Introduction

The modern route to success in all things is to plan based on measurable data. Knowledge is power. Or, to rephrase the expression slightly, the right knowledge is power. Measuring observable data is possible in so many different avenues and digital measurement of such is becoming the preferred way of doing so because of its simplicity and accuracy. People even measuring happiness with wearable tech[3]. However with such an incredibly diverse amount of data available on the new and daring digital era of the 21st century, how one know that he or she is collecting that data which will give an edge to their organisation?

To get a bit more focused, with developers leaving such a colourful and detailed digital footprint of their work, how can we use this data to help perfect the process of software engineering? What software metrics are there out there are the best indicators of an organisations' overall performance and how can they help improve it?

Perhaps, seeing such statistics motivates engineers to perform better in their work. Measuring such data and publishing it could indeed have a positive effect on engineers as many may conform to better engineering practices upon reflection of their own statistics.[2]

There is also conversely a potential issue arising about the social/ethical effect that such data could have upon engineers that do not conform with the higher-bar of the statistics. Not to mention that some engineers could perhaps not have the majority share of insertions and commits into a codebase but the insertions that were made were of a supreme quality. Such information may not come through in conventional software metrics.

In this report, I will define what a software metric is and how it should be constrained to ensure its accuracy. I will also explore conventional platforms that allow for the gathering of metrics, the analysis of such and inventive new ways in which such acquisition is happening. Finally, I will discuss the human factor involved with software metrics and how it can effect the performance of software engineers in both a positive and negative manner.

Contemporary Software Metrics

To explore existing means of software metrics, there are a series of rule to follow which we must use to make sure that the data that we are measuring on software engineers is in fact valid and what we should be focusing on. Essentially, they are constraints that need to be considered when defining software metrics. These boundaries on the measurement of software engineering data, as laid out by Cem Kaner and Walter P. Bond at the 10th International Software Metrics Symposium and the IEEE Standard 1061 are as follows:[4]

Note: " A *quality factor* is a management-oriented attribute of a piece of software that contributes to its quality ".

Correlation

The given metric should be linearly related to a certain quality factor of the software. This quality factor could be anything ranging from overall customer engagement on a platform to the efficiency and/or reliability of the software. The point is that the given input of a metric will have a direct affect on the given quality factor.

Consistency

In the given report of the symposium, consistency is defined as follows:

“ Let F be the quality factor variable and Y be the output of the metric function, M: F → Y. M must be a monotonic function. That is, if $f_1 > f_2 > f_3$ then we must obtain $y_1 > y_2 > y_3$ ”

In other words, the outputs obtained from a certain metric function much consistently correlate (see Correlation above). Examples might be the output of a team versus its size or the speed of solution delivery versus customer satisfaction.

Tracking

Essentially, a software metric fits this trait if for the function M: F → Y, as the input F changes from f_1 to f_2 in real-time, M(f) or Y should also change, So for a metric to possess this trait, a system update should yield an at least fast result. Say, a user experience update should yield an immediate positive, negative or ambivalent effect on customer satisfaction.

Predictability

This trait is fairly self explanatory. A metric has predictability if with an output Y (Y being a quality factor), the approximate value of F in M(F) can be approximated. So a certain amount of crashes in a system might be attributed to the amount of changes made to a certain codebase within the system.

Discriminative Power

In essence, this trait of a metric describes its ability to yield accurate results on the system. In layman’s terms, will the metric allow analysts to distinguish between results that speak of a negative quality factor in the system and a positive one? In other words, can a piece of high quality software (defined as a high mean-time-to-failure or MTTF) be easily distinguished from low quality software based on metric results?

Reliability

Speaks for itself. As long as the metric can consistently yield results that correlate with the constraints defined above, it can be considered reliable.

Algorithmic Approaches to Measuring Software Engineering

Now, I will discuss some of the algorithmic approaches to software metrics, their faults, triumphs and how widely used they are. Do please note however that here I have only mentioned a few methods of measuring software engineering as a process. There are in fact 100s if not 1000s of methods of measuring software engineering.

