MEASURING SOFTWARE ENGINEERING

BRIEF

To deliver a report that considers the ways in which the software engineering process can be measured and assessed in terms of measurable data, an overview of the computational platforms available to perform this work, the algorithmic approaches available, and the ethical concerns surrounding this kind of analytics.

INTRODUCTION

Sommerville's Software Engineering defines Software Engineering as: "an engineering discipline that is concerned with all aspects of software production." How do we approach the task of measuring this discipline? How can we judge if a software engineer is performing his/her job well? How to we rank the product of the software engineering process? How do we assess this process itself?

These are all questions that this report aims to answer under the headings of Measurable Data, Computational Platforms, Algorithms and Ethics.

It must be noted, first and foremost, that the software engineering process can vary immensely, depending on the project being undertaken. When attempting to judge the software engineering methods and techniques involved in a software process, we need to give consideration to the type of application that is being developed. Therefore, we will never be able to definitively define the "best" software engineering technique, the characteristics of the "best" code, or the traits of the "best" software engineer. The projects we will consider require software engineering, but we cannot say that they all require the same software engineering techniques.

It is against this background that this report is written. We must find general ways of measuring this process that we can apply to all software engineering methods and techniques.

The key challenges facing software engineering today are coping with increasing diversity, demands for reduced delivery times, and developing trustworthy software (Sommerville, 2011). It is hoped that accurate measurement and analysis of software engineering can further the field in the pursuit of these aims.

The report also aims to give colour to the computational platforms available to perform this analysis and the algorithmic approaches available to monitor this data. With this level of intense analysis, it is also important to consider the ethical impact of this work. This is detailed towards the end of the report.

Measuring Data

Software Metrics

Software metrics are numbers, tools and analysis that are used to measure performance in software engineering (Fenton & Neil, 1999). It is an umbrella term for numbers that characterise properties of software code, to models that help predict software resource requirements and quality. Software metrics are needed to improve the way we monitor, control and predict various attributes of the software engineering process.

Fenton and Neil's article "Software metrics: successes, failures and new directions" goes into detail on the history of software metrics and where we stand now with their use and implementation in industry.

The use of active software metrics dates back to the late 1960s (Fenton & Neil, 1999). At this point, the main software metrics used were LOC (Lines of Code) and KLOC. These metrics were used to represent the size of the software engineering project. However, this basic metric overlooks other important aspects of code such as functionality and complexity.

Since then the amount of academic work on the subject has blossomed. There has been a vast array of books, journal articles, academic research projects and dedicated conferences on the subject. However, the increased academic scrutiny does not mean that the measurement of software engineering is happening in industry. In fact, R.L. Glass stated in an article for the Journal of Systems Software in 1994 that: "What theory is doing with respect to measurement of software work and what practice is doing are on two different planes, planes that are shifting in different directions." (Glass, 1994)

Almost all industrial metrics programs incorporate some attempt to collect data on software defects discovered during development and testing (Fenton & Neil, 1999). McCabe's cyclomatic number and function points are two metrics that have also been proposed in academia but have proved complex and not very practical.

Therefore, a necessary criterion to judge the success of software metrics is the extent to which they are bring routinely used across a wide section of the software production industry (Fenton & Neil, 1999). Why are the metrics not applicable to industry? Firstly, the academic studies into software metrics have only really been applicable to short programs. Secondly, the metrics are focused on detailed code metrics, that do not have much use in real process (ibid.).

Steven A. Lowe discusses the relevancy of metrics in software engineering in his article "Why metrics don't matter in software development". His view strongly suggests that measurements should be designed to answer business questions. This is in stark contrast to the academic and scientific emphasis placed on the metrics as detailed above. He suggests that we should measure the success, and hence the quality of a software engineering project by the customer happiness it creates and the business value it delivers. This represents a shift from traditional, academic and scientific metric use, to more subjective and difficult to measure ones. In his view, the first step to recording useful metrics is to define the 'value hypothesis' of your project – what does it set out to achieve for the customer. After this, we can go about selecting the correct metrics against which to test our hypothesis.

It is also worth noting at this point, that there is a notable difference between metric creation in software development and manufacturing. In software development, everything that is created is unique and incomparable – what Lowe describes as 'a snowflake'. The metrics that we choose will only hold relevance to the individual, team or project for which they were created. Metrics are not comparable between individuals, projects or teams.

Defining quality

Steven A. Lowe discusses nine metrics that are still applicable for use in industry today in his article "9 metrics that can make a difference to today's software development teams".

