Coláiste na Tríonóide, Baile Átha Cliath
Trinity College Dublin
Ollscoil Átha Cliath | The University of Dublin

Report on Measuring the Software Engineering Process

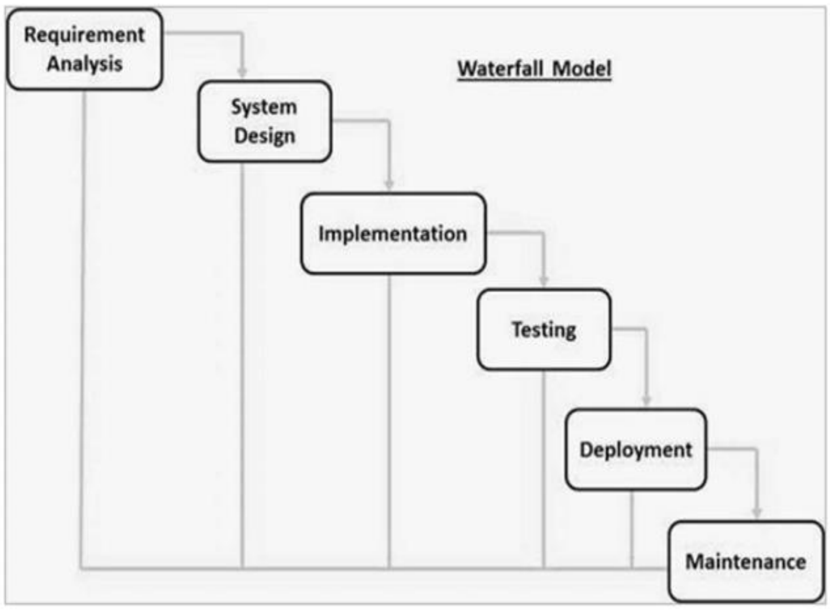
Colman Kinane | kinanec@tcd.ie | 16321208

Table of Contents

Introduction

To begin, we must first be familiar with what exactly software engineering is. According to the IEEE software engineering can be defined as the "application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software." This report will deal with how we measure this process and make it quantifiable.

Furthermore, we must know the software engineering process. The most widely known model is the Software Developing Life Cycles (SDLC) Model or Waterfall Model. This model clearly maps the flow of the processes involved in the development of software. It consists of 6 principle stages: *Requirement Gathering and analysis, System design, Implementation, Integration and testing, Deployment, Maintenance* (SDLC Waterfall Model, 2018). Each of these stages are sequential and generally the output of one phase serves as the input for the next phase as seen in Fig A.



(Fig A)

Measurement is vitally important for improvement and this is no different with software measurement as it is used to control and manage the software process to improve performance. Software projects consist of software engineers creating a product that satisfies the requirements of a client. But software projects are in their nature, not straightforward with Frederick P. Brooks giving a great insight into the reality, "It is usually innocent and straightforward in nature but is capable of becoming a monster of missed schedules, blown budgets, and flawed products" (Brooks, 1986). Measurement of software processes is not the silver bullet in preventing this issue but is a necessity if processes are to improve.

This report will deal with the topic of collecting measurable data through the use of metrics, the platforms used in industry to track this data and the algorithmic approaches to gain valuable insights from this data. We will also explore potential pitfalls to be found in these sections and the ethical implications of such analysis.

Measurable Data

"Not everything that counts can be counted. Not everything that is counted counts."
- Albert Einstein

