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

CS3012 Software Engineering
Measuring Software Engineering Report

Kamil Przepiórowski
17327895

November 5, 2019

Contents

1 Introduction **1**

2 Measurement and Assessment **2**
2.1 Why should we measure? 2
2.2 What can we measure? 2
2.3 How do we assess the data? 3

3 Analytical Platforms **3**
3.1 Personal Software Process 3
3.2 Hackystat 4
3.3 GitPrime 4

4 Algorithmic Approaches **5**
4.1 Neural Hidden Markov Model 5
4.2 Cyclomatic Complexity 6
4.3 Computational Intelligence 7

5 Ethical Considerations **8**

6 Sources **9**

1 Introduction

The aim of this assignment is to deliver a report that considers the ways in which the software engineering process can be measured and assesed in terms of measurable data, an overview of some analytical platforms available to perform this work, the algorithmic approaches available, and the ethical concerns surrounding this kind of analytics. The goal of the measurements is to make incremental improvements to processes and production environments, and to have truly data-driven software development.

2 Measurement and Assessment

2.1 Why should we measure?

Software engineering is an extremely complex process. The team has to worry about efficiency, reliability and the stability of the product, along many other things. By applying measurement to different areas we can identify where change is needed. Numbers related to up-time, service availability, budget adherence are important, but they fail to tell you the complete story of the engineering teams performance and product health.

"Software measurement is a baseline component of good software engineering. You can understand exactly when your development team does their best work and what factors contribute to that." [1]

It is a common perception that you can't you "can't improve, what you can't measure". [2] However we encounter many problems once we look deeper into the measurement process. What can we even measure about software engineers? How do we determine what is a useful measurement? How can we use the data to "improve" performance?

2.2 What can we measure?

Finding the best individual software metrics to track is extremely challenging. We shouldn't consider data useful just because it is easy to obtain, but then how do we determine which measurements are worthwhile and which ones can give us valuable data? My definition of 'valuable data' is meaningful and useful data, i.e. data which can be analysed to give meaningful results, which are relevant to our goals. To choose the optimal software metrics to measure we should rely on three principles:

- You can effectively measure some area of application development or process.
- A relationship exists between what can be measured and what you want to learn.
- This relationship can be validated and expressed in terms of a formula or a model.

Some of the software metrics which are currently being measured in the industry [1] [3] are:

- Amount of Commits
- Frequency of Commits
- Lines of Written Source Code
- Code Coverage
- Sprint Burndown - complexion of work throughout the sprint based on story points
- Bug Rates - average number of bugs that come up as new features are being deployed
- Mean Time To Repair
- Mean Time Between Failures
- Leadtime - time between the definition of a new feature and its availability to the user
- Cycle time - time between when work is started on an item and its completion.
- Open / close rates - how many production issues are reported and closed within a specific time period

There is some ambiguity over what the definition of 'software productivity' really is. In this report I define software productivity as "the ratio between the functional values of software produced to the efforts and expense required for development".[1]

2.3 How do we assess the data?

Productivity metrics for