He details them in the article as outlined below:

1. **Leadtime** – how long it takes you to go from an idea to completed software.
2. **Cycle time** — how long it takes you to make a change to your software system and deliver that change into production.
3. **Team velocity**—how many units of software the team typically completes in an iteration**.**
4. **Open/close rates**—how many production issues are reported and closed within a specific time period.
5. **Mean time between failures (MTBF)**
6. **Mean time to recover/repair (MTTR)**
7. **Application crash rate**—how many times an application fails divided by how many times it was used. This metric is related to MTBF and MTTR. Both are overall measures of your software system's performance in its current production environment.
8. **Endpoint incidents**—how many endpoints have experienced a virus infection over a given period of time?
9. **MTTR (mean time to repair)** - the time between discovery of a security breach and a deployed, working solution.

Limitations of Metrics for Predicting Software Quality

It is easy for a manager to look at an alarming metric and jump to conclusions. However, it is very important to stress that you cannot use metrics to assume causes. You must go and talk to the team responsible to get the full story, to see if they agree that the metric result is a cause for concern.

It should also be noted that care should be exercised when interpreting software metrics. It is easy to look at an absolute number and jump to conclusions but we must remember to consider the result in context. This sentiment is best summed up with Bill Gate's iconic quote: "Measuring programming progress by lines of code is like measuring aircraft building progress by weight."

Another example of this when we look at the absolute number of defects as opposed to the defect density. The defect density is the number of defects per new or altered line of code in a program. Of course, we expect longer code to have a larger number of defects. Our primary interest lies in determining the number of post-delivery faults in our software. However, hypothesis tests have been carried out on the metrics described in academia and it indicated that complexity metrics are very poor predictors on fault density post-release (Fenton & Neil, 1999).

Computational Platforms Available

The second part of this report deals with the gathering and interpretation of data with which to analyse the performance of a software engineering project. Now that we know the data that we want to collect and use, we need to go about actually collecting and analysing it.

There are a number of computational and software packages that allow us to do this.

Collection of Data

First we need to assess the data that we have at our disposal. The paper, "Searching under the Streetlight for Useful Software Analytics", by Philip M. Johnson discusses some of the ways of gathering this data from software engineers. The paper puts an emphasis on trying to counteract the effects of observational bias when collecting and analysing data from software engineers. Observational bias is the phenomenon whereby those collecting data may only collect the data that is easily and readily available.

In 1995, Watts Humphrey gave details of the Personal Software Process in his report: "A Discipline for Software Engineering". PCP collects information from the software developer, like a project plan summary, a defect-recording log, a design checklist, and a code checklist. The idea being that these metrics can help us to measure the software engineer's performance. PSP also enables software engineers to monitor and control their personal workflows on a daily basis. Key metrics monitored by PSP include: productivity, reuse percentage, defect information and earned value metrics (Humphrey, 1995). PSP can help to improve project estimation and quality assurance.

However, there are flaws with the PCP. Within the paper, Humphreys says that "It would be nice to have a tool to automatically gather the PSP data. Because judgment is involved in most personal process data, however, no such tool exists or is likely in the near future."

The Leap toolkit advances on the PSP process through automation. It is lightweight, portable and provides reasonably in-depth conclusions. However, Leap is far more inflexible than PSP as the data analyst would have to design and implement a whole new Leap toolkit for each new task.

The paper “Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined” by Johnston et. al., gives details of the next innovation in software data collection. Hackystat requires the development of client-side sensors that attach to development tools and automatically collects metrics such as: effort, size, defect, etc.

Metrics are collected by sensors and sent to the server at regular intervals. The “Daily Diary” shows the developer metrics at 5 minute intervals. The “Daily Diary” also serves as a basis for generating other analyses, such as: the amount of developer effort spent on a given module, the change in size of a module per, the distribution of unit tests across a module, their invocation rate, and their success rate per, the average number of new classes, methods, or lines of code written in a given module per day, and so forth (Johnson, 2003).

The main advantage of Hackystat over PCP is that it is automated and collects the information automatically at regular intervals. However, it analyses much of the same information as PSP, such as time spent on the project, size of the project, commits, as well as some more advanced data, such as code coverage and complexity.

Visualisation of Data

After collecting all of this data from the software engineer, we then have to look at how to measure, interpret and report our findings. With this information, we can use data analytics to improve the engineering process.

There are a number of platforms available to track the metrics obtained by the above processes.

GitHub

GitHub is a web-based version-control and collaboration platform for software developers. It is an open-source platform where managers can analyse developer data and measure software engineering progress for free (Github, 2018).

GitHub then also provides tools to allow us to visualise this data. For example, one visualisation that every member has on GitHub is the contribution heat map. The heat map is a coloured graph that gives details of the number of commits over a certain period of time. Managers can use this information to compare the contributions of individuals on a given project.