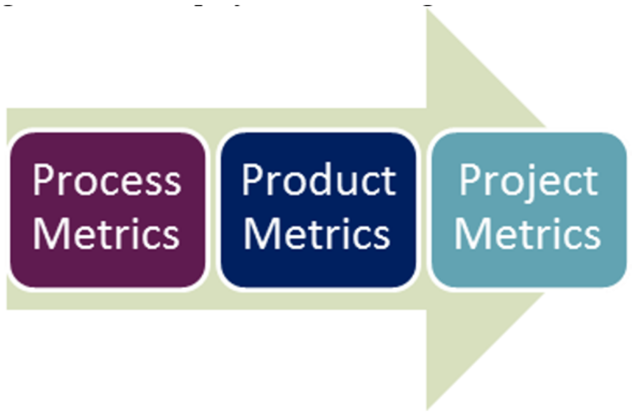
This quote is hugely relevant on the topic of software measurement. Before we think about collecting data we must be sure we know what data we need so that our analysis of said data will be value adding to our process and *counts*. The same sentiment shared by Einstein here is by Philip Johnson and is known as the streetlight effect and is a common pitfall when attempting to create useful software analytics. This pitfall is called the *streetlight effect* and in a software engineering context involves *"collecting and analysing metrics that are easily obtained with little social, political or developmental impact"* (Johnson, 2013).

Software Metrics

After a long teething period the subject of software metrics has been accepted into the mainstream. There are very few, if any, IT companies that don't implement them as the positive effects are widely known. It took years for the metrics to be adopted in industry after being researched at an academic level.

Software Metrics are measures which range from *"producing numbers that characterise properties of code through to models that help predict software resource requirements and software quality"* (Fenton & Neil, 1999).

Software metrics are used to provide information to management to support decision making and foster an environment of improvement. There are two principle types of software engineering metrics, Product and Process Metrics. Product metrics describes attributes of the code such as dimensions and physical size. Process metrics measure less quantifiable and measurable qualities of the process such as effort required, reusability as well as more measurable things such as commits and editing time. (Sillitti, Janes, Succi, & Vernazza, 2003) A combination of these divisions of metrics give the overall project metrics (Fig B).



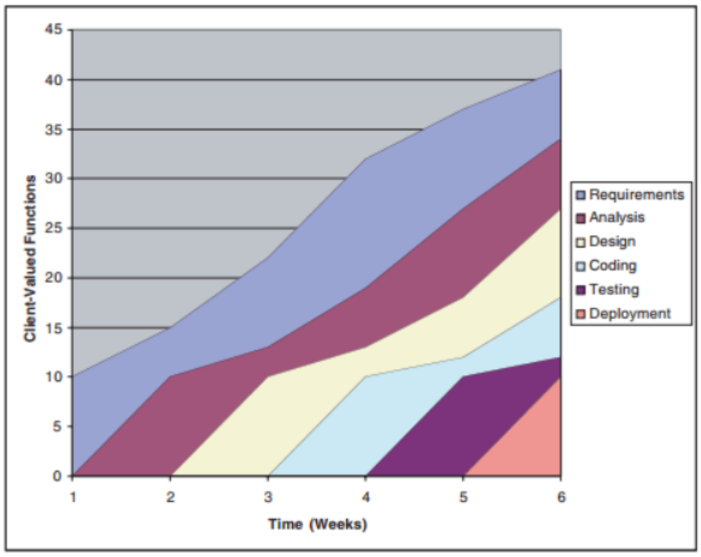
(Fig B)

It is advised to not use basic metrics in isolation as they are not telling of the full picture in most cases (Fenton & Neil, 1999). I will briefly mention a few common software metrics and their respective strengths and weaknesses.

Lines of code is the most straightforward method there is. Simply count the number of lines of code produced by a developer. This is a poor metric in isolation as developers can write long winded code that could be done in far less lines just to appease the metric. As Einstein said, "*Measuring programming progress by Lines of code is like measuring an airplane by how much it weighs*".

Code Churn is a useful metric for indicating when a software engineer is in a bit of bother or is struggling. This is because it shows that they are deleting a lot of code and recoding it. It is useful as it could be an indicator of the project running into a bigger problem and potential delays.

There are also some Agile process metrics which focus on how agile teams make decisions and plan. They do not directly describe the software but are used for improving the software process. Lead time is an example of this (Fig C). It quantifies how long it takes for ideas to be developed and delivered as software and is a key metric for companies as it measures how responsive they are to the demands of their customers.



