Software engineering Measurement report

Student Name: David Hooban

Student ID: 17328486

Lecturer: Professor Stephen Barrett

Introduction

For as long as there have been hierarchical organisations there has been a desire to measure performance in quantitative terms that indicate just how effective someone is at their job. Some professions may be easier to measure than others. A telemarketer might be assessed based on the number of calls they make in a day, or the number of successful sales they make per unit time. But how do you measure the performance of an artist, or a musician? It is difficult to quantify much of what they do into simple, atomic metrics which can be objectively collected, analysed and compared to their peers, or indeed the industry as a whole.

This same problem is faced when assessing software engineers. Consider for example, comparing two software engineers. How would you determine which is better, if it were even possible? Any metric seems to collapse without further context. Even a combination of metrics only tells some of the story.

In this essay I examine the measurable data that is available be analysed about software engineers, the computational platforms that exist to do this very task, the algorithmic approaches that attempt to address this problem and the ethical dilemmas surrounding the collection and analysis of data derived from software engineers. I attempt to address the question of whether software engineers can be effectively measured and if so, whether we should pursue this area or not, from an ethical perspective.

Measurable data

In order to evaluate the performance of a software engineer, it is essential to determine objective metrics which can be used as an indication of their performance. It is important to establish the motivation for metrics, as we can examine endless metrics about a software engineer. One perspective is that, when measuring software systems, "quality metrics are beneficial for determining if the software system delivers the intended functionality"(Fenton and Pfleeger, 1998). Therefore, we may also be interested in similar metrics for software engineers.

Lines of Code

One of the earliest measures of software engineering performance is the Lines of Code (LOC) metric (Fenton and Neil, 1999). This is a simple measure of how many lines of code a software engineer has added to the overall repository of code for a given system and can be described as a numerical value. Many derivative metrics emerged from this such as Thousands of Lines of Code (KLOC), and defects per KLOC. It may also be useful to examine an individual's LOC count as a fraction of the overall LOC count in the repository to see roughly how much they have contributed to the project. LOC can be regarded as a static code metric, meaning it can be measured objectively and is found from parsing the source code, as opposed to process metrics, which aim to examine the quality of a system (Fenton and Pfleeger, 1998).

The issue that the LOC metric presents is that it implicitly assumes a position of quantity over quality. Those who contribute more to the code base are considered better software engineers if we just consider LOC as the sole metric of evaluation. This is problematic as it fails to take into account the quality of the code being introduced to a system. Consider two software engineers, one who writes a thousand lines of code but in doing so, creates twenty major bugs. Another software engineer may write twenty lines of code fixing these bugs, and ultimately making the application feasible to be released to clients. The former would be regarded as the clearly superior software engineer, though it is obvious that this is not the full story. As Coelho and Basu (2012) point out "A

skilled developer may develop the same functionality with less SLOC (Source Lines of Code) compared to a novice developer. As a program with less SLOC may exhibit more functionality than a program with a larger value of SLOC, SLOC is a poor productivity measure".

Defects per thousand lines of Code

Based on the previous discussion around LOC, it can be reasonably said that assuming code quality is equal across the board, LOC would be a more relevant measure. Therefore, perhaps an improved metric would be to consider the number of defects per some unit amount of lines of code. In effect, this may be an indication of code quality given that a better software engineer would introduce fewer bugs to a system. A defect or 'bug' can be defined as a piece of code which has the capacity to lead to unintended behaviour. It is important to note that the piece of code may not always lead to unintended behaviour, but perhaps in some edge cases, bugs may appear. This metric could be measured by recording the number of tests failed, or errors encountered while testing the code.

