Measuring Engineering - a report	To deliver a report that considers the ways in which the software engineering process can be measured and assesed in terms of measurable data, an overview of the computationial platforms available to perform this work, the algorithmic approachs available, and the ethics concerns surrounding this kind of analytics. Reading list available here. 10 pages.

Measuring Software Engineering Report

Paul Molloy 15323050

In this report I will consider the various ways in which the software engineering process can be measured and assessed in terms of measurable data. I will also outline some of the current computational approaches available to perform this work. I will also give an overview of the different algorithmic approaches available today for assessing the software engineering process. Finally I will discuss the different ethics concerns surrounding this kind of analytics.

Methods for Measuring and Assessing Software Engineering

In this section I will give an introduction to and analysis of several different ways in which software engineering is measured today.

Traditional Metrics

Undoubtedly the most ubiquitous metric in software engineering is Lines of Code (LOC), but yet its popularity does not necessarily correlate with its usefulness. According to *Software metrics: successes, failures and new directions Norman E. Fenton, Martin Neil* There has been negligible evidence that LOC correlates with higher fault density or even number of faults in software at either pre-release or post-release. Other similarly crude metrics that we see regularly such as size counts and man hours and other time measurements have similarly not been found to be able to predict the quality of a software system.

Another common metric used in measuring software engineering is cyclomatic complexity. This is a simple measure of the complexity of a piece of software eg. a program with more nested loops and if statements will have a higher cyclomatic complexity. Cyclomatic

complexity also has not been found to be a better predictor of failure prone modules of software than size and defect counts.
The reason why these metrics are used so commonly in software engineering today is that they are some of the lowest effort metrics to use. For example lines of code is displayed prominently somewhere on every IDE and version control system. All of these metrics can easily be calculated by running an analyser on the code e.g. cyclomatic analysis.

Code churn is another common metric that is used to measure the rate of change of an engineers code. It is normally measured in LOC over a short period of time. Naturally code churn is high at the beginning of the development process, but it should trend downwards as a software engineering project progresses. If there is a sudden spike in this metric then this can be a warning sign that something is going wrong in the development process. For example some feature may be repeatedly being changed without making much progress towards solving the problem, this metric can help identify this problem in the software engineering process of engineers or in an organisation.

Code coupling is a measure of the quality of software engineering that helps in the understanding of how well structured a piece of software is. As a general rule all of the elements of a piece of software that are closely related to each other should be structured so that they are in the same module or a class in the software. This means that there is high cohesion in the code. Designing a piece of software in this way will result in generally less coupling that is there will be less dependencies on different parts of the software on each other. That is more of the modules of the software will be independent of each other. On the other hand if coupling is high then this is an indicator that the software may not be optimally architected.

Another major idea in the area of measuring Software engineering is Personal Software Process or PSP. The philosophy is that each developer fills in forms to document their process of doing development work. For example they would fill in time and size complexity estimates, a time recording log of how they spend all of their time doing development work, a defects log of every error in their code as they discover them, even including syntax errors. It is clear that there is a huge overhead of recording work added to a developers day by following the Personal Software Process fully and that is why it is not used in many companies today.
It is argued in *Searching under the Streetlight for Useful Software Analytics Philip M. Johnson, University of Hawaii at Manoa* that as it is more difficult to get this data as it is more labour intensive that there are more valuable insights to be gained from it in analysing software development. It is also argued that the methodology is flexible as a developer or organization can modify PSP to measure specific areas of the Software engineering process that are more relevant to them, the example given in the paper was measuring disruptions to work. They simply need to record this data in a spreadsheet as they would any other metric that they are collecting as part of the Personal Software Process.
On top of the time cost of manual entry of PSP data into spreadsheets another issue with Personal Design Process is that it is prone to inaccuracies in data as it is manually inputted. There is also evidence that much of the data collected such as number of defects from a defects log in pre-release has no relation to the number of defects in post release.