(Fig C)

Techniques for gathering the data

So once it has been decided by the management what metrics would be most valuable to analyse for a team we can go about collecting them. We have tools for collecting data from the software, but that doesn't mean they are without problems. Collection is a time expensive task. This is particularly evident because of the deadline driven nature of software projects. Engineers do not want to be wasting precious time recording in their mind, arbitrary, data about their activities.

Additionally, manual data collection is unreliable and errors can badly effect the already costly analysis process. The latter of these problems will be dealt with in this report at a later point. One form of data collection proposed to gather the "right data" is the top-down, goal orientated Goal Question Metric (GQM) approach to gathering data which was developed by Dr. Victor Bassili (Software Metrics and Measurement, 2018). There are 3 levels to the model:

1. Conceptual: Goals are set here prior to metric collection.
2. Operational: Get the data to answer the goals.
3. Quantitative: Each goal has a set of data gathered to answer it and is analysed.

There are 7 total steps to this process which has greatly helped in making goals and metrics which are insightful and impactful and are as follows:

Seven Steps of GQM	
Step 1:	Develop a set of goals
Step 2:	Develop questions that characterise these goals
Step 3:	Specify metrics needed to answer the questions
Step 4:	Develop Mechanisms for data collection and analysis
Step 5:	Collect, Validate and Analyse the data
Step 6:	Analyse in post mortem fashion
Step 7:	Provide feedback to stakeholders

The Personal Software Process (PSP)

PSP is a structured software development process that was designed by Watts S. Humphrey to apply the principles of the Software Engineering Institute's Capability Maturity Model to a single developer.

The PSP was highly innovative in 3 ways:

1. It adapted organisational software processes to individuals.
2. It showed how these analytics could drive significant improvement
3. It presented the practices in an incremental fashion so that organisations could easily implement them.

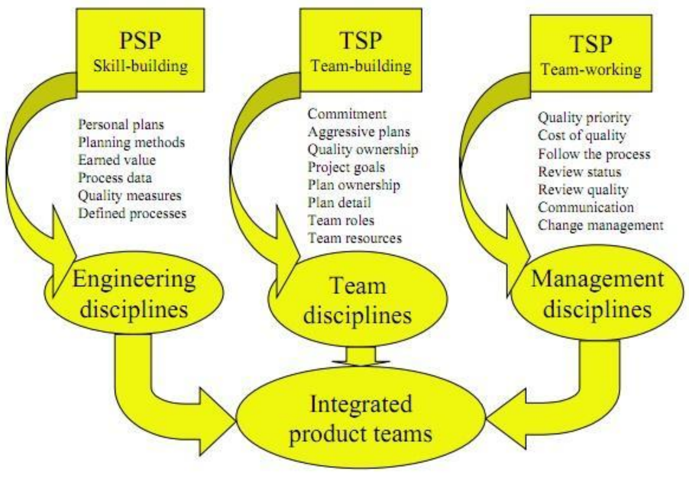
The PSP platform has 4 core measures: *Size, Effort, Quality and Schedule*. The key data collected in the PSP are time, defect and size data. The time data recorded how long each phase took, the defect data consisted of when and where bugs were found and fixed and the size data was simply the size of the individual parts. Developers used many other measures derived from these basic measures. Some of these derived measures include defect density, estimation accuracy and failure cost of quality.

In the first manual iteration of the PSP the user had to complete 12 forms from which yielded over 500 distinct values that were manually calculated. The manual nature of the initial PSP meant the system was highly flexible but also gave rise to several incorrect conclusions due to human error. Although to prevent these errors the PSP uses the PROBE method to improve a developer's skills for accurate project planning. A potential pitfall is that this relies hugely on the integrity and honesty of the developer in being forthcoming with truthful information.

The purpose of the PSP was to make the metrics used more focused and avoid the streetlight effect and improving performance that way. The original PSP can be seen as "lighting a candle" (Johnson, 2013) as it did not expel a huge amount of light comparatively to later iterations which built upon it but nonetheless was focusing in the correct areas.

