasizes privacy. It is also very lightweight as it stores as XML files rather than a backend database.

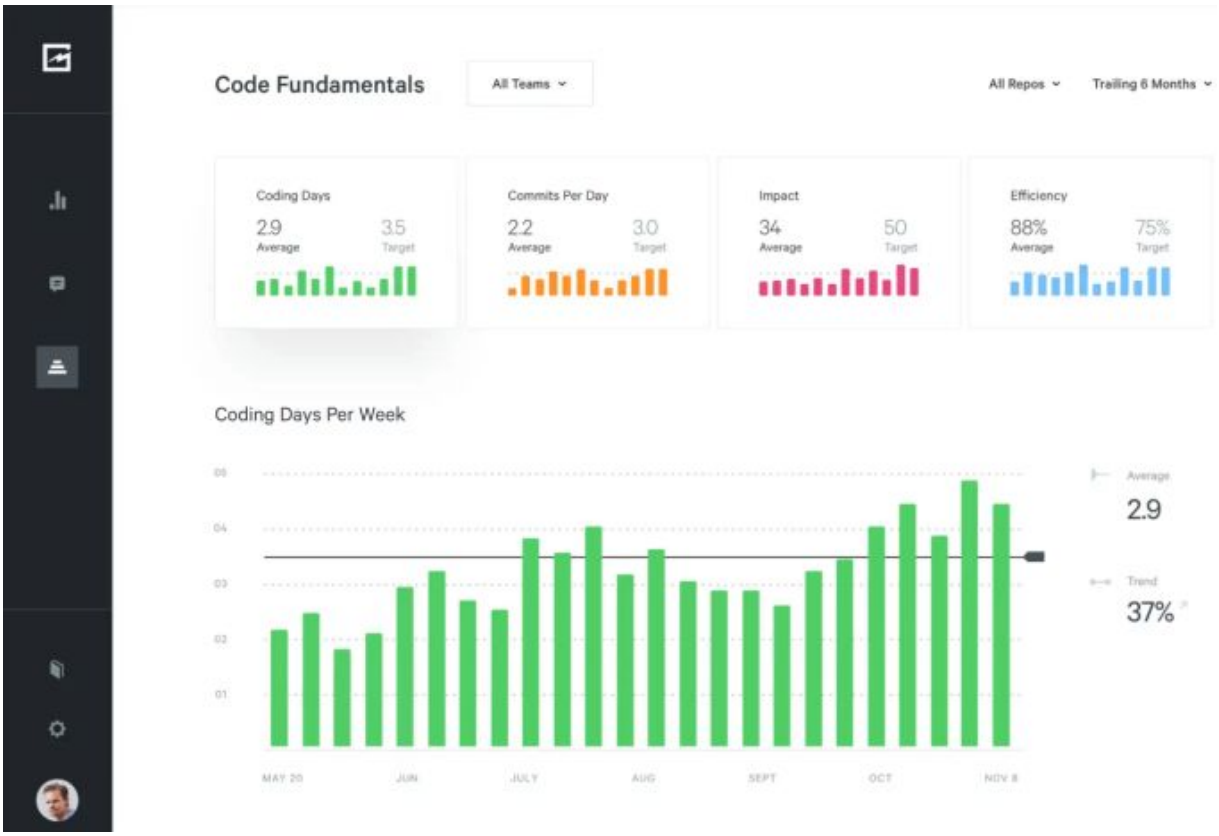
PROM

PROM stands for Pro Metrics (Sillitti et al, 2003) and is similar to Hackystat but slightly more comprehensive with an emphasis on individual privacy. It collects and analyses data at different levels, such as: personal, workgroup and enterprise. It uses personal data in a group model, protecting individuals privacy and only gives aggregated data to managers. It integrates accounting methods such as Activity Based Costing by automatically tracking data for both developers and managers. It includes a wide range of PSP metrics making it a very useful tool.

Source Control Tools

Source control tools such as Git allow data to be collected throughout the normal software development process without any additional tools. Common LOC measurements mentioned above are easily measured along with platform specific measurements such as number of commits. There are also many platforms which further analyse or display this data such as GitPrime or Gitential. These platforms allow more complex computations across many areas useful for analytics and creating visual displays of the analysed data.