One might even argue that there is no right way of measuring such a thing. I have only chosen the ones here that appear to be most relevant. I will describe and investigate the methods below based on both my own personal analysis as well as how the approaches fit in with the constraints I have described above.

Lines of Code

This is probably the most immediately obvious metric amongst the huge arsenal out there and is in my opinion (and in many of my peers' opinions) one of the least useful. The idea is that the quality of the software is strongly tied the number of lines of code to produce the system.

The issues here are instantly prevalent as a piece of software being incredibly complex does not imply that it is robust and efficient. In many cases it is indeed the opposite. Not only this but very complicated software also leads to very complicated bugs which can lead to large slow Open/Close rates and bugs per line of code which I will discuss shortly.

From what I have said above about complexity not necessarily yielding good functionality, it is clear that measuring software in this manner does not fit our correlation trait as discussed above. It certainly isn't consistent as lots of functionality can be reduced in length by smart use of syntax and a knowledge of Algorithms & Data Structures. It is therefore also unpredictable, has a weak discriminatory power and is hard to track. Not only this but it does not account for the various means and work rates of various Software Engineers.

Lead Time

Lead time is described as the time taken to move from a bare-bones idea to a production ready implementation of the idea. For example, if one wanted to add a certain new large UX feature on a website such as a bookings manager or the like, how long will it take to go from first function to deployment.

This of course can then be applied to ideas of a larger scale too. For example the implementation of a whole platform to fit a series of use cases. This process is not so much for measuring the ability of engineers to realize a certain vision for an idea but more for exploring the efficiency of an organisation's concept to market-ready product or implementation pipeline.

With this idea of a product pipeline in mind, the metric is key for identifying blockages or stalls in the development process. For example, in a development pipeline of say **Code, Peer Review, Merge, Build, Test** and **Deploy**, there could be a lack of adequate testing in the Test phase which forces engineers to manually plug unit tests in while only making simple modifications to the codebase. Or perhaps unnecessary dependencies or a lack of modularity makes merging difficult.

The issues I have described are symptomatic of an unnecessarily long lead time. This can lead to consumer dissatisfaction with update delays or to a lack of meaningful work being done due to such blockages. Lead Time therefore is key in indicating that there may be severe blockages in a development system preventing the team/individual from working at their full potential.

I would certainly argue that lead time correlates to various features of a development pipeline being streamlined. The Jenkins' delivery pipeline is a good example of why such streamlining is necessary and is a model to be emulated by developers worldwide. It perhaps is not the most consistent as certain projects can be theoretically complex and require lots of time spent in the design phase.

Yet in general, it has a strong discriminative power and is easily tracked. With regards to predictability, a well structured development pipeline is a predictable pipeline. However not all development routes are so well mapped and not everyone has standard steps to follow when pursuing an implementation. Due to this it is, in my opinion, not predictable.

Code Churn

Code churn is defined roughly as the rate at which a certain codebase evolves. I say roughly as there are a number of means by which to approximate this that all have there individual issues. Lines of code per unit time that have been modified, inserted or deleted is usually how it is measured. I discussed earlier the issues involved with measuring lines of code as a software metric. While there are many issues with measuring insertions alone, modifications and deletions are interesting metrics in there own right.

A high amount of deletions of redundant code not only has a direct effect on the size of the application but also could have a positive effect on the speed of the system. Modifications too are an interesting metric as they could be testament to the modularity of the system.

What I mean by this is that a high amount of insertions, a low amount of modifications and a high amount of deletions have occurred it indicates that many large portions of code have had to be completely redone which is bad news. Conversely, if there is a high amount of modifications and a low amount insertions and deletions, it could be a positive indication of the overall architecture of a system.