THE PSP AND TSP

So as PSP was adapted to make it applicable to single software engineers the skills learned transition into the Team Software Process (TSP). TSP teams consist of PSP trained developers and as such the team is self-managed (Fig D). In a study by Watts Humphrey he found that 1/3 of all software projects fail but a similar study by the SEI of TSP led software projects found the failure rate to only be at the 6% mark. (Davis & Mullaney, 2003) This success in part can be attributed to using historical data to make better estimations, so that projects are realistic. To build on this as the TSP is built on the PSP there is a lower rate of software defects.



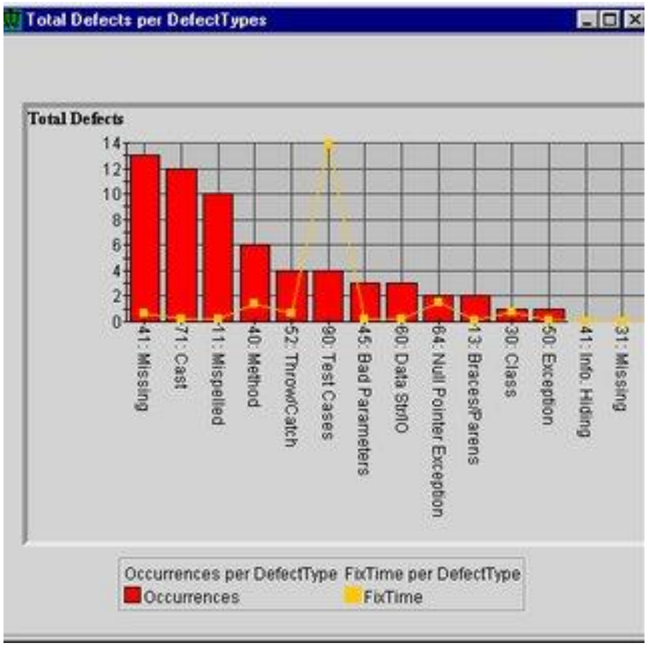
(Fig D)

Computational Platforms

The highly time intensive nature of manual data collection and calculations in models such as the highly effective PSP make them unattractive to software engineers and affects the adoption of the model. So the methods underlying the systems work but the time and effort involved are far from ideal. This paved the way for the recording and calculations to become automated. There are now a variety of effective tools used for this purpose used by software companies the world over.

LEAP

The Leap Toolkit was an attempt to address some of the problems out lined above with the PSP by "automating and normalising data analysis" (Johnson, 2013). The developer still manually enter most of the data but the toolkit automates subsequent PSP analyses and in some cases provides analysis. The system is portable meaning developers can bring their information from project to project. Fig E shows that the analysis done by the LEAP toolkit unlike the classic PSP is far from paper based and is displayed nicely and charts making data readable and meaningful.



(Fig E)

Hackystat

When development of the LEAP toolkit was stopped because it could never have sufficient ROI due to it not fixing the problem of manual data entry in the first place. Hackystat was the next iteration in trying to solve this problem. It is *an "opensource framework for collection, analysis, visualisation, interpretation and dissemination of the software development process and product data"*. (Hackystat, 2018)

Users of hackystat attach sensors to their development tools, these unobtrusive sensors collect and send 'raw' data about development to a web service called the Hackystat sensor base where the data is stored. This base is queried and abstractions made generating visualisations of the data among other things.