There are a number of issues with this measure. Firstly, it assumes perfect testing is being performed and that the number of bugs found is correct and exact. However, in reality this is often not the case, as any kind of testing software can be subject to the same flaws as any other piece of code. Secondly, it doesn't take into account the nature of the work being done. Some code may be 'cutting edge' so to speak, and may be attempting to accomplish something new, or perhaps the task at hand is quite complex. Other code might be simply a 'rehash' of similar code, and therefore is less likely to contain bugs. It seems hardly accurate to classify the former example as authored by a worse software engineer. Finally, it fails to actually account for code quality, it is just an indicator of thoroughness when it comes to writing code with less bugs, but not necessarily higher quality code. For example, two pieces of code may accomplish the same task but one may be clearly superior in terms of performance. This would not be accounted for in this metric. Therefore, there may be some utility to this measure, but it should be regarded as a primitive metric and would require further context to provide any meaningful insight.

Volume of Code Churn

Code churn can be defined as the "amount of code changes taking place within a software unit over time" (Knab, Pinzger and Bernstein, 2006). Code churn is an easily measured metric as the number of lines modified in a commit is recorded by most version control software systems such as Git. Code churn could be a good indication of code quality and therefore software engineering quality, as it effectively reduces the amount of measurement needed on the source code, but rather transfers the responsibility of determining quality to the engineers, and we simply measure the number of changes that engineers made to the existing code as an indication of how much code was deemed sub-optimal. If we take this further and measure code churn per engineer, we get a better indication of how much code had to be modified, and we could use this as an indication of a software engineer's performance.

This is not an unreasonable proposal, but some initial issues arise. Firstly, you cannot take code churn as an absolute numerical value if you plan to use it to compare software engineers, as larger code contributions that have a smaller fraction of changes may still have a higher absolute score. Therefore it is much more appropriate to consider 'relative code churn', such as the work done by Nagappan and Ball (2004). Nagappan and Ball (2004) show through experimental data that relative code churn metrics can be "excellent predictors of defect density in a large industrial software system". In other words, relative code churn is a good indicator of the number of bugs we should

expect and therefore can be used as an indication of the quality of a software engineer, as it would follow that a larger number of bugs consistently introduced may suggest a lower quality engineer.

Therefore, I would conclude that relative code churn is a reasonable indicator of the quality of code, but it is not without its own flaws. Further consideration needs to be given to the project management of the software team, as 'iterative development' based teams or 'prototype-driven' methodologies may see a larger amount of code churn compared to more traditional approaches such as the waterfall methodology. It also encounters similar issues as the measure of 'defects per number of lines of code' that I described previously, as it doesn't actually directly measure code quality but just gives an indication to how many bugs may be introduced.

Technical Debt

Technical debt can be defined as "what results when development teams take actions to expedite the delivery of a piece of functionality or a project which later needs to be refactored" (Productplan.com, 2019). Technical debt is mainly focused on measuring code that has been added to the repository out of necessity to push a feature, and comes at the cost of code quality. Therefore, it may be difficult to capture technical debt automatically, but it is certainly a manual metric that would be of high value to know about. By examining technical debt on an engineer by engineer basis, we get a sense of perhaps how much pressure an engineer feels under, or it could be an indication that an engineer is happy to sacrifice code quality to simply get the job done, which in general would not be a desirable characteristic.

A certain amount of technical debt is inevitable but by measuring an engineer's contribution to technical debt in the repository over time we can get a sense of how the engineer is progressing in terms of writing higher quality code.

Commits to the Repository

A measure that could be useful is the number of commits to the repository. This could be an indicator of the volume of work being completed by a software engineer. The metric that would be worth measuring is the volume of commits over time, as simply acknowledging that a commit just happened isn't of any great value, but seeing how the number of commits changes over time could provide a useful insight into the consistency of work ethic from a software engineer.

Some immediate issues surround this metric. For one, it encourages software engineers to commit often, even if a commit is not necessary. This can lead to a repository where it is difficult to identify meaningful commits in the history, and also it would be difficult to find the origin of a particular piece of code as there would be a large number of commits. It also may encourage developers to commit code which they know contains bugs or is low quality as long as they return to fix it later, which would count as two separate commits. This can lead to technical debt being introduced or hidden bugs in code that would've been spotted by a more thorough code review before committing.