However, besides all previous, churn rate usually incorporates all three of these measures into a combined number. Which in some ways is good as it represents how much a system has changed over a unit of time from a very high level. However I have reservations about the measure of code insertions being included in the figure in terms of how much of that new code is indeed non complex, fully-simplified and non-redundant. So, with this in mind, I feel that the definition should be considered as the rate at which a codebase changes as evolves seems to imply positive change. We are only viewing the overall change of the system. For better or for worse.

When simply considering the overall change then as discussed, code churn is does correlate with the amount of code inserted, modified or deleted quite directly and consistently does so. It is completely predictable and traceable as well. However the Discriminatory Power of the metric is called in to question as a very fast churn rate does not necessarily imply a positive change to the application.

Open/Close Ratio

Open/Close rate is a simple metric which measures the effectiveness of testing. It is defined as the the amount of software bugs found before launch/the total amount of encountered bugs post deployment. The closer to 1, the better. This is a very good metric to measure by as it is mostly related to the coverage of a testing suite within a project. It is also partially related to the variety of tests written for a certain implementation of a feature.

It is for the most part a consistently correlating metric as large code coverage in testing can bring the ratio closer and closer to 1 due to more bugs being caught pre-production. It is predictable as more tests leads to more potential bugs being trapped and easy to track. It is probably one of the most reliable metrics within this list as it seems to fit all constraints quite clearly.

Bugs per Line of Code/Churn

An easy enough metric to comprehend. For any given amount of insertions, modifications or deletions by a certain individual/team, how many bugs will be introduced. Some might argue a harsh metric to be sure. I would say it is a necessary one. If a team is constantly high up the BPLC (Bugs per Line of Code) scale with no steady improvement, clearly something is not quite right with either the tool set that the team is using or with the team itself. Perhaps inadequate testing is being done before deployment of the software or there are some internal team issues that need to be resolved.

These however are the extreme cases. No one that I know of has not ever introduced a bug into a piece of software on at least one poor commit or otherwise. It happens. It happens because we’re human. Most developers will likely use such a metric to help consistently yield higher and higher quality software than before.

This metric therefore is an essential. It is one of the most basic yet useful metrics of them all. Unlike the others I have listed, this metric offers great Discriminatory power as the individual who does not write a certain number of lines of code can be fairly compared with the developer who churns out a far greater number of lines.

For example, a low-quantity/high-quality developer who produces 1 bug in 100 lines can be fairly compared with the high-quantity developer who produces 1 bug in 1000 lines of code. It is clear that low-quantity developer simply produces more bugs than the high quantity. However in practice it will usually be the greater quantity of code that yields the greater sum of bugs. This example was only for demonstration purposes. The metric does not quite fit for the correlation or consistency traits due to the low-quantity high-quantity situation described above. However it is indeed a strong metric in its Discriminative Power as the ratio of bugs to churn yields a clear result on the consistency of a team or individual’s work.

Computational Platforms for Software Metrics

There is an ever growing list of platforms which not only enable the use of metrics but also specialize in its delivery. There are also some new and pioneering ways in which it is being done via use of a piece of ergonomic hardware to track when coding, designing or otherwise and by use of Neural Networked platforms to learn the specific patterns of development that some teams adopt.

Github

Github is probably the one of the easiest and most widely used version control services available. With over 31 million developers on the platform and more than 96 million repositories on the platform as of 2018, there is an absolute treasure trove of data on software engineering available for public viewing. The company was founded in 2008 by just three developers and was built on the fast moving Ruby on Rails framework. It has been confirmed that Microsoft will be acquiring the platform this year for 7.5 billion dollars.

Github, being the developer centric site that it is does in fact offer an API to query for data on public or private repositories (private of course will involve credentials). One can use the API to attain statistics on their own wonderful projects or set out to create a survey of the millions of public repositories that are available out there.

The API offers hundreds of endpoints for exploring various sections a repository's data. Requests to the API are queried via standard HTTP queries and all responses are returned by default within a JSON body. The fact that the API accepts HTTP requests means that software can be easily built to tailor any kind of statistical needs an organisation might need.