The REST API on GitHub can be downloaded to allow metrics such as: the number of commits made to the project, the number of developers contributing to the project and the frequency of in individual contributor's input (Github, 2018). This data obtained from the REST API can then be accessed using other programming languages such as R and Python to create visualisations of the data and create meaning for managers.

Another way of visualising the data collected from GitHub is the online software platform "Screenful Metrics". It is an add-on to GitHub and allows you to create visual dashboards and automated reports based on the metrics obtained from the software developers that you can then share with the development team and management (Screenful Metrics, 2018).

Code Climate

Code Climate is a software product that offers static program analysis tools for developers. Its focus in on helping software developers to improve the quality of their source code (Carpenter, 2018).

William Carpenter wrote an article for Investopedia on what Code Climate is and how it works. Code Climate performs automated analysis of software code to identify and flag potential code errors, security vulnerabilities and instances of flawed coding methodology. It can collect and present quantitative and qualitative metrics describing and summarizing various aspects of the code. The platform can be hosted on-site in a company data centre or accessed as a cloud-based service. Code Climate has two fee-based service plans and a number of free service options. In June 2015, Code Climate also began providing free access to its static program analysis tools through a locally installable command-line interface (CLI). This option disconnects the analytical tools from the integrated platform, allowing developers to run manual analyses on local computers. The tools themselves were also made available to the public through a free open-source license. Code Climate hopes to spread the use of its base technology to build a bigger market for the services it provides through its paid platform. Code Climate can also be used in conjunction with GitHub, where it will show a user things like their code coverage, technical debt and a progress report (Carpenter, 2018).

Algorithmic Approaches

After we have collected all of the data, we can now begin to look beyond the simple techniques described above and begin to look deeper into the data.

Bayesian Belief Nets

The association between size and fault density is not a causal one. Therefore, we are challenged to produce models that will take account some of the concepts missing from traditional statistical approaches. The most significant of these being uncertainty and incomplete or subjective information. Fenton and Neil in their paper "Software metrics: successes, failures and new directions" discussed a potential solution to the problem: Bayesian Belief Nets (BBNs). It could be said that these BBNs paved the way for more modern algorithmic techniques used in software engineering today.

The development of algorithms and software tools that implement them made it possible to incorporate Bayesian probability into models. BBNs are seen as a tool to support decision making under uncertainty. At a high level, a BBN is a graphical network with an associated set of probability tables. A probability is associated with each variable in the BBN and these probabilities are arrived at through empirical analysis and subjective estimates. A BBN will compute the probability of every variable regardless of the amount of evidence for that variable. The BBN collects data like the defect counts at different stages of testing, the size on complexity metrics at each development phase and approximate development and testing effort. The BBN will then make forecasts of effects of process changes, perform 'what if' analysis and ultimately aims to predict defects post-release. This is one of the major metrics in software engineering that was identified earlier.

BBNs have been used by Microsoft in the Microsoft Office wizard and in decision making on the Space Shuttle (Fenton & Neil, 1999). It is interesting to see how this algorithmic approach can be applied in measuring software engineering and how it has progressed to the complex algorithmic techniques that we see today.

Process Quality Index

Following on from this, another interesting algorithmic approach to looking at a software engineer's data is the Process Quality Index (PQI). It is discussed in "The Personal Software Process," (Pomeroy-Huff, 2009).

The PQI has five components:

1. Design quality is expressed as the ratio of design time to coding time.
2. Design review quality is the ratio of design review time to design time.
3. Code review quality is the ratio of code review time to coding time.
4. Code quality is the ratio of compile defects to a size measure.
5. Program quality is the ratio of unit test defects to a size measure.

The five PQI element measures are then multiplied together to give a number between 0.0 and 1. Values below 0.5 indicate that the product is likely to be of poor quality. The lower the value, the poorer the quality is likely to be.

This index is a simple algorithmic metric to measure the software engineering and it makes use of data collected from the platforms mentioned above.

Multivariate Data Analysis

The data collected from the methods outlined above is quite complicated and there are many variables associated with each data point. To this end, the data can be described as multivariate and we can look at some the multivariate analysis techniques to gain a better insight into the data (Houlding, 2018).

When dealing with algorithmic analysis, we need to be aware of the two different families of techniques that we can use, supervised methods, and unsupervised methods.

Supervised methods, are so called because they 'teach' the mapping function though trial and error. The mapping function is an algorithm that maps an input to an output. After the 'teaching' the algorithm sufficiently, the function should be able to predict the output for new inputs. Some supervised methods that we can use to measure the software engineering process are linear discriminant analysis, k-nearest neighbours and logistic regression. They fall into two main categories, classification and regression.