The quality of Testing and test coverage on a piece of code has been found to correlate with a decreased number of faults in release code. The use of metrics to measure the coverage of testing on code is another valuable metric that can be measured. A much more valuable indicator good software engineering would be the quality of the testing performed on the source-code rather than just the code coverage of the tests.
One way to get a measure of test quality is using mutation testing. Mutation testing is an automated way of evaluating test quality by mutating small parts of the code effect by the tests, for example replacing an == comparison in the code with a != , or changing a variable's value in some way. If these mutations cause a test to fail then this is a good indicator that the test quality is high and so the quality of the code more likely to be defect free. Code mutation testing is used extensively in Google to check Test quality before commits are sent for review.

Of course coding is only a small part of the software engineering process and there must be many other ways of measuring software engineering other that just measuring the code output of the process. Both the written documentation and presentations from meetings contain valuable insights into the quality of software development. There are many other opportunities to glean metrics from all other aspects of the work of software engineers such as in person communication between developers as both formal and informal discussion of software engineering problems occurs in the workplace. If the quality of this communication can be measured then this can form part of the measurement of overall quality of software engineering. *(Software Intelligence: The Future of Mining Software Engineering Data Ahmed E. Hassan, Tao Xie).*

Computational Platforms for Measuring Software Engineering

There are many computational platforms available for the measuring of software development. Hackystat is one such platform which adds instrumentation to tools used in software development to generate metrics about the software development process of engineers. The intention of it is to measure performance with no added development overhead as all of the data is collected passively as the the developer works. This has a clear benefit over the manual record keeping required for PSP. For example in the case of Java developer the Eclipse IDE would be instrumented to record keystrokes of which method is being edited, when files are saved, every time the program is run, every time the JUnit test suite is run and a multitude of other data-points in the background.

Hackystat sends all of this data to a data repository whenever the developers workstation is connected to the internet. As the data is automatically and passively collected it can be collected constantly and in real time which means a much more accurate picture of the

development process can be collected than with manual methods. Hackystat also integrates data from all developers in an organisation together so that insights about teamwork and synergies from working together on the same files can be analysed. It also allows problems caused by the interaction between developers on the same piece of the code base to be identified.
As well as these client side developer tool metrics, code analysis tools are also run on the server side to determine the overall code health of the piece of software.

There are also several platforms that hope to go further than simply analysing the actual process of writing code but also the social interactions which are as important in the development process. One such platform has been developed by humanyze which uses IOT connected employee badges to track employees. The badges have two microphones which do real-time voice analysis of employees as they work and sensors to detect movement and location throughout the work day.
On the company's website they advertise that their platform helps employers to measure effectiveness of diversity & inclusion programs by measuring the integration of new employees by recording who they talk to and for how long and how much they are listened to by others by measuring the tones of voice as well as location and proximity to others.
The website also describes using the platforms ability to measuring the amount of communication and information sharing between employees to identify how top teams and leaders in the organisation collaborate to give insights on how to improve other teams. It is also able to detect if there are some high impact employees that the organisation is dependant on.

In the past essentially all of the focus in measuring software engineering was placed on the physical software development and analysing the code that was produced. This was primarily because it was so much easier to measure and generate metrics from the structured data that can be gathered by code, bug reports and other artifacts of development on computers. *(Software Intelligence: The Future of Mining Software Engineering Data, Ahmed E. Hassan, Tao Xie)* It is only recently that technological advances allow unstructured data such as in person conversation and location data in the workplace to be used to evaluate the quality of software development of developers.

These kinds of insights are hugely valuable for measuring software engineering as a huge amount of the work of a developer is communicating with other developers in meetings and also informally “at the water cooler”. This is where many of the design decisions for a piece of software are really made. This means that the quantity and quality communication during the software development can now be quantified in a way that it never could before.

With humanize the interactions and relationships between different employees can be analysed in such a way as to create a graph of employees can be made so the most critical employees can be identified and rewarded and likewise developers who do not seem to contribute to discussions in development and behave in an isolated can also be identified for improvement. This is important as developers who are not communicating with the rest of the team they are more likely to diverge from project requirements and have more faults in their code.

