Diarmuid Coffey 17326175

Measuring Software Engineering

Abstract

The task of this assignment is to explore the various means that have been developed to measure and assess the process of software engineering, in terms of measurable data, an overview of the computational platforms available, the algorithmic approaches available, and the ethical concerns that surround this kind of analytics.

Introduction

Software engineering is one of the newest fields of engineering. The term was used for the first time at a NATO conference in 1968, when it was decided that a new, more systematic approach must be created for the development of software to combat issues in the ever-declining quality of the software being produced. It aims to ensure software meets client's requirements in a timely manner at a predictable price.

Software engineering is an inherently complex structure that does not lend itself easily to measurement. There have been many different attempts to measure the performance of a software engineer, with no official way yet adopted.

Measurable Data

The collection of data is immensely important in the development of any field. Without data, a process cannot be analysed for flaws, and the process itself can never become any more effective or efficient in terms of time or money. In a 2003 paper called Agile Methods, Alberto Sillitti says that "the productivity of very good programmers is ten times better than average." With the possibility for a tenfold increase in productivity, it is massively important to figure out what makes a great software engineer, and how to maximise their output.

Diarmuid Coffey 17326175

Lines of Code

In the early years of software engineering, the first metric to emerge was LOC, or lines of code. Lines of code evaluates a codebase according to its size. This metric was commonly used in early coding due to its simplicity. It is however fundamentally flawed, as it encourages large, bloated, overly complicated solutions instead of more streamlined, lightweight solutions, and shows no preferential treatment to code that addresses the problem in a more favourable way. As was said by Bill Gates, "Measuring [the software engineering process] by lines of code is like measuring aircraft building progress by weight". There are several well-known ways to game Lines of Code analysis, one of which is by making code artificially longer. This is most commonly done by adding extra space in functions, or not making use of specific methods designed in different languages to streamline code. The most extreme examples of this can lead to programmers declining to use loops, in favour of the longer, less efficient method that is more profitable under Lines of Code analysis. This is obviously a disastrous use of the programmer's time, with the attempt to measure productivity negatively affecting their output itself.

Number of Commits

Another metric used to measure the software engineering process is to count the number of commits to the codebase. Git, a version control system, describes a commit, or revision, as "an individual change to a file (or set of files)". However, this method fails in the same ways Lines of Code fails: the size and frequency of commits has no indication of how effective or efficient a particular programmer is. It is an effective way to measure activity itself, but conflating activity with productivity leads to a number of problems.

For example, one of the problems that Number of Commits analysis can encourage is the choosing of short-term solutions over more thoughtful, planned out implementations. As the method of evaluation encourages committing to a solution as often as possible, there is no reward in taking the time to think out any

possible problems that may be encountered as the codebase grows.

Keystrokes

And, as Lines of Code and Number of Commits analysis can encourage inefficiency, keystroke analysis is a deeply flawed way of measuring efficiency. Keystroke analysis encourages long winding solutions and discourages conciseness. The three aforementioned metrics all encourage more lines of code in a solution. As Edsger Dijkstra once said, “if we wish to count lines of code, we should note regard them as lines produced but as lines spent”. Conciseness is a feature of strong, clear programming, while all of these metrics encourage the opposite. However, trying to measure code while simultaneously rewarding conciseness is impossible.

Cyclomatic Complexity

Cyclomatic complexity is a metric that was developed by Thomas McCabe in 1976 that is used to measure the complexity of a program. In essence, it measures the number of linearly independent paths that can be taken. For example, if the code contained no if statements or conditional statements, the cyclomatic complexity would be 1, as there would only be one single path to take through the code. Cyclomatic complexity is used to measure how complicated a program is. The higher the cyclomatic complexity, the more complicated the program. It has been claimed that more complex code leads to a higher probability of bugs, although this has never explicitly been proven.

Code Churn

Code churn is the percentage of a developer’s own code that is an edit to their own recent work. It is computed by measuring the lines of code added, deleted and modified over a certain period of time, divided by the number of lines of code that have been added altogether. Code churn is a useful metric to evaluate the competency of a software engineer, as a high churn rate can mean that a developer is experiencing difficulty or is under-engaged and repeatedly going over previously finished work. Code churn can

also occur when the aims that a team have been given are too diffuse or unclear.

Test Coverage

Test coverage is a percentage figure that measures how much of the code is tested by the tests that have been written by the software engineer. A high-test coverage percentage usually signifies a lower chance that the program has undiscovered bugs in the software, and as such, can be seen as a fairly good indicator of a competent software engineer. However, test coverage would hypothetically favour poorly written code that is tested rigorously over flawless code that has few tests written. It is imperative that if test coverage is intended to be used as a metric for performance, a high emphasis must be placed on writing test cases.