Number of stories completed in a sprint

One of the issues with examining source code is that, as I have described above, it's difficult to assess code quality and therefore a software engineer's performance, and also it often seems to reduce down to subjective views on what code quality actually means. Therefore, we can look at measuring software engineers from another perspective: The number of tasks they've completed that they either were assigned to do, or assigned to themselves.

One example of this is teams that make use of 'user stories' which can be defined as "the smallest unit of work in an agile framework" (Atlassian, 2019). What is unique for user stories is it defines tasks from the perspective of how it would benefit the client. User stories are similar to a 'scrum board' or kanban system and therefore can be generalised to apply to those ideas too. In fact, most software teams will at some point break larger tasks down into small manageable tasks and so this metric can be reasonably applied. A storyboard contains all of the user stories and software engineers can pick up tasks from this storyboard. Typically, stories are rated based on expected difficult on some scale such as Fibonacci numbers. This is to account for the fact that more challenging stories will take longer. Therefore, a metric to consider is monitoring the number of stories completed during a 'sprint' (typically a two week period) or the number of points gained by an engineer.

One of the nice features of this approach is it makes it relatively easy to gauge the performance of a software engineer as a function of how much they contribute to the overall goal of the software team, without having to delve into the source code. It is also easy to compare engineers based on the number of points they accumulated during a 'sprint', which is usually just a two week period. Due to the fact that more difficult stories are assigned more points, we don't have to worry as much about an imbalance due to the stories difficulty causing engineers to spend longer on a given task. Also, a manager can monitor the software engineer's velocity which means the amount of work done during a sprint. Usually velocity is measured at the team level, and is an indication of how the team is performing, but by monitoring an engineer's performance from sprint to sprint, it can be a good indication of their performance, both compared to their peers and compared to themselves in the past. Finally, by allowing software engineers to assign other engineers who have assisted them, you can reasonably measure who is more effective on the team, as an engineer who constantly assigns others is indicating that they require assistance frequently, while another engineer who is assigned to many tickets indicates that they are capable of performing as what would be regarded as a more 'senior' role within the team.

However, when it comes to measuring a software engineer, this metric has some problems also. Once again there is the issue with code quality. A user story may be completed but this doesn't necessarily indicate that it was completed in the best possible manner. A concern that may arise from this is that some engineers may wish to provide lower quality solutions for user stories in order to gain more points, but at the cost of code quality. To remedy this, previous metrics like code churn can be used to attempt to assess the quality of the code. Most teams will have some form of code review process, and as a result, solutions deemed poor will be subject to code churn after the fact. This could be a useful measure of the quality of solution provided by an engineer. Finally, there is the issue with the points assigned to user stories. Given the fact that this scale is usually applied before the story has been picked up by an engineer, there is a certain amount of guesswork involved, and this will distort an engineer's velocity away from the true measure of how they are performing. A solution to this is allowing engineers to re-assign points to a story based on unforeseen circumstances, but that can also lead to engineers changing scores to boost their perceived performance.

Lead time

Lead time can be defined as "how long it takes for ideas to be developed and delivered as software" (Stackify, 2019). Therefore, it naturally would be applied to teams of software engineers. For example, the time between initially outlining a new application and delivering the application would be considered the lead time. However another interesting metric may be 'lead time per developer'. This would be defined as the time actively spent working on the development of the application per

developer. This could be further refined to remove or add in time spent testing the application or to account for time waiting for other engineers to complete their work, depending on what is being specifically measured. Lead time per developer could be an indicator of who are completing their work on time, and who is taking longer and can be a proxy indicator of productivity in software engineering.

Communication output or volume

Communication is a major component of working on a team. It is essentially what enables teams to be more effective than the sum of the individual efforts of the engineers if they worked separately. The ability to organise, delegate and reconcile issues can make a major difference in the performance of a team. For most of history, measuring communication between team members would have been a manual task that managers would've been charged with and would be difficult to capture accurately. However, as technology has evolved, more and more communication takes place online and therefore can be processed. Also, with dispersed remote teams becoming more common, digital communication is rapidly becoming a key component of software teams all over the world.