There are 4 important design features that make it a valuable framework:

1. There is both client-side and server-side data collection
2. The data collection is inobtrusive
3. Fine grained data collection
4. Can be used for personal and group-based development

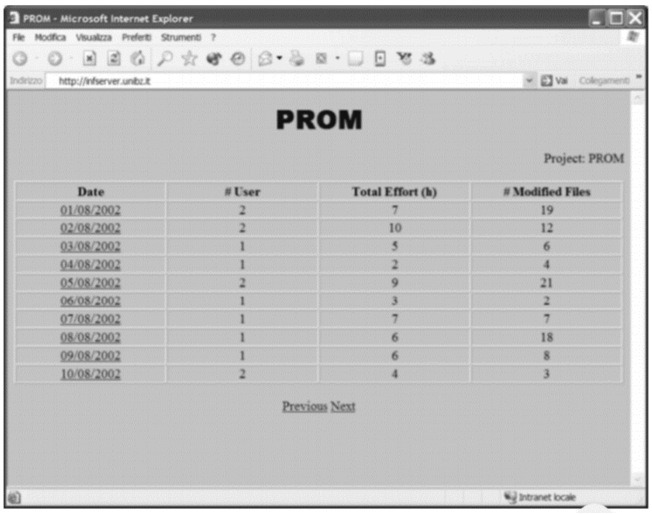
Hackystat becoming used led to a variety of innovations and advancements in the field of measuring software processes but did not come without its problems. Some developers did not appreciate the level and volume of data collection happening without telling them about it. This affected whether developers would download the framework, and many viewed it as a bug.

It was also discovered that hackystat could lead to discord among teams of programmers. The fine-grained nature of the data collected and the immense transparency let people "Stalk" each other and led to conflicts etc when certain members were not pulling their weight. On a similar note it was widely reported that developers were not comfortable with the level of information available to their management about their work habits. (Johnson, 2013)

The greatest weaknesses of Hackystat was that it provided too much data to people that the developers themselves were not happy with and that because the whole process was considered unobtrusive they were not sure of the data collected about them. (Johnson, 2013)

PROM

PRO Metrics (PROM) is a tool for automated data acquisition and analysis that collects both code and process measures. Collected data include all PSP metrics, procedural and object orientated metrics and some ad-hoc developed metrics. It collects and analyses data at different levels of granularity: personal, workgroup and enterprise. This preserves a developer's privacy providing managers only aggregated data. This fixes some of hackystats problems regarding privacy. (Fig F+G) (Sillitti, Janes, Succi, & Vernazza, 2003)


PROM - Microsoft Internet Explorer
File Modifica Visualizza Preferiti Strumenti ?
Indirizzo http://infserver.unibz.it
Vai Collegamenti
Intranet locale

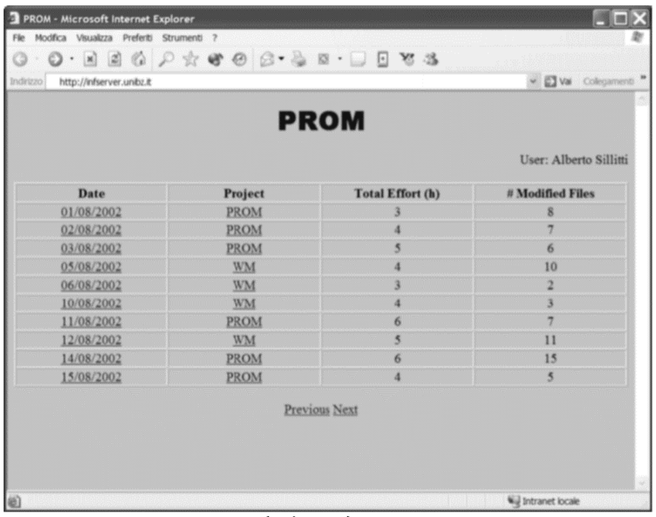
PROM

Project: PROM

Date	# User	Total Effort (h)	# Modified Files
01/08/2002	2	7	19
02/08/2002	2	10	12
03/08/2002	1	5	6
04/08/2002	1	2	4
05/08/2002	2	9	21
06/08/2002	1	3	2
07/08/2002	1	7	7
08/08/2002	1	6	18
09/08/2002	1	6	8
10/08/2002	2	4	3

Previous Next

(Fig F)


PROM - Microsoft Internet Explorer
File Modifica Visualizza Preferiti Strumenti ?
Indirizzo http://infserver.unibz.it
Vai Collegamenti
Intranet locale