For example, one could easily build tools that gather simple metrics like Code Churn, commit counts or to view past merges within a repository. One only has to find the metric that they wish to measure, explore the available endpoints offered by the API and structure a simple query to be returned quality data that is well structured and accurate.

Common API calls include listing repositories of a user, all of the languages a certain user uses and how much they have really coded within those languages. This flexible and open-ended API has led to the emergence of many fascinating reports about developer practices and patterns en-masse. www.octoverse.github.com is a great example of an accumulation of such statistics. It gathers statistical data from all of github's public repositories and coagulates the data into a graphically stimulating site that is well worth the visit.

Bitbucket

Bitbucket is another type of version control software and is a major player in the sphere. It is owned by Atlassian and allows for easy integration with other Atlassian tools. The Bitbucket API offers an API that is at least as flexible as it's counterpart's.

There is very simple and readable documentation available on the API at www.developer.atlassian.com/bitbucket/api. It is clear from the start that just about any information one would want to know about any given repository is easily available via intuitively structured HTTP requests. Hook events, addons, issues, pipelines, milestones and even more are readily available via simply structured requests in your language of choice.

Interestingly enough, Bitbucket also offer a service whereby you may install your own private Bitbucket server locally using there own REST API and subsequently have complete control over your own version control. There are potentials here it seems not only that can one use the already incredibly useful data available with the API's endpoints to gather information their chosen software metrics but also that one may construct their own API endpoints to tailor their own statistical needs.

Amazon Web Services

Amazon Web Services, a venerable titan of the web hosting industry is unsurprisingly fantastic at providing metrics about cloud based software along every single step of the development pipeline. If an organisation or individual uses the Amazon CodeCommit, Code Build, Amazon Pipeline and Deployment services all in one, Amazon offers easy on the eyes visually represented data through the AWS console interface for every single service they offer.

CodeCommit, AWS' answer to version control logs all of the data that would be expected of the likes of BitBucket or Github. Examples include code churn, contribution share, test coverage and even more. All of this data is visually presented within AWS console tools but is also available via queries through the AWS command line interface tools.

AWS being the hosting platform that is is also offers valuable post deployment metrics such as the average time before failure of the service, CPU usage, memory usage and healthchecks. All of these are of course valuable indicators of the success of a certain deployment. If the deployment is unsuccessful, it will provide hints as to why it was.

Standard Testing Frameworks

Worth a mention at least is are the huge amount of testing frameworks out there for every single major programming language out there. Testing frameworks return vastly important data to developers. There are three major methods of testing software internally. They are API/UI testing, Integration testing and Unit Testing.

Of these three, unit testing is the most commonly implemented and takes the form of basic assertions to check that the returned values of internal functions are correct and if not, returns the expected value alongside the actual result. Such testing, when coverage of the tests is very high (the benefits of which were discussed earlier), proves a useful tool in tracking down exactly what is causing a specific error or a certain piece of code's failure. Test results can be of course logged as well as the coverage and amount of tests. These are useful metrics in the own right and are some of the many benefits wrought from following good testing practices.

Integration testing focuses largely on checking minor functionality of merged elements of code. Those elements being either built internally or being sourced from an external provider (such as Stripe, MongoDB, nginx) to make sure that the code communicates correctly and consistently.

API/UI testing then covers the testing of higher-level functionality which requires all base functionality to be strong and consistent. Such tests are trickier to come up and require some creativity. However all of the types that I have listed give important metrics on the overall health of the software and helps teams of developers to greatly reduce their respective Open/Close Ratios.

Scrutinizer-ci

Scrutinizer is a smart platform that helps manage software quality via continuous analysis. It also offers Integration and testing tools to assist with quick and smooth deployments. It can be built into Github and Bitbucket repositories to deliver advanced analysis and metrics. It offers specialized code reviews, analysis and testing for PHP, Python, Ruby on Rails, NodeJS, Java, GO and Docker (critiques fields within Dockerfiles).