Unsupervised methods, on the other hand, do not have a teaching process. We must discover the structure of the data, using only the data that we are given. Unsupervised learning methods include clustering where the input data is placed into distinct groups based on similarities and differences.

Computational Intelligence

Computation Intelligence (CI) is a tool used by the software companies in the previous section to gain a better insight into the data provided by the software engineers (IEEE, 2018). It refers to the ability of a computer to learn a specific task from data or experimental observation (ibid.). Computational Intelligence is particularly useful in scenarios where mathematical approaches are not adequate to solve a complex problem. The three main areas of CI are Neural Networks, Fuzzy Systems and Evolutionary Computation.

Artificial neural networks (NNs) are not algorithms themselves, but they provide a framework for machine learning algorithms to work together and process data inputs (IEEE, 2018). NNs are interconnected groups of nodes that use examples to generate identifying characteristics from the learning material that they process (ibid.).

Fuzzy Systems can be used to model uncertain problems such as linguistic imprecision by altering the use of traditional logic to include things that may be 'partially true' (IEEE, 2018). We can this use this output to perform approximate reasoning.

Evolutionary computation (EC) is used to solve optimization problems by generating, evaluating and modifying a population of possible solutions (IEEE, 2018). EC includes genetic algorithms, evolutionary programming, evolution strategies, genetic programming, swarm intelligence, differential evolution, evolvable hardware and multi-objective optimization (ibid.).

ETHICAL CONCERNS

While there are undoubtedly many benefits of the machine learning and data analytics services mentioned above, there are also risks and responsibilities involved with the practice.

All those involved with the software engineering profession must aim to be proactive rather than reactive to the ethical questions raised by some of the new techniques being introduced to the field. For example, one of the prominent questions raised today regarding ethics in software engineering, is the monitoring of the use of AI and machine learning. Orlando Torres wrote a piece entitled "7 Short-Term AI ethics questions" discussing the matter.

BIASES

Machine learning algorithms learn from the training data they are given, regardless of any biases that are present in the data. This is a great feature for some things, but it becomes an ethical issue when we consider that the data may have sexist or racist biases in it. The resulting prediction will also reflect this bias. Some examples include the scandal when the new Xbox software identified some people as gorillas, or biases in credit-worthiness and hiring algorithms (Torres, 2018). We need to make sure that these algorithms are fair, but this poses a particular issue when we consider that the algorithms are privately owned by corporations and not regulated in any way.

Related to this ethical issue is the problem posed by deep learning algorithms. Deep learning algorithms can arrive at results, without being able to explain why it arrived at that conclusion. For example, some algorithms haven been used to fire teachers, without being able to give them an explanation of why the model indicated they should be fired (Torres, 2018). From an ethical standpoint, transparency of algorithms is critical.

In addition, if software engineers are designing algorithms to make decisions – who will have the final say on these decisions? Are the biases present in human judgement greater or less than the biases present in algorithmic decisions?

Another very topical issue raised on the concept of ethics in software engineering was the concept of misinformation. Machine learning is used by social media platforms to decide what advertisements to show to what people. However, the algorithms take into account 'screen time' as a measure of success (Torres, 2018). This 'screen time' gives a bias to controversial, biased and inflammatory material. This allows the highly sensationalised 'fake news' to spread faster and farther than genuine news. This has the ability to influence political elections and other sensitive issues. We have seen this in the headlines after the Trump election campaign in America.

Another major example of ethics in software engineering is the question of how we program self-driving cars. Who is liable when accidents happen? How does the car choose what to do in equally catastrophic situations? These are all ethical questions that the software engineer must face in his/her work.

Privacy

Most pressingly at the moment is the question of privacy in data analytics research. The algorithmic techniques outlined in the previous section of this report cannot function without data. The lengths that we can go to in order to collect this data is another question of ethics.

For example, the Facebook and Cambridge Analytica Scandal caused a major stir in the field of data analytics. How far should firms be able to go to collect our data? How much data should we feel comfortable giving these companies? And what responsibilities to these companies have in order to keep our information safe? Developers may be faced with an ethical trade-off between copious amounts of data and detailed analysis and potentially breeching people's privacy.

As recently as this week, Apple's CEO Tim Cook speaking to media claimed that he felt regulation in the tech industry was "inevitable", in the midst of numerous data privacy scandals in the industry of late (Kuchler, 2018). He said: "This is not a matter of privacy versus profits, or privacy versus technical innovation. That's a false choice." (Cooke in Kuchler, 2018) It is now up to developers, researchers and managers alike to find the solution to this conundrum, as currently, it is not evident.