Another platform that can be leveraged for measuring software engineering is through gamification of software engineering. Gamification is the process of adding game design elements and rewards to non-game contexts. Through gamification developers can be motivated to do their development work more effectively by rewarding some positive behaviours in the software development process. Leveled achievements can be used to promote individual and team performance in some areas of the software development process such as increasing test coverage of code or increasing the number of iteration cycles a development team completes. In “Turing Real-world Software Development into a game” (Erick B. Passos et al.), achievements such as “Clockwork developer : Number of development tasks completed in a planned time frame.” with bronze, silver and gold achievements were used to help improve software development in real production software development teams. Developers were found to be motivated by these achievements to be more productive in their work and development goals were met at a greater rate.

This gamficiation platform could be further extended to add RPG like elements to software engineering. The researchers identified that they could use more of the detailed metrics like those collected by Hackystat which were already being collected by the management of the software team taking part in the research. These metrics could be used classify developers into character classes such as “master-tester” in this RPG system. They could then potentially gain experience to level up their character and possibly gain unique achievements. Another idea posited was to have immediate feedback for the developer as they made progress towards an achievement. I found this creative use of metrics very interesting, as I think it has potential create positive feedback loops of software engineering measurement as metrics are displayed in a more fun way and then this could lead to motivating developers to continue to improve their process in an interactive way, much like the software development process itself.

This kind of gamification can also foster competitiveness between developers to see who can reach higher achievements. This is seen in the research paper “It was a Bit of a Race: Gamification of Version Control” (Leif Singer et al.) Which experimented with giving positive feedback messages on a custom platform for undergraduate students designed for keeping track of progress of a group software engineering project. The platform linked with their groups Subversion repository and gave positive feedback emails to the students when a team reached a milestone such as 1000 commits. It also allowed students to view the number of commits other students made.

This led to students pushing better, cleaner commits which just changed one thing rather than large commits which changed many things. Some students became competitive using the platform and competed to see who could reach certain milestones more quickly. This gamification helped to increase the engagement of students in the software engineering project, although some students were discouraged by the competitive nature of the platform.

Algorithmic Approaches to Measuring Software Development

There are two main categories of algorithmic approaches to measuring software development, traditional code quality algorithms such as cyclomatic complexity algorithms and Halstead complexity and machine learning methods which have much broader applications and potential.

Code complexity algorithms such as Halstead complexity and cyclomatic complexity aim to measure the overall complexity of a piece of software. With the given axiom that the simpler a piece of software is and the easier it is to understand the better engineered it is. The thinking is that even if the code performs better or is a smarter solution if it is difficult to understand then defects and bugs in the code will be much more difficult to understand.

Using the Halstead algorithms by measuring the number of distinct operators and operands, and the total numbers of operators and of operands, measures for program length, volume, difficulty and effort can be calculated. Measures for time required to program, and number of delivered bugs can also be derived from these measures.

Cyclomatic complexity is another measure of the complexity of a piece of software. Specifically it is a measure of the number of linearly independent paths through a program's source code. It is calculated using a control flow graph of a program. Nodes represent indivisible groups of statements in a program. Directed edges connect two nodes if the code in the next node can be executed immediately after the previous node. Cyclomatic complexity is calculated by subtracting the number of nodes from the number of edge in the the graph and adding two times the number of connected components. It is given by the equation.

M = E - N + 2P
Where:
M = Cyclomatic complexity
E = Number of edges
N = Number of nodes
P = Number of connected components.

The value of these software complexity algorithms is disputed. Research has shown that Halstead and Cyclomatic complexity correlate with program size i.e. Lines of Code. It has also been shown in research that lines of code is often just as effective of a prediction of defects in a piece of software. Perhaps the extra computation of calculating these complexity metrics is not worth it.

Earlier in this report I discussed the value of mutation testing in evaluating the quality of testing and by proxy the quality of software engineering. The algorithms used in mutation testing are very important due to performance issues of having to run the tests on a piece of code potentially a great number of times to do thorough mutation testing. The performance of effective mutation testing has been a major barrier to widespread adoption in adoption. It is also quite a difficult problem to algorithmically identify all possible points to mutate the code to test the unit tests in an efficient way to keep mutation testing performance to acceptable levels.