It offers flexible and high-level means to analyse a codebase easily. Tools include Coverage Reports, prediction of Failure Conditions (to help build edge case tests), Code Rating and the aforementioned Github plug-in known as code intelligence. Scrutinizer is widely used in enterprise solutions to help ensure that in-house code is production ready and can be released to the marketplace with fears of customer destroying crashing being alleviated by standardised frameworks for code analysis of the languages which I have described above.

Timeflip.io

One of the newer and perhaps more interesting developments in the metrics pertaining not just to software developers but to office workers in general. The company produces a type of customisable 12 sided dice. Each face of the dice is covered by a glyph indicating the activity being performed. The worker simply just faces up the side which best represents the activity that he or she is carrying out while it is being carried out. The dice then links up with an application (either personal or managed by the organisation) that tracks the amount of time spent on a certain element of work.

In the context of software engineers it would provide interesting metrics as it would detail what elements of the development process took up the longest amount of time and which were the shortest. These timing metrics when paired with a software metric such as Lead Time could help an organisation to clearly identify stalls in their product development process which in turn will allow the organisation to resolve the issue an deliver their product faster.

For example, team A may recorded (based on dice data) to have spent a large amount of time debugging. The resulting action may be to focus on implementing better testing or simplifying certain complex elements of the system. Team B then say, could be noted to have spent an excessive amount of time on the design process which could then lead to an organisation hiring in more specialists of that area to help speed up the Lead Time.

The major issue that of course stands out with this nifty piece of hardware is not only that workers may forget to switch the Dice around whenever they change tasks but also that there is the opportunity for workers to be amusingly dishonest with how they are spending there own personal time within working hours.

For example, a dev might have the dice flicked up on the code insertion side for a very long time when he or she is in fact spending time debugging their own mess. Of course though, if an organisation is smart enough, simple analysis of data such as code churn for suspected employees will reveal such a ruse.

AI for Software Testing

Though it is in the early days of its use, there is as in all things these days, the potential for software that is enabled by artificial intelligence to test software packages and provide statistical data as needed. The Artificial Intelligence for Software Testing Association or AISTA for short explores and seeks to fund projects that allow software systems to write tests for themselves using AI to eliminate human negligence as well as coming up with standardized tests for AI packages and using AI to test software quality.

Well taught machine learning and neural networks can be trained to recognise poor patterns of code in many languages and can advise against it in such tests. AI systems can also be trained in generating exhaustive lists of test cases that a human simply would not even begin to think of and will in turn make for more robust systems. Such developments would lead to safer software as well as faster Lead Times in software developments. It would give engineers less to worry about in the testing sphere (aside from a higher-level supervisory role of course) and allow them to focus more on concepts of design and infrastructure. Thereby allow all of the previously mentioned metrics to be improved upon. AI testing suites would deliver consistently more and more relevant metrics on software based on developer feedback and would make for powerful aides in the overall process of development. I recommend visiting www.aitesting.org for more information on the topic.

Ethical/Social Consequences of Modern Software Metrics

Within all of the fancy platforms, methods and means of acquiring metrics on code, production level deployments, time management and so on, there is a major factor to consider amongst the cacophony of digitally acquired data. The engineers that are behind it. The people. We must ask the question: Can we really with any kind of accuracy and with no implications measure acutely the performance of software engineers?

Well, firstly, we must ask the question of how does one know that the metrics being used to measure a software engineer's work are painting an accurate depiction of the worker's habits, flaws, strengths and efficiency?

To bring up a previously used example, the high-quality/low-quantity coder may appear to be weaker than the low-quality/high-quantity coder when comparing the churn rate of the two. It is also certain that such metrics of the moment do not account for the engineer having a bad week or month thereby affecting his or her statistical performance.