Certainly one important component of communication is whether communication is occurring at all. By measuring the engagement levels of an engineer with group chats, or the number of active private chats at any time, we can get a sense of how much an engineer is communicating with the team and can be an indicator of their role within the group. This could be measured using the communications channels such as Slack or Microsoft Teams.

The general sentiment of an engineer is also something that could be measured through the use of natural language processing and sentiment analysis. This would be subject to some constraints based on the accuracy of the natural language processing algorithms but might provide some insight into the general mood of the group and specific engineers.

The algorithmic approaches available

Natural language processing / sentiment analysis

Natural language processing can be used on communication data to attempt to perform sentiment analysis on the communications of an engineer. Sentiment analysis refers to "the automated process that uses AI to identify positive, negative and neutral opinions from text" (MonkeyLearn, 2019). This could be useful in determining the attitude of an engineer and perhaps there may be a correlation between sentiment in communications and job performance. Tools like the Stanford NLP sentiment analyser can be applied to natural language and outputs an estimate of the sentiment of the statement(s).

Unfortunately, work by Jongeling, Datta and Serebrenik (2015) based on experimental data shows that current sentiment analysis tools are not suitable for accurate sentiment analysis in the software engineering field. As natural language processing improves, which there is no doubt that it will over time, we may see sentiment analysis become more relevant in measuring a software engineers general sentiment as an indication of how they may be performing.

On a theoretical level, it would be interesting to see how sentiment correlates to factors like code churn, commit count and user story velocity, and whether there is a significant link between general mode and the performance of a software engineer. Further to this, it may be the case that sentiment between employees could correlate to a software engineer's performance. If the engineer is

generally unhappy with others, this could be linked to poorer performance. In my opinion, the value of sentiment analysis would be more as an 'early warning' indicator to management that the mood of an individual or team has shifted.

Personal Software Process

The Personal Software Process (PSP) is a method of evaluating a software engineer to help them improve over time. It is comprised of different levels that the engineer should work to incorporate into their work over time to become a better software engineer. PSP is based on the idea that: "To consistently improve their performance, engineers must personally use well-defined and measured processes" (Humphrey, 2000).

The main process involved in PSP is to gradually begin to record and analyze information and use this to improve over time. For example, the first stage (PSP 0) focuses on how to "record development time and how to log each compile-and-test defect" (Ferguson et al., 1997). The engineer proceeds to record issues they encounter to improve processes in PSP 0.1. Design and code reviews are brought in at PSP 2, and then at PSP 2.1 specification techniques are introduced.

In essence, PSP 0 focuses on process discipline and measurement, PSP 1 focuses on estimating and planning and PSP 2 focuses on quality management and design (En.wikipedia.org, 2019). The key focus of PSP seems to be the accountability factor as a method of improvement. In other words, when an engineer records their work in quantifiable measures, there is an implicit need to justify that output and an incentive to improve over time now that they have a benchmark to help guide them. It also helps to identify an engineer's shortcomings that they may not even be conscious of.

There have been criticisms of PSP such as how "The amount of PSP paperwork and manual calculation has long concerned us" (Johnson and Disney, 1998). However as the field of big data emerges with larger computational power, and more analytics to be recorded, applications like 'Process dashboard' or 'Hackystat', attempt to automate the collection and analytics of PSP can be significantly beneficial to organisations.

By automating the process of information recording, PSP can provide insights into things like how an engineer is performing over time, how an engineer performs on certain teams or when working with certain engineers, how an engineer performs on certain tasks that may be new to them, versus tasks they are familiar with, which can provide an indication of who is more suitable to be moved to new teams if needed. If correctly implemented, PSP can help managers make an informed decision about who is performing best, who can adapt quickest, and who works better when working together and more. However, by automating this process, some of the PSP benefits may be lost. As outlined by Nasir and Yusof (2005), By moving the point of analysis from the individual engineer critiquing themselves to the automated system which may be monitored by management, we may see a shift in behaviour away from self-improving activities to behaviour intended to 'game the system', or the concept of 'measurement dysfunction', which means the act of recording information changes the way people behave as they know they're being monitored.