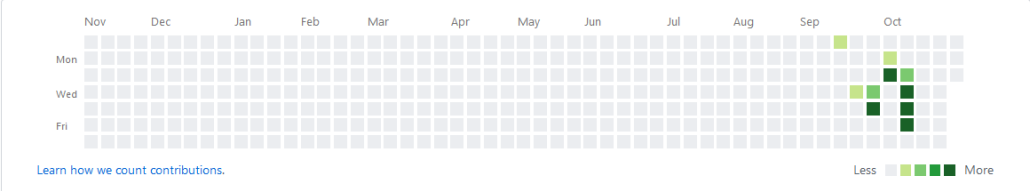
Computational Platforms Available

Once you have identified which metrics you want to collect, you find yourself confronted with your next question: how do I want to analyse it? Thankfully, there exists a multitude of platforms that can perform this job for you.

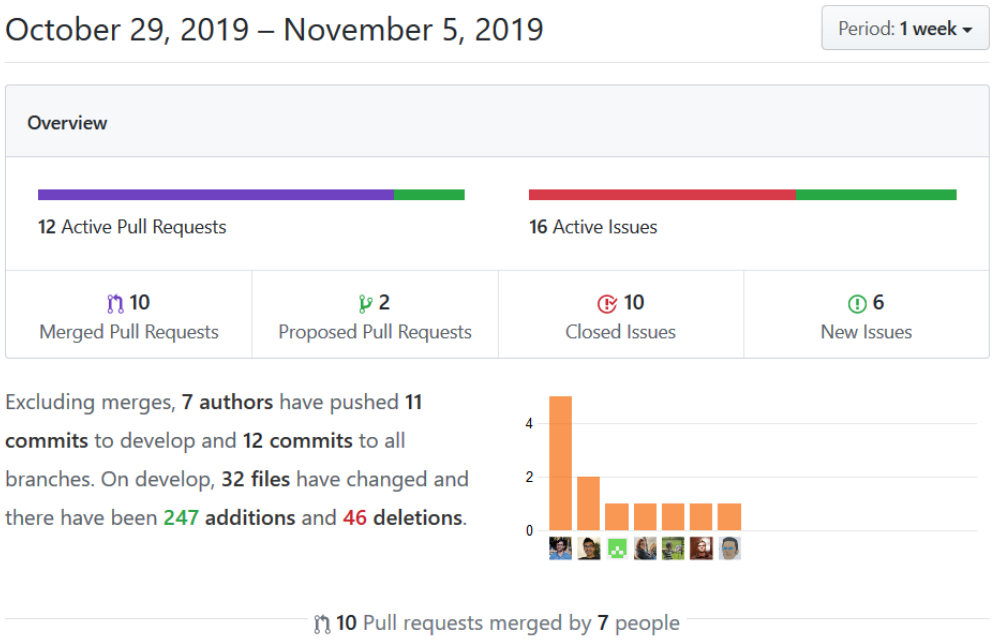
Git

Git is the first form of source control that most programmers experience. It was developed by Linus Torvalds in 2005 to maintain the open-source Linux kernel after the relationship between the community that developed the kernel and the commercial company that developed BitKeeper (another software tool used for revision control) broke down, with the tool's free-of-charge status being revoked. Git has steadily grown since then, with several other hosting services like GitHub and GitLab being built on the framework. Github in particular is a popular and powerful repository, allowing projects to be stored and worked on with ease by a multitude of people. It features some very useful statistical tools.

For example, its developer's punch card tool allows you to see the frequency of your commits. For example, here is my contributions in the last year (Bear in mind, I've only just started using Github!)



Github also allows you to analyse the statistics for particular repositories. For example, how much two repositories overlap, who contributes the most material to a particular shared repository, and so on. For example, here are some insights for a popular trending repository called blueprint, by Palantir.



The Personal Software Process (PSP) and The Leap Toolkit

The PSP platform was the first real attempt that was made to improve the software development process for individual software developers. Created by Watts Humphrey, third of his name, in the 1980's, the program was designed to work with software engineers to remove the intangibles from the software development process by tracking their goals against their progress. The platform involves the engineer evaluating their own progress using their own personal judgement. As relying on personal judgement usually ends in the introduction of bias, this platform was quickly superseded when the Leap toolkit came about.

The Leap toolkit, first proposed by Carleton Moore in 1999, was "intended to help individuals in their efforts to improve their development capabilities" by:

- preventing "many important errors in data collection and analysis",
- supporting "data collection and analyses that are not amenable to manual enactment", and
- reducing "the level of '*collection stage*' errors.

It was after the development of the Leap toolkit that it finally became clear that fully automating the PSP was an impossible task. This realisation, as Phillip M Johnson (who served on the dissertation committee for Moore's initial PSP paper) states in his 2013 paper, it became necessary to "[abandon] any pretence of supporting PSP analyses", which lead to the development of our next platform, Hackystat.