GitPrime example



Algorithmic Approaches

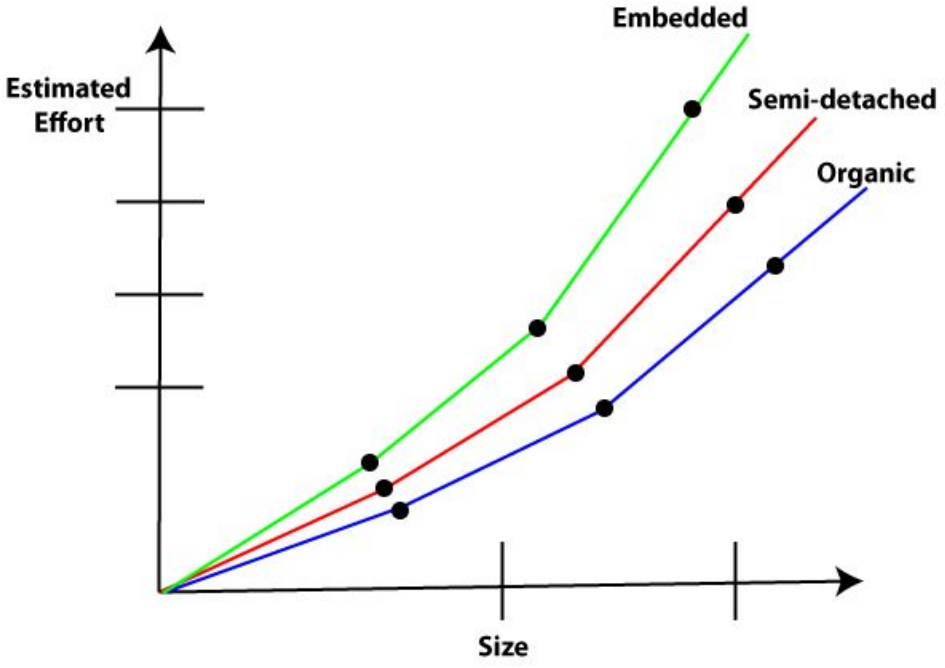
Machine Learning Algorithms

Machine Learning (ML) algorithms are particularly useful in the analysis of data in the software engineering process. Complex ML algorithms are often used for fault prediction in large software projects. They are useful in these projects as they often have a lot of good data which it can learn from. This can then be used to recognise potential future faults in new code units. There are numerous machine learning platforms out there which are improving at a rapid pace.

Algorithmic Modelling

This is the use of relatively easy to implement algorithms to predict variables such as cost. There are many models available but one of the most commonly used for cost is COCOMO.

The Constructive Cost Model (COCOMO) is a procedural software cost estimation model (GeeksforGeeks, 2019). The formula uses regression and data centered around LOC based off previous projects. It was developed by Barry Boehm. The model is based off of two parameters, effort and schedule. **Effort** is the amount of labour required to complete a task, measured in person months. **Schedule** the amount of time required to complete the job, measured in time (such as weeks).



Effort versus product size

COCOMO has three increasingly accurate models, basic, intermediate and detailed. Each is useful for different complexities of projects. Boehm also defines three kinds of systems; organic, semi-detached and embedded of different complexities. Different formulas exist for each, making it a very versatile model. It has also been adapted and altered numerous times making it one of the most researched and used models, implying that it is somewhat effective.

- **Basic COCOMO Model:** The effort equation is as follows:-

$$\mathbf{E} = \mathbf{a} * (\mathbf{KLOC})^{\mathbf{b}}$$

$$\mathbf{D} = \mathbf{c}^{*}(E)^{d}$$

Where:

E = effort applied by per person per month

D = development time in consecutive months

KLOC = estimated thousands of lines of code delivered for the project.

An example of the Basic COCOMO Model (diwan, 2019).