The major ethical question now is whether or not we can regulate these software developments (Torres, 2018). There are a lot of disagreements within society on how we should regulate for machine learning algorithms.

Recently, we have seen the introduction of GDPR legislation that has reformed the whole area of data privacy. GDPR provides for higher standards of data protection for individuals and increased obligations on organisations that process personal data (Citizens Information, 2018). Any organisation that collects or uses personal data (data that relates to, or can identify a living person), are now subject to GDPR regulations (ibid.). This may now include software engineers who wish to work on complex algorithms using the personal data of people. Therefore, software engineers will have to pay increases attention to such legislation moving forward.

As a software engineer, it goes without saying that the job involved wider responsibilities than the application of technical skills. Professional engineers must conduct themselves in an ethically and morally responsible way. Principles like confidentiality, honesty and integrity are key to the successful performance of a software engineer (Sommerville, 2011).

Conclusion

Of course, many of the points raised in this article come from a statistical, objective standpoint. However, it must also be noted that there are numerous soft-skills required by a software engineer that we cannot measure and quantify as easily (Hackermoon, 2018). For example, how can we measure the empathy of a software developer; their ability to relate to their team and find solutions that work for everyone? What about communication? How well can we measure a developer's ability to communicate with their team, management and the customer? A developer's patience, open-mindedness, creativity and time management are all other soft skills to which the metrics described above will not do justice. We should always keep this in mind when trying to rate the abilities and potential of a software engineer and the software engineering process within which they are working.

Bibliography

Carpenter, W., 2018. *Code Climate: How it Works and Makes Money.* [Online]
Available at: https://www.investopedia.com/articles/investing/022716/code-climate-how-it-works-and-makes-money.asp#ixzz5XCtdZq26
[Accessed 1 November 2018].

Citizens Information, 2018. *Overview of the General Data Protection Regulation (GDPR).* [Online]
Available at:
http://www.citizensinformation.ie/en/government_in_ireland/data_protection/overview_of_general_data_protection_regulation.html
[Accessed 15 November 2018].

Fenton, N. & Neil, M., 1999. Software metrics: successes, failures and new directions. *The Journal of Systems and Software,* Volume 47.

Github, 2018. *REST API v3.* [Online]
Available at: https://developer.github.com/v3/
[Accessed 3 November 2018].

Glass, R., 1994. A tabulation of topics where software practice leads software theory. *Journal of Systems Software,* Volume 25.

Hackermoon, 2018. *10 Soft skills every developer needs.* [Online]
Available at: https://hackernoon.com/10-soft-skills-every-developer-needs-66f0cdcfd3f7
[Accessed 14 November 2018].

Houlding, B., 2018. [Online]
Available at: www.scss.tcd.ie/Brett.Houlding/ST3011

Humphrey, W. S., 1995. A Discipline for Software Engineering. *Addison-Wesley.*

IEEE, 2018. *What is Computational Intelligence?.* [Online]
Available at: https://cis.ieee.org/about/what-is-ci
[Accessed 14 November 2018].

Johnson, P., 2003. Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined. *25th Annual Conference on Software Engineering.*

Johnson, P. M., 2013. Searching under the streetlight for Useful Software Analytics. *IEEE Sofware,* 30(4).

Kuchler, H., 2018. *Apple boss says tech industry regulation 'inevitable'.* [Online]
Available at: https://www.ft.com/content/b8bc0108-eb51-11e8-89c8-d36339d835c0
[Accessed 19 November 2018].

Lowe, S. A., 2018. *9 metrics that can make a difference to today's software development teams.* [Online]
Available at: https://techbeacon.com/9-metrics-can-make-difference-todays-software-development-teams
[Accessed 5 November 2018].

Lowe, S. A., 2018. *Why metrics don't matter in software development (unless you pair them with business goals)0.* [Online]
Available at: https://techbeacon.com/why-metrics-dont-matter-software-development-unless-you-pair-them-business-goals
[Accessed 12 November 2018].

Pomeroy-Huff, M., 2009. *The Personal Software Process, Body of Knowledge, Version 2.0.* s.l.:Carnegie Mellon.

Screenful Metrics, 2018. *Instant visual dashboards for GitHub issues.* [Online]
Available at: https://screenful.com/dashboard-for-github/
[Accessed 12 November 2018].

Sommerville, I., 2011. *Software Engineering.* 9th Edition ed. s.l.:Pearson Education.

Torres, O., 2018. *7 Short-Term AI ethics questions.* [Online]
Available at: https://towardsdatascience.com/7-short-term-ai-ethics-questions-32791956a6ad
[Accessed 2 November 2018].