Machine learning methods are being increasingly used to measure software engineering as the technology becomes more widely available. The smart employee badge company humanyze mentioned earlier in this report uses machine learning extensively to obtain insights into employee communication from employee audio conversations and employee location. this unstructured and unlabeled data is passed into a machine learning models. A machine learning model detects the tone of voice of wearers of the badges. Another machine learning model could be used to create a model of who is having a conversation with who at a given point using the tones from the previous model and location data. These models are trained to decide how employees are interacting with the team and how they are communicating.

When these models are designed they are then trained with a large volume of training data with labeled results so the machine learning model will learn how to make predictions correctly over time. For example for tone of voice recognition the model would be fed a database of voice recordings of people with different tones of voice along with a categorisation done by a human e.g. aggressive. Eventually with enough training data the machine learning model approaches the ability of a human at this task.

Machine learning algorithms are used in many other ways to measure software engineering. One example of this is measuring software with Bayesian networks. Machine learning with Bayesian networks and activity-based quality models can be used to assess software quality (Wagner). By inputting metrics about the source code and facts about the software process and team for a of a software engineering project (e.g. A NASA space-craft instrument in C) the model outputs the difficulty level of Code Reading, Modification or Testing on this piece of software. The model can give a software defect removal cost in developer hours.

Ethical Concerns

It is clear that there are many possible ethical concerns with the numerous methods and algorithms for measuring software engineering which I have outlined earlier in this report.

People have a right to privacy even in their place of work. With each of the different ways of measuring software engineering I have talked about so far the use of metrics has the potential to reduce the privacy of an engineer. If people are able to look up details about how they work that is encroaching on their privacy.

On Github today anyone can go and look up a user and look at their work and see statistics about how much they are working. Data can be gathered about how much a user is contributing to a project and what times they are doing their work. Much more detailed and fine grained data about how engineers do their work can be obtained by querying the Github api.
There is a large difference between gathering metrics from the code produced by people and actually measuring the process that developers go through as they produce code. This is especially true when the developers are not actively supplying this data but having their activity passively monitored by computer systems.

The platform Hackystat is one such example of this. Since it monitors many aspects of how people develop such as time spent on each part of the code, number of builds and runs of the test suite from monitoring add-ons to the IDE, many developers are not comfortable with management accession such data, despite promises that it would be used appropriately. (Jonson) Metrics measuring the health of code such are not necessarily as personal as they measure the quality of what they have produced. On the other hand metrics such as the amount of time a developer spent coding or the number of test suite runs is much more pervasive as it gives personal data about how they do their work.

It is clear that the ethical implications of this kind of monitoring are very worrying. They evoke an image of 1984. Where everything developers do is monitored and people do not have the privacy to do their work as they see fit. This kind of monitoring would create pressures on developers to work in a way that would make their metrics appear better than they are at the expense of the quality of their software engineering work. For example people may repeatedly build their code to boost their performance ratings.

If algorithms are making decisions about developers quality of software engineering work they will potentially have life changing effects on them. It could increase or decrease their likelihood of getting a promotion. The results of measurements of their software engineering

ability could even get them fired. It is for this reason that it is important we know the validity and accuracy of the results of any measurements or metrics.

This is especially true in the case of machine learning solutions to measuring software engineering. As these algorithms are trained rather than programmed we do not know exactly the process the machine learning model uses to decide on the quality of software engineering work. It is really a black box system. So there is a real potential for machine learning solutions to software engineering measurement to make decisions in ways that we do not expect or even want.
A possible example of this is a hypothetical model which measures software engineering performance of an engineer, which is trained using their commit history and engineering performance reviews from managers. A machine learning model trained in such a way could take into account any unconscious biases (regarding race, sex, religion or anything else) which the managers had in making those engineering performance reviews.

Other ethical concerns arise from attempts to integrate gamification as a platform for measuring into software development. I think gamification can be valuable but at the same time it can be a dangerous tool for measuring software development as it can lead to perverse incentives. Developers could end up prioritising reaching achievements and other game goals over actual development. This could for example lead to abusing metrics like committing every time a line or two of code is changed or writing tests for code in such a way to make it easier to earn an achievement rather than thoroughly testing the code.
In essence this problem comes down to making sure that the goals created by the gamification platform precisely match the goals of developing a more effective software development process and creating better pieces of software. Otherwise ill-thought-out gamification can cause problems that actually hurt the quality of software.