PROM

User: Alberto Sillitti

Date	Project	Total Effort (h)	# Modified Files
01/08/2002	PROM	3	8
02/08/2002	PROM	4	7
03/08/2002	PROM	5	6
05/08/2002	WM	4	10
06/08/2002	WM	3	2
10/08/2002	WM	4	3
11/08/2002	PROM	6	7
12/08/2002	WM	5	11
14/08/2002	PROM	6	15
15/08/2002	PROM	4	5

Previous Next

(Fig G)

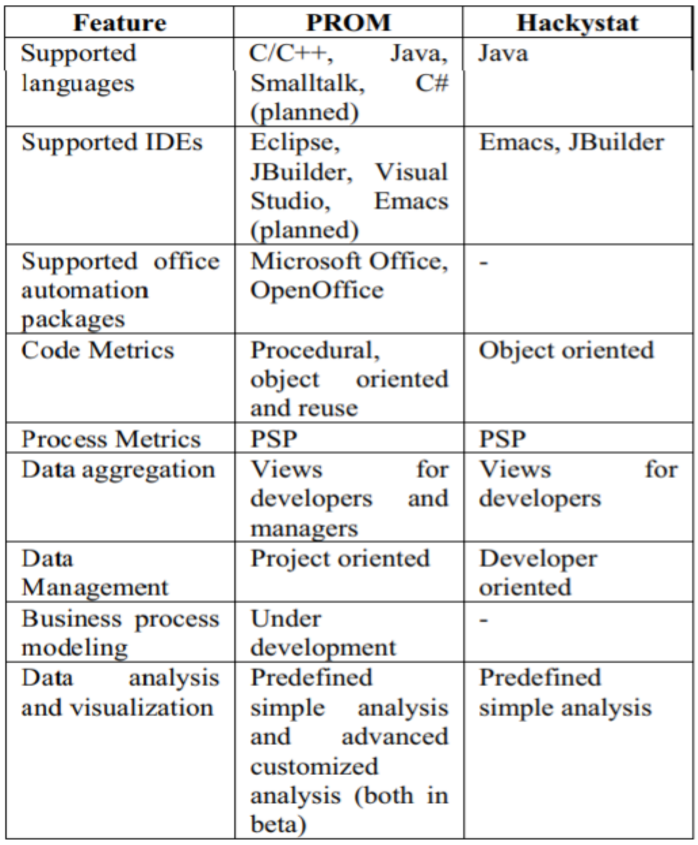
The chosen architecture allows both managers and users to maximise benefits of data acquisition. The benefit for developers is code and process improvements and the benefit for management is easier cost management and control on project evolution for managers. It does all of this on a need to know basis and therefore respects the privacy of the developers. (Sillitti, Janes, Succi, & Vernazza, 2003)

PROM vs Hackystat

Fig H briefly summarises the main features of PROM and Hackystat.

From this table we can see that hackystat focuses on the coding activity and the target users are the developers. PROM is more well-rounded and analyses the whole development process

Hackystat is better suited to the improvement of the performances of a single developer whereas as PROM is better for a detailed overview of the whole process.



Feature	PROM	Hackystat
Supported languages	C/C++, Java, Smalltalk, C# (planned)	Java
Supported IDEs	Eclipse, JBuilder, Visual Studio, Emacs (planned)	Emacs, JBuilder
Supported office automation packages	Microsoft Office, OpenOffice	-
Code Metrics	Procedural, object oriented and reuse	Object oriented
Process Metrics	PSP	PSP
Data aggregation	Views for developers and managers	Views for developers
Data Management	Project oriented	Developer oriented
Business process modeling	Under development	-
Data analysis and visualization	Predefined simple analysis and advanced customized analysis (both in beta)	Predefined simple analysis

(Fig H)