Computational Intelligence

Computational Intelligence (CI) is a bit of an umbrella term to refer to "fuzzy sets, neural networks, and evolutionary computing" (Pedrycz, 2002). CI therefore deals naturally in the area of uncertainty and fuzzy data, versus concrete algorithmic processes that can be mapped out ahead of time and data which is guaranteed to conform to an exact standard. Neural networks would be best suited

for the discussion of an algorithmic approach to measuring software engineers and as such will be the focus of this discussion.

A Neural Network (NN) is a form of machine learning algorithm that is trained on data, to make decisions based on fuzzy sets, by outputting probabilities that correspond to the NN's decision. This is the power of using NNs: It is designed to handle non-conforming data that may not have been seen by the system before, but use what it has seen and has been trained on, to attempt to produce reasonable output that would be in line with the data it's seen before.

The key component of NNs is a large source of good data to train the NN on, and this can be difficult to find. Helie, Wright and Ziegler (2018) use LGTM ('looks good to me'), which is a website, designed to evaluate code quality from commits. LGTM analyses "over 10M commits submitted by ≈ 300K developers to ≈ 56K open source projects". This provides a rich source of 'code quality' for which they can train a neural network on. The aim of this is to create an algorithmic process which would determine code quality.

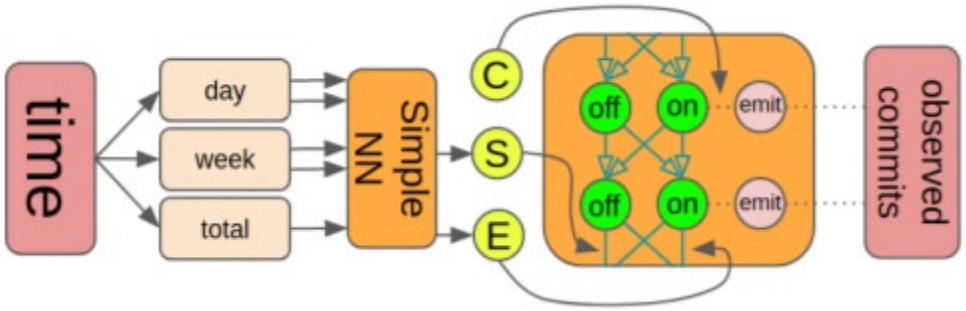

Figure 1: Work by Helie, Wright and Ziegler (2018) to attempt to construct a Hidden Markov Model with a neural network to measure software development productivity as a function of code quantity and quality

One challenge associated with NNs can be determining what a successful outcome should look like. If you train the NN to associate quality with metrics like LOC, it will output that engineers who write more code are performing better. Therefore, when training a NN we must invest a significant amount of time into determining what metrics to train the NN on, where we can find high quality data (or if we would have to obtain that data ourselves), and the potential unforeseen effects associated with ranking engineers based on this NN.

Supervised learning would be, in my opinion, the preferred choice for training neural networks, as through the use of certain metrics, we can define what a 'good' software engineer may look like. Supervised learning means we provide both the input data and also the expected output, and the NN learns to map that input to the given output. Once the NN has seen enough examples, it can reliably link a certain type of input to the correct expected output.

This can lead to the topic of data bias which is not a trivial issue to overcome. Due to the fact that someone is determining the 'correct' output based on given input data, any form of bias that may appear in these data sets will affect the NN and how it perceives data. For example, if there was a situation in which two factors have an equal but opposite effect on the outcome but a decision must be made about which is more important, the author who composed the dataset may implicitly include their bias towards one factor in the 'output' or correct result. Since the neural network is trained on this information, it too will possess this bias.