Which leads me on to my next question. How can one be sure that their statistical analysis of a worker based on specific criteria and software metrics concerning the individual do not, for better or for worse, paint a false image of an engineer.
Different combinations of different metrics can produce varying results for varying workers. A well disciplined worker could be condemned in an organisation for having spent too much time debugging code instead of churning out new code. While a the developer may realise the absolute necessity of his actions, mathematical and and statistical analysis lacks the human emotion of empathy.

Furthermore, could such misdiagnoses of poor developer practice due to the inaccuracies of certain forms of measuring software development lead to animosity within an organisation? Slapping bad labels on common practices as a result of statistical conclusions can be hard for well established workers to accept. Such analysts might become the bearers of a lot of bad press in such organisations.

Yet conversely, perhaps such sweeping analysis indeed has a positive effect on employee performance overall due to budding engineers wishing to emulate the effectiveness and performance attributed to their organisations veteran engineers. A sort of healthy competition based on personal statistics would perhaps ensue and make for faster Lead Times for an organisation as well as a higher standard of coding being implemented.

An organisation with healthy internal competition such as this would discourage “bad behaviour” in software development not in a negative manner but in a constructive way which helps build the skills of new and old coders.

All of this being said, it is a brave new world of analytics where data is sacred, delicate and private. Companies and organisations investing in large scale means of acquiring software metrics should always consider the impact not only the potentially positive impact it would have on delivery times and open/close rates but also on the potential social situation of the organisation and any ethical implications it may have.

CONCLUSION

The social and ethical risks to modern software engineering are many. The generation of animosity, false labelling of actually decent developers and certain metrics detailing how apparently flawless some engineers' work is compared with others (Bugs per Line of Code). Organisations should gather such data on their developers responsibly and in a constructive manner.

Nobody wants a future for software engineering where their merit is based on there metrics alone. Such a thing should only be a part in the overall make up of the engineer. Design thinking, creativity and an ability to innovate are human traits that are unquantifiable and yet still contribute massively to workflow.

Certain employees of a company could play an integral role in the organisation by spending most of their time helping other developers create new features or infrastructure within a system. Such a thing is poorly reflected within digital media.
In short, organisations need to pick and chose the metrics they choose to base their decisions on very carefully indeed. Whether those decisions are internal or external, they require the digital equivalent of foundations of stone. That is, well founded, clear and useful data.

Looking to the future however, I cannot forsee that many organisations will have such an attitude as described and I fear that a hard-line approach, so to speak, may become commonplace within the workplace.

Perhaps in the not-so-distant future, is such a reality comes to pass, there will be a larger focus on the anonymity of such data with regulations to protect anonymity even while an employee is still actively working at a large scale organisation. Almost a sort of advanced version of the "right to dissappear concept" with the difference being that you may do so at any time even when on contract. It may more aptly be put the "right to dissappear in plain sight".

I digress. However there could be collective benefits to be reaped if such data is gathered anonymously and returned to a team or organisation as a whole. It could strongly drive a communal attitude and lead to increased performance and possibly pride in the work that a team collectively produces rather than drive envy for an outstandin developer,

In all this, one thing is for certain. Software metrics is here to stay and this is only the beginning. It is certain that such analytics will drive standards of engineering up worldwide and help make technological leaps and bounds more frequently in the future. I can only hope that this revolution takes place with the human factor constantly in mind.

Sources

1. McKinsey Global Survey Results, http://www.nextlearning.nl/wp-content/uploads/sites/11/2015/02/McKinsey-on-Impact-social-technologies.pdf

2. Network Effects on Productivity, https://www.researchgate.net/profile/Jan_Sauermann2/publication/285356496_Network_Effects_on_Worker_Productivity/links/565d91c508ae4988a7bc7397.pdf

3. Hitatchi on statistics from wearable tech, http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf

4. 10th Software Metrics Symposium, http://www.kaner.com/pdfs/metrics2004.pdf