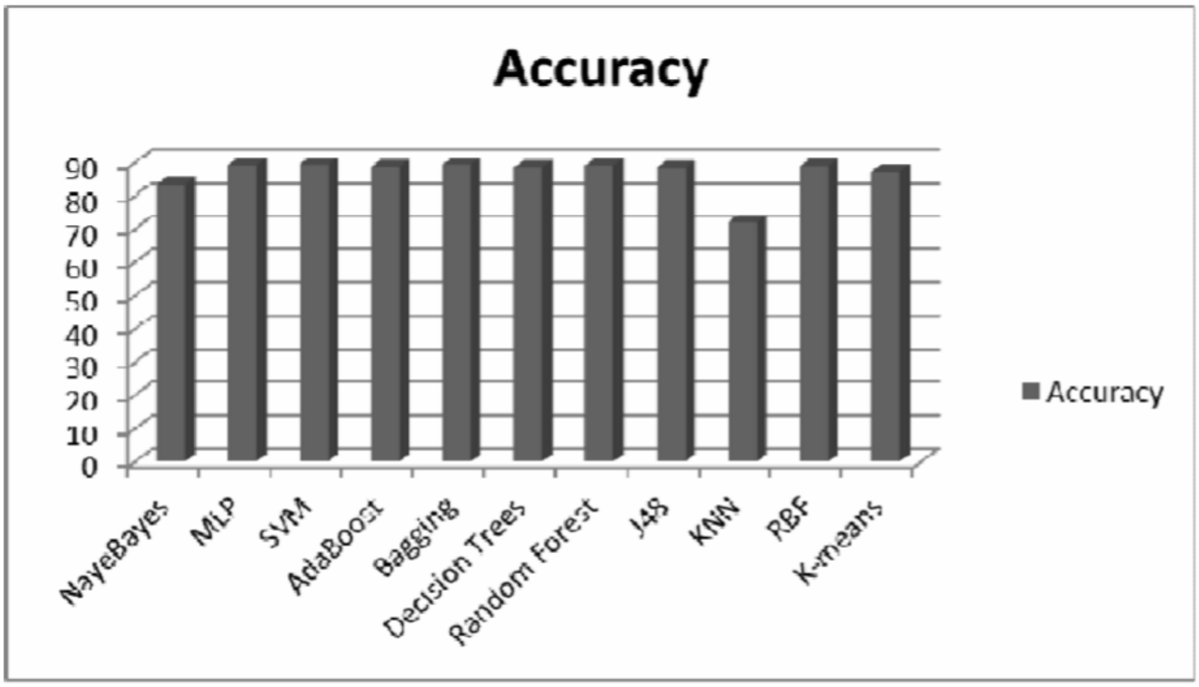
ALGORITHMIC APPROACHES

An algorithm is an unambiguous specification of how to solve a class of problem. They are heavily used in calculations and data processing. (Algorithm, 2018) Algorithms are used in software engineering measurement and the recent growth in the strength of machine learning algorithms is a great potential avenue for analysing data. Deciding what code is good and what is bad automatically is not a straightforward exercise and is not doable with one simple algorithm. Machine Learning is a section of AI that assumes that a system can evolve and learn from data, identify patterns and make decisions without human input. (Faggella, 2018)

Machine Learning techniques can be used to detect bugs in data sets through the following methods: Clustering and Classification. Classification involves dividing software into defective and non-defective by use of a classification model of software complexity metrics such as cyclomatic complexity and code size. Clustering makes assumptions about a data set and if the assumptions hold true the that results in a good cluster. (Aleem, Capretz, & Ahmed, 2015)

In the paper *Benchmarking machine learning techniques for software defect detection* the two most successful Machine Learning methods for identifying bugs in large sets of software modules were a Support Vector Machine (SVM) and a Multi-Layer perceptron (MLP). A SVM uses non-linear mapping to train the data. It then searches for the optimal plane for separation. It is a supervised learning based and is used for classification purposes. (Aleem, Capretz, & Ahmed, 2015) A MLP is quite different as it is a supervised learning approach but makes use of neural networks.