Hackystat

Hackystat grew from a desire to collect software development data with little to no overhead costs for developers, as the developers of the process were tired of the high overhead costs that are inherent in heavy manual input platforms like PSP and the Leap toolkit. Hackystat was successful in collecting workable data in an unobtrusive way, but developers found the process intrusive and violating, as they were uncomfortable with the ease of access that the platform gave to managers.

Algorithmic Approaches

The embracing of machine learning in the industry has led to some algorithmic approaches to software development analysis. The algorithmic approach allows for more data to be analysed at a faster speed, which means more concrete results can be drawn from more complex analysis.

There are 5 main components of machine learning:

- **Neural Networks (or NN):** modelled after the way biological nervous systems like the brain handle data, a neural network

has 3 main distinct sections – a section to process the information given in, a section of hidden nodes to produce an output and a section to control the type of output being produced.

- **Probabilistic Methods:** a roadmap of the process of learning, so that the algorithm can adapt to the information it has been given, much like a human would.
- **Evolutionary Computation:** Analogous to throwing mud at the wall and seeing what sticks, this trial and error style technique is very useful for handling large, complicated problems.
- **Learning Theory:** Learning theory aims to imitate the reasoning of humans by taking environmental and emotional effects into account to decide if a problem is suitable for a algorithmic solution.
- **Fuzzy Logic:** Used to model real world problems, this method can handle incomplete data sets but cannot learn like a human would.

There are three ways to train or teach a machine learning algorithm: supervised, unsupervised and reinforcement.

Supervised algorithms feature an input and an output, with a strictly defined process being used to turn the input into the output. It is called supervised as we are able to approximate the output value that a certain input value will produce. Supervised algorithms can be further broken down into regression (output is a quantifiable figure i.e. kilograms or dollars) and classification (output is a category i.e. true or false) algorithms. An example of a supervised algorithm is k-nearest neighbour, which is a greedy algorithm which takes the best-case solution at every step along the way.

Unsupervised algorithms, on the other hand, allow the algorithm free reign to find its own patterns and trends. Again, unsupervised algorithms can be further broken into clustering (bringing together similar data points to try to split the data set into groups) and

association (trying to find correlation between different parts of the set) algorithms.

Reinforcement algorithms follow the trial and error style of problem solving, which makes them very computationally intensive, like the Monte Carlo Simulations.

Ethics

From a legal point of view, as the software engineering field has developed so fast, it is currently not regulated very heavily. As such, there is not much legislation of what is illegal. One of the only things an employer is not allowed to monitor is the actual value of keystrokes, as this may contain private information such as bank details or passwords. As mentioned earlier, employers can however monitor the frequency of keystrokes in an attempt to measure productivity.
Ethically, sometimes a company's attempt to push its workers to maximum efficiency can cross the line into unethical behaviour. For example, the amount of data collected by the aforementioned platform Hackystat made developers uneasy, as if big brother was watching them. This kind of atmosphere creates a rift between the employer and employee, which is not conducive to a productive work environment.

There have also been many scholarly articles about the benefits of using wearable technology to analyse a worker's productivity using such metrics as face-to-face interaction, conversational time and vocal features. Many in the working community would find these devices a breach of personal privacy, as the worlds of work life and personal life become more and more inseparably linked.

Bibliography

https://books.google.ie/books?id=EaefcL3pWJYC&pg=PA347&lpg=PA347&dq=KLOC+history&source=bl&ots=8Ehm6Xfm9H&sig=ACfU3U3AaR1gSwlvCMa6sr9vVBDbLh_6ZQ&hl=en&sa=X&ved=2ahUKE

wi6hv3tltHlAhWhVRUIHTopBOcQ6AEwBHoECAkQAQ#v=onepage&q=KLOC%20history&f=false

https://help.github.com/en/github/getting-started-with-github/github-glossary

https://hackernoon.com/measure-a-developers-impact-e2e18593ac79

https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf

https://drive.google.com/file/d/1s7vIfuq2ARjSQLarpe0HLAHXEDskvxfI/view

https://www.guru99.com/test-coverage-in-software-testing.html

https://git-scm.com/book/en/v2/Getting-Started-A-Short-History-of-Git

https://www.academia.edu/22196770/Automated_Support_for_Technical_Skill_Acquisition_and_Improvement_An_Evaluation_of_the_Leap_Toolkit

https://ieeexplore.ieee.org/document/6509376

https://pdfs.semanticscholar.org/bfc1/480cf04a263a7a184c94578f7be5fd7f788c.pdf

https://www.repository.cam.ac.uk/bitstream/handle/1810/248538/Ghahramani%202015%20Nature.pdf

https://dspace.mit.edu/handle/1721.1/42169

Siddique, N. and Adeli, H., (2013) – Computational Intelligence. Wiley