Another ethical issue with gamification which I had not considered before researching this area is the potential of patronising developers. I was particularly struck by this thought when I was reading about the efforts to emulate elements of an RPG in a software development gamification platform. Although some developers may like the idea, I imagine many would find having character rolls created for them patronising and may think their employer is treating them like a child.
I think developers may also feel that this kind of gamification is just a thinly-veiled disguise for more pervasive monitoring and measuring of their performance. I think most software engineers would feel more comfortable if companies were more honest about the monitoring of their development and did not try and hide it.

Software developers consider themselves professionals for the most part and should be given a certain amount of autonomy and trust to do their job correctly. I feel that too much detailed monitoring and metric collection about developers (especially when there are gamified goals for them based on this) will lead to developers feeling less empowered and proactive. If developers feel that their managers are always monitoring them then there may be a breakdown in trust.
Finally in all of these software engineering metrics and measuring methods it is imperative that the developers are fully aware of what is being measured about their work and what is

being done with the information. They must also give consent for this information to be collected to about them. This is not just an ethical issue but a legal issue, which is enforced by the data protection commissioner.

In summary there are many different methods, platforms and algorithms for measuring software engineering with varying evidence of their reliability. I have also outlined some of the ethical issues which arrive due to this measurement.

Bibliography:

1. *Fenton, N. E., and Martin, N. (1999) "Software metrics: successes, failures and new directions." Journal of Systems and Software 47.2 pp. 149-157*
2. *https://www.humanyze.com/*
3. *http://www.businessinsider.com/tracking-employees-with-productivity-sensors-2013-3?IR=T*
4. *https://www.washingtonpost.com/news/business/wp/2016/09/07/this-employee-badge-knows-not-only-where-you-are-but-whether-you-are-talking-to-your-co-workers/*
5. *The Active Badge Location System Roy Want1, Andy Hopper2, Veronica Falcão3 and Jonathan Gibbons4 Olivetti Research Ltd. (ORL) Cambridge, England http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.36.123&rep=rep1&type=pdf*
6. *http://www.hitachi.com/rev/pdf/2015/r2015_08_116.pdf*
7. *Searching under the Streetlight for Useful Software Analytics Philip M. Johnson, University of Hawaii at Manoa*
8. *Software Intelligence: The Future of Mining Software Engineering Data, Ahmed E. Hassan, Tao Xie https://www.nitrd.gov/nitrdgroups/images/f/f1/Software_Intelligence_The_Future_of_Mining_Software_Engineering_Data_p161.pdf*
9. *Does Test-Driven Development Really Improve Software Design Quality? David S. Janzen, California Polytechnic State University, San Luis Obispo, Hossein Saiedian, University of Kansas http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.304.1723&rep=rep1&type=pdf*
10. *On the Effectiveness of Unit Test Automation at Microsoft, Laurie Williams , Gunnar Kudrjavets , and Nachiappan Nagappan, https://collaboration.csc.ncsu.edu/laurie/Papers/Unit_testing_cameraReady.pdf*
11. *A Bayesian network approach to assess and predict software quality using activity-based quality models Stefan Wagner * https://klevas.mif.vu.lt/~sigitas/Kokybe/Straipsniai/1-s2.0-S0950584910001175-main.pdf*
12. *BENCHMARKING MACHINE LEARNING TECHNIQUES FOR SOFTWARE DEFECT DETECTION, Saiqa Aleem, , Luiz Fernando Capretz, and Faheem Ahmed https://arxiv.org/pdf/1506.07563.pdf*
13. *Turning Real-World Software Development into a Game, Erick B. Passos IFPI Lims Danilo B. Medeiros Infoway Sol. Inf. Pedro A. S. Neto UFPI Esteban W. G. Clua http://www.sbgames.org/sbgames2011/proceedings/sbgames/papers/comp/full/30-91552_2.pdf*
14. *It Was a Bit of a Race: Gamification of Version Control Leif Singer Et al.*
15. *Ethical issues in empirical studies of software engineering, J. Singer , N.G. Vinson, http://sci-hub.bz/10.1109/tse.2002.1158289*