For training purposes, it uses a technique called backpropagation. In Fig I, we can see the accuracy results for various machine learning methods detecting bugs in large datasets. Despite none of them being perfect 100% detection rate it clearly shows the use of machine learning in analysing the software modules.



(Fig I) (Aleem, Capretz, & Ahmed, 2015)

Ethical Implications

There are a few ethical implications to consider when measuring the software engineering process, some of which have already been touched upon.

One that was touched upon earlier is employee wellbeing and a positive work environment. With all this data being gathered about employees it is important to consider the implications it will have on the individuals. Some individuals may not mind the process, but it might adversely affect others. If an employee’s data is accessible by colleagues this is another opportunity for negative consequences. The accountability the system such as hackystat might have will be more than offset by the worsened mental state of an employee who is constantly worrying he is being watched. There is great potential for internal friction among teams and with management if too much information is passed to people who don’t necessarily need to know. This ties in with privacy and the right the developer has to a degree of privacy and as such none of their private information should be on these systems as it is not necessary for the companies to track his performance. It is important that a company respect the employee and only do what they have been permission to do.

There is concern for the security of the employee information and data gathered by the platforms companies use. Without proper security protocols company’s data and employee’s private information may be breached. When dealing with data the upmost respect and prudence should be taken in regards the employee.

Conclusion

It is clear from this report that measurement of the software engineering process is not perfect but has been at the centre in a range of vast improvements in the process to date. The field will continue to improve as technology advances. The prospect of machine learning becoming far better is not that distant and this I believe is the next area for a breakthrough in the software engineering measurement field. Organisations should not limit themselves to one strict approach to measuring the process, but all the available approaches mentioned in the report and combining the results would yield more well-rounded results. I would also echo a constant theme of the report that gathering data for the sake of it is not beneficial and organisations and teams should use systems such as GQM and TSP to ensure they remain focused on the important issues and not under the streetlight. The ethical implications should continue to be monitored closely in this age where data is becoming all powerful as more and more of it is collected.

Bibliography

Aleem, S., Capretz, L. F., & Ahmed, F. (2015). BENCHMARKING MACHINE LEARNING TECHNIQUES. *International Journal of Software Engineering & Applications (IJSEA), Vol.6, No.3* .

Algorithm. (2018). Retrieved from Wikipedia: https://en.wikipedia.org/wiki/Algorithm

Brooks, F. P. (1986). No Silver Bullet - Essence and Accident in Software Engineering.

Davis, N., & Mullaney, J. (2003). The Team Software Process (TSP) in Practice: A Summary of Recent Results. *SEI - Software Engineering Process Management Program*.

Ebert, C., Dumke, R., Bundschuh, M., & Schmietendorf, A. (2005). *Best Practices in Software Measurement*. Berlin: Springer-Verlag Berlin Heidelberg.

Faggella, D. (2018, October 29). *What is Machine Learning?* Retrieved from techemergence: https://www.techemergence.com/what-is-machine-learning/

Fenton, N. E., & Neil, M. (1999). Software metrics: successes, failures and new directions. *The Journal of Systems and Software 47* , 149-157.

Hackystat. (2018). Retrieved from Github.io: https://hackystat.github.io/

IEEE - Standard Glossory of Software Engineering Terminology. (1990, December 31). Retrieved from IEEExplore.ieee.ord: https://ieeexplore.ieee.org/document/159342

Johnson, P. M. (2013, August 4). Searching Under the Streetlight for Useful Software Analytics. *IEEE Software*, pp. 57-66.

SDLC Waterfall Model. (2018). Retrieved from Tutorials Point: https://www.tutorialspoint.com/sdlc/sdlc_waterfall_model.htm

Sillitti, A., Janes, A., Succi, G., & Vernazza, T. (2003). Collecting, Integrating and Analyzing Software Metrics and Personal Software. *Proceedings of the 29th Euromicro Conference. IEEE*, 336-342.

Software Metrics and Measurement. (2018). Retrieved from Wikiversity: https://en.wikiversity.org/wiki/Software_metrics_and_measurement