An overview of the computational platforms available

Hackystat

Hackystat is an open source framework for "collection, analysis, visualization, interpretation, annotation, and dissemination of software development process and product data" (Csdl.ics.hawaii.edu, 2019). Effectively it automates the process of collection of data for PSP. Hackystat makes use of something they call 'software sensors' which are conceptually 'attached' to applications and can monitor usage on the application, and feed that raw information back to Hackystat. From there, Hackystat can analyse the information. One benefit is it will reduce recording errors, as it is done automatically and so the data will be more reliable. Another benefit of this approach is that it removes the burden of manually recording information, which is what is expected for users of regular PSP. This is one of the major criticisms of PSP, and is outlined in a report on the topic with the following quote:

"Although the training was generally well received, use of the PSP in TIS started to decline as soon as the classes were completed. Soon, none of the engineers who had been instructed in PSP techniques was using them on the job. When asked why, the reason was almost unanimous: "PSP is extremely rigorous, and if no one is asking for my data, it's easier to do it the old way"

- Webb and Humphrey (1999)

The fact that there are computational platforms which exist to fully automate PSP make it feasible, as otherwise it appears that it would simply be too large a task, despite the potential benefits that could be realised from doing it. Even if the paperwork of manually filling in the PSP forms wasn't overly burdensome, it does take away a software engineer's attention from the task at hand to do something that is intended to make them better, which is counterproductive. By having an automated platform running in the background, engineers are free to continue their work without interruption.

GitPrime

GitPrime is a platform which performs analytics on Github repositories. It targets metrics like the codebase contents, commits velocity, activity and more on the platform and presents this information in a graphical or report format for team leaders to analyse and interpret.

A sample of an analytics visualisation by GitPrime
Source: https://blog.gitprime.com/gitprime-insights-email-reporting-feature/

The visual representation of data in this way makes it easy for team leads or managers to interpret this information and extract important information from it quickly, and saves a lot of time compared to manually reviewing the team's repositories on a regular basis. A key feature of any analytics platform is its ability to visualise information in such a way as to take the work out of reading it, otherwise the person reading it could simply look at the underlying data such as the repository. It must be effective in conveying as much meaningful information as possible in a relatively short period of time. It also allows team leads to also examine the individual impact of an engineer, and see how they are performing within the team, and whether they are improving over time.

One factor to consider is the price. GitPrime is a premium service that costs money, and the cheapest option is $749 per month as listed on their website. Therefore smaller companies or start-ups may not be able to afford this service.

Humanyze

Humanyze is an analytics platform solely based on measuring employees in the workplace. The platform looks at a range of metrics such as communication within a team and between teams, the layout of offices and the impact this has on performance, employee workloads, employee turnover and more.

One of Humanyze's products is a 'sociometric badge' which employees wear which aims to monitor communication and face-to-face interactions. The aim of doing this is to help diagnose organisational issues which may be caused by a lack of communication.

Fitbit

Fitbit is a fitness platform which offer wristband devices that record a range of health and fitness metrics like steps taken, how a person sleeps, a person's heart rate, how a person is breathing and more (Fitbit Health Solutions, 2019). Fitbit now offer solutions for employers to provide their employees with Fitbit wristbands to help monitor health and fitness information on the individual level and encourage healthier behaviour from employees. Fitbit also offer health wellness services such as personal coaching and advice to an individual to help improve their health over time.

Fitbit metrics can provide an insight into how an employee's behaviour in the real world affects their job performance. For example, we may observe that when an employee takes a break and goes for a walk (which would be indicated by a large rise in steps for a block of time), their job performance after completing this exercise increases. This may help inform employers about the type of work environment employees need or the structure and length of their workday and they can even tailor this to an individual level.

SeaLights

SeaLights is an analytics platform to help monitor repositories and looks at addressing issues like testing optimisation, performance analytics, technical debt management, test coverage, sprint planning, co-ordination between teams and more. SeaLights differentiates itself as a platform by taking a testing oriented perspective on measuring software engineers.

SeaLights monitors code changes and how the product is actually used in production to help inform the testing process. It does this to identify what they call "Testing gaps", which are features or areas of the code that are used in production but are not adequately tested. This is a very appealing feature as for most software teams, it can be difficult to know if test coverage is properly applied to the main features being used, and may also inform teams about how the application is being used

and therefore can be indirectly used to gain an insight into the user experience. Testing gaps also help prioritize tasks as clearly the features being used by users which are not adequately tested will need to be looked at as quickly as possible. As SeaLights point out on their website, this can avoid over-testing and instead focus on the areas that are in most need of attention.

