CSU 33012 Software Engineering

Measuring Software Engineering

Emmet Morrin - 19334750

1 - Measuring Software Engineering

There have been many ways that have been used over the course of Software Engineering to attempt to measure an individual Engineer's productivity, each with their own inherent flaws.

1.1 - Code Quantity

Arguably the easiest way to measure a Software Engineer's productivity would be to count their lines of code produced. This is a simple way to predict how much effort it would take to produce this code by a Software Engineer, and plotted against time could yield a metric usable to compare two different Engineers.

This method of measuring productivity has two distinct advantages. It is very easy to calculate, meaning that very little infrastructure needs to be built before a project manager, or any parties interested can begin to see some of the metrics produced. Secondly, it is a very easy to understand method, even those not in Software Engineering would be able to understand what is actually being measured, since its very similar to grading by the length of an essay.

Just like grading the length of an essay, this method comes with some major downsides. The quality of the code is not assessed using this method, and programmers who just churn out long, unoptimized and buggy code will chart way higher than those who may take their time and produce longer lasting code.

1.2 - Code Quality

Another method of measuring Engineers, which focuses on that last part, would be to test the quality of the code being written. One of the best ways

to achieve this is to measure the amount of code that a programmer writes will be rewritten, either by themselves or another member of the team. This type of measuring would be useful for two separate reasons, the individual programmer and how the project as a whole is treated.

If many members of the team show a high rate of code reworking, the project itself may be managed following the Agile Methodology, where products are produced faster, but the process is more open to the individual customer, and will have more constant input, leading to a cycle of production, where it may take a lot longer to come to a final product, if at all.

Using this metric to assess an individual programmer may be missing some essential context by using this alone however. As above, if the project follows the Agile Methodology, then a low reworking rate wouldn't be as desirable as it may be for other projects. This may be a good indication of how buggy an individual programmer's code may tend to be, if taken with the correct context. Since context is needed to understand the implications of this metric, it would be easy for it to be misused by those who do not understand it.

1.3 - Other Methods and Conclusion

There are many other ways of measuring an Engineer's activity other than the two stated above, such as; how long they spend on a project, how many errors their code produces, how much test coverage they produce, how many, the size and the type of commits, and so on.

Each will have their own problems while trying to compare Software Engineers, but together may produce an adequate map of an Engineer which can be used by a manager or leader of a team, as long as they know the meaning and context behind each of the metrics, which unfortunately often may not be the case.

2 - Platforms Available

If a company wants to start looking into the productivity of their current and or prospective employees, the first and most obvious one would be through

GitHub, or a Git-like site. GitHub, having over 56 million users, is used widely in professional and personal settings. It allows for proper source control, code sharing for private and open source projects, feature requests and general comments, and bug tracking, among many other features. GitHub also hosts its own publicly available API, which, due to the Visualisation project, I have had the experience in working with. Because of its widespread usage and open nature, GitHub has been home to many other Software Engineering productivity metric tools, such as Pluralsight's Flow (who acquired GitPrime in 2019 for $170m).

Instead of using an already existing product, a company can use an in-house measurement method. Although it will take much longer to get setup and running for the business at hand, it would be entirely customized for that business, and if there are any issues, it should be taken care of relatively quickly since the people who wrote the code would be current or past employees. In terms of cloud storage and computation, services such as Amazon Web Services, Google Cloud and Microsoft Azure allow for an inexpensive method to store and calculate any of the data points and statistics being used to measure an engineer or team.

3 - Computation of Software Engineering Data

3.1 - Machine Learning and AI

Over the recent years, we have seen machine learning being used more and more, almost to the point of overuse, where many current "innovative" projects are done through machine learning. It's advantages are clear, once properly set up, a machine learning algorithm can very quickly evaluate a set of data inputs. The biggest problem with these methods, is that it heavily depends on the data input to train the algorithm, and can VERY easily develop its own biases.

The three main types of ML (Machine Learning) algorithms are Supervised, Unsupervised and Reinforcement Learning.

An example to explain Supervised learning would be if you wanted to train a ML algorithm to detect if there was a face in an image. You would have a very large set of images, each with or without a face, and each image is labelled as such. The ML algorithm checks each image, comes up with an answer, and is checked by the label with the image. Once a supervised learning algorithm has the required accuracy rating, it may be used on then unlabeled data to process and label it on its own.

Similarly, Reinforcement Learning also has a critiquing system of the algorithm as it "learns". This time instead of providing a data set with a correct answer, the machine's answer is graded according to another, premade algorithm based on its result. This could be used in skill acquisition of the AI, we don't know the perfect answer for the AI to perform a particular task, but we can grade how well it does, and the AI itself through many iterations can learn itself how to do the task by the critiquing algorithm.

Lastly, Unsupervised Learning is used on data which is not labeled, and is not critiqued. An AI using this algorithm may find patterns previously unnoticed, but can also often be much less accurate than other methods. Although it uses more of a semi-supervised learning algorithm, a good example of this would be if you wanted to construct an image of a person who doesn't exist (https://thispersondoesnotexist.com/). After feeding images of a very large set of people, without labeling, or guiding the results like with reinforcement learning, unsupervised learning can produce very interesting results. This method on its own can get strange and inaccurate, so it may be mixed with the other methods to make something very accurate like the website above.

For Software Engineering, ML and AI can be used to process the incredibly large amount of data points available generated due to the fact that most of what a Software Engineer produces can be recorded. On the other hand, it may be very difficult to actually train a ML system because of a lack of sample data to train it on. ML algorithms are also infamous for producing unforeseen biases. Such biases could be detrimental to those who may have an atypical work flow or working habits, but are just as valuable to the team as others who score much higher.

3.2 - Other Methods

A more traditional way of algorithmically measuring Software Engineering could look something like the Halstead complexity measures, designed in 1977. The goal with Halstead's measures was to produce a type of algorithm which could produce a number of metrics to quantify a piece of code, independent of its language. He used 4 different inputs:

$n1$ = the number of distinct operators

$n2$ = the number of distinct operands

$N1$ = the total number of operators

$N2$ = the total number of operands

Using just these 4 input metrics, from a piece of code, one could calculate the following metrics:

Program vocabulary: $n = n1 + n2$

Program length: $N = N1 + N2$

Calculated estimated program length: $\widehat{N} = n1\ log_2 n1 + n2\ log_2 n2$

Volume: $V = N \times log_2 n$

Difficulty: $D = \frac{n1}{2} \times \frac{N2}{n2}$

Effort: $E = D \times V$

Each of these metrics could be very useful to measure new individual contributions. Although being a product of its time, I think this method of attempting to measure an individual piece of code, on a function level, rather than a whole program, could be useful to a programmer.

To try it out, I used my getPieData() function in App.js from my GitHub Visualisation (linked in ReadMe) to get the following metrics:

$$n = 40, N = 121, \widehat{N} = 175, V = 644, D = 34, E = 22000$$

One of the more interesting statistics generated here, without having another piece of code to compare it to, is E, the Effort required. According to Halstead, the estimated time to write a code of this effort, should take around 20 minutes ($T = \frac{E}{18}$), which I think is reasonably accurate to what just this piece of code took me to write.

Some problems with this method would be that the amount of distinct operands and operators are not consistent across languages, for example, Python would have significantly less operators than another language like JavaScript. Even within the same piece of code, different interpretations of distinct and number of operators may result in different results.

4 - Ethics

The act of actually measuring Software Engineering, whether as a group or an individual requires a level of surveillance upon said people. This of course will bring the question of ethics and also legality into question.

4.1 - Data Privacy