Ethics

The ethics of the companies which gather and analyse data on us is becoming increasingly scrutinised. Edward Snowden (2019) highlighted this recently stating that “this is not about **data** privacy, it is about **people** privacy.” Our data is constantly being collected and analysed to market us products and services which has been brought to light with various media scandals lately. But this practice is nothing new to the manufacturing world where worker

productivity has been tracked for centuries. Modern software engineers have some of the most robustly collected data due to it's easy to collect nature, but the analysis and results of this analysis is where ethical issues lie. There is somewhat of a trade off between collecting useful data, and collecting data which then impedes on a developers privacy.

Continuing on from this, the thought of collection of such robust data on individuals often makes them uncomfortable. This reigned true for Hackystat as developers found their information being collected without notifying them intrusive.

The collector may also disassociate the data with humans and may be more inclined to be unethical. This is explored in "Predictably Irrational" by Dan Ariely where we see an experiment that proves when data is not directly associated with a result, we have less of an issue with misusing it. This has lead to a move towards more aggregated team measure data being used, such as with PROM, and individuals kept private, which is a step in the right direction.

There are also laws in place regarding unethical analysis of data, such as data touched on on page 3, regarding points such as gender or race. These laws are currently the bare minimum and we need to further explore these boundaries. To keep a humane element in the software engineering process, it would be ethical and wise to only allow managers to access mainly aggregated data as a knowledge worker may have a particularly bad day and make no "breakthroughs" which could lead to what looks like unproductivity. In reality he could be stuck on a particularly complex problem and if it lead to him getting in trouble or even fired this would be wrong. Particularly with knowledge workers analysing data based on pure productivity is unethical yet this is no clear easy solution. Ultimately, we can't treat knowledge workers like machines, yet we continue to do so.

Conclusion

The collection and analysis of data during the software engineering process is at an advanced stage. Measurable data is plentiful. We have many platforms and models along with particular data types we collect which have developed over time, although some have stood the test of time and has been

used for many years. By combining and contrasting different data points we can paint a picture of the software process and use this to predict points of interest such as predictions of cost or faults, an area where this data is extremely valuable. Using the most advanced computational platforms and algorithms allows companies to gain an edge over their competition, but there must be a balance with employees personal privacy, hence the move towards aggregated data. The “what“ should we actually measure needs a valid “why”. Capable software engineers are a valuable resource in current times, meaning if they feel their privacy is being breached by one firm, they can simply move to another without much fear of becoming unemployed. This is a luxury not many other professions have. However, we must personally monitor this and ensure our data is being used ethically to help us personally and our teams improve, and not in a way that impacts our personal privacy and wellbeing.

Bibliography

- Rosenthal, C. (n.d.). From Scientific Management to Slavery | *Accounting for slavery*.
- GeeksforGeeks. (2019). *Software Engineering | COCOMO Model - GeeksforGeeks*. [online] Available at:

https://www.geeksforgeeks.org/software-engineering-cocomo-model/ [Accessed 2 Nov. 2019].
- Diwan, A. (2019). *COCOMO MODEL*. [online] Uniqueanswer.blogspot.com. Available at: http://uniqueanswer.blogspot.com/2011/06/cocomo-model.html [Accessed 1 Nov. 2019].
- Snowden, E (2019). *Websummit Opening.* Websummit, Lisbon, 4/11/2019.
- Fenton, N. and Neil, M. (1999). Software metrics: successes, failures and new directions. *Journal of Systems and Software*, 47(2-3), pp.149-157.
- Walker, M. (2017). *Why We Sleep*. [Place of publication not identified]: Scribner.
- P. M. Johnson, H. Kou, J. Agustin, C. Chan, C. Moore, J. Miglani, S. Zhen, and W. E. J. Doane, “Beyond the personal software process: Metrics collection and analysis for the differently disciplined,” in Proceedings of the 25th international Conference on Software Engineering. IEEE Computer Society, 2003.
- A. Sillitti, A. Janes, G. Succi, and T. Vernazza, “Collecting, integrating and analyzing software metrics and personal software process data,” in Proceedings of the 29th Euromicro Conference. IEEE, 2003, pp. 336– 342.
- Ariely, D. and Jones, S. (2008). *Predictably irrational*. New York: HarperAudio.