SeaLights also examines areas such as technical debt. SeaLights view low-risk technical debt as a good thing as you simply cannot make every line of code perfect and therefore in order to make reasonable progress you have to accept a certain amount of technical debt. They try to account for this, and identify and separate higher-risk technical debt through analytics on their platform.

The ethics of analytics

The ethics of measuring software engineers is a crucial topic to discuss because of the fact that there is no end to the list of data points a large corporation could gather on an employee in pursuit of learning what makes employees perform the way they do. A corporation could measure an employee's health information to try and figure out how long they will be with the company. This isn't science fiction, it is already happening, as shown in an article by The Washington Post (2019) (Article link found in reference list). The reality of the future seems stark if it continues and our privacy is effectively stripped from us in the pursuit of improved analytics.

There is also the issue of automating analytics such as PSP as described above. PSP was originally a manual process of filling in forms. As we move towards automated collection and analytics of information, whether that is PSP or some other platform, we lose the ability to choose what data we input and become subjects to a process we would've once controlled. We may also lose our right to justify certain statistics recorded by these automated processes. The point is not to focus on PSP and whether it is automated or not, but that, the ability to automate data collection processes that will be ultimately used against us is improving every day and in many ways poses the threat of limiting our contribution to how we are viewed by our employers, is deeply concerning.

Platforms like Humanyze are in many ways even more concerning. These platforms seem to believe that all data is fair game and free to be recorded and analysed by employers. Data points like whom in the office we speak to and who we don't, and how this affects how we perform, is something that Humanyze specialise in. These types of metrics can quickly move into monitoring our conversations and trying to analyse our behavioural patterns, all in the pursuit of better performing employees. The reason I believe this is unethical is because we are humans first, and employees second. When dealing with people, we should not treat others as a scientific experiment to get a potential boost in performance.

Devices like Fitbit pose an interesting ethical dilemma also, as it focuses on measuring health and wellness information. Fitbit's website cites that improving employee health is in the direct benefit of the employer due to the fact that "the associated healthcare costs—especially for those employees living with or at risk for chronic disease—are unsustainable." (Fitbit Health Solutions, 2019). However what if devices like Fitbit could use employee data to detect employees who were likely to be affected by some form of chronic illness in the future. This type of prediction could be made based on movement information, sleep information, heart rate information or more. Given the large data base Fitbit would have access to it wouldn't be hard to make a reasonable prediction about who may suffer from what issues in the future. Employers who are eager to cut costs can view this prediction and begin to terminate employees based on this information, or worse, force them to quit due to unfair working conditions, to avoid paying any form of healthcare costs in the future.

A major issue of data collection is the fact that a small group of people will have access to this information. Naturally, most data collected on engineers would be regarded as confidential and therefore would not be made available to everyone, but there is a small group of people within an organisation who will have access to this information and are free to make decisions based on this information. The average employee has no idea how this information is being used, or who is viewing it and this creates a 'big brother' illusion. Employees might feel like they are constantly actively monitored even if they aren't. Not only will this lead to issues like 'measurement dysfunction' as I discussed previously, but it is reasonable to say that this is unethical because of the psychological effect it will have on employees.

Government legislation such as GDPR which is imposed by the EU shows that governments need to police the storage of a person's information on servers around the world. The infamous "right to be forgotten" highlights this fact; without legislation, the right to be forgotten is easily taken from us without our knowledge.

In my opinion, privacy is a human right and as such there should be a reasonable entitlement to protect information which we deem private from the organisations we work for. This might include health information, sexual preference or gender identity and so much more. The reason I believe this is simple: We are employed based on our ability to do our job, and should only expect to be assessed on this basis. Insofar as the data organisations collect about us is based around our performance as a software engineer within the bounds of the office or working environment, I believe it is entirely ethical to do so. When organisations assess us based on factors unrelated to our work performance, such as health, I believe it becomes unethical.

Reference List

Atlassian. (2019). 'User Stories | Examples and Template | Atlassian.' [online] Available at: https://www.atlassian.com/agile/project-management/user-stories [Accessed 3 Nov. 2019].

Coelho, E. and Basu, A. (2012). Effort Estimation in Agile Software Development using Story Points. International Journal of Applied Information Systems, 3(7), pp.7-10.

Csdl.ics.hawaii.edu. (2019). Hackystat | Collaborative Software Development Laboratory. [online] Available at: http://csdl.ics.hawaii.edu/research/hackystat/ [Accessed 4 Nov. 2019].

En.wikipedia.org. (2019). Personal software process. [online] Available at: https://en.wikipedia.org/wiki/Personal_software_process [Accessed 3 Nov. 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.

Fenton, N. and Pfleeger, S. (1998). Software Metrics: A rigorous and practical approach. 2nd ed. Boston, MA: Course Technology.

Ferguson, P., Humphrey, W., Khajenoori, S., Macke, S. and Matvya, A. (1997). Results of applying the Personal Software Process. Computer, 30(5), pp.24-31.

Fitbit Health Solutions. (2019). So Much More Than Steps | Fitbit Health Solutions Features. [online] Available at: https://healthsolutions.fitbit.com/somuchmorethansteps/ [Accessed 5 Nov. 2019].

Fitbit Health Solutions. (2019). Support Healthy Behavior Changes in Your Employees | Fitbit Health Solutions. [online] Available at: https://healthsolutions.fitbit.com/employers/ [Accessed 5 Nov. 2019].

Helie, J., Wright, I. and Ziegler, A. (2018). Measuring software development productivity: a machine learning approach. Presented at the 'Machine Learning for Programming' Workshop affiliated to the 30th International Conference on Computer Aided Verification.

Humphrey, W. (2000). The Personal Software Process.

Johnson, P. and Disney, A. (1998). The personal software process: a cautionary case study. IEEE Software, 15(6), pp.85-88.

Jongeling, R., Datta, S. and Serebrenik, A. (2015). Choosing Your Weapons: On Sentiment Analysis Tools for Software Engineering Research. IEEE, pp.531-535.

MonkeyLearn. (2019). Sentiment Analysis. [online] Available at: https://monkeylearn.com/sentiment-analysis/ [Accessed 4 Nov. 2019].

Knab, P., Pinzger, M. and Bernstein, A. (2006). Predicting Defect Densities in Source Code Files with Decision Tree Learners. [online] pp.119-125. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.452.8933&rep=rep1&type=pdf [Accessed 31 Oct. 2019].

Nagappan, N. and Ball, T. (2004). Use of Relative Code Churn Measures to Predict System Defect Density. Proceedings of the 27th international conference on Software engineering, [online] pp.284-292. Available at: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.85.7712&rep=rep1&type=pdf [Accessed 1 Nov. 2019].

Nasir, M. and Yusof, A. (2005). AUTOMATING A MODIFIED PERSONAL SOFTWARE PROCESS. Malaysian Journal of Computer Science, 18(2), pp.11-27.

Pedrycz, W. (2002). Computational Intelligence as an Emerging Paradigm of Software Engineering.

Productplan.com. (2019). What Is Technical Debt? Definition and Examples. [online] Available at: https://www.productplan.com/glossary/technical-debt/ [Accessed 5 Nov. 2019].

Stackify. (2019). What are Software Metrics? Examples & Best Practices. [online] Available at: https://stackify.com/track-software-metrics/ [Accessed 5 Nov. 2019].

The Washington Post. (2019). With fitness trackers in the workplace, bosses can monitor your every step — and possibly more. [online] Available at: https://www.washingtonpost.com/business/economy/with-fitness-trackers-in-the-workplace-

bosses-can-monitor-your-every-step--and-possibly-more/2019/02/15/75ee0848-2a45-11e9-b011-d8500644dc98_story.html [Accessed 4 Nov. 2019].

Webb, D. and Humphrey, W. (1999). Using the TSP on the TaskView Project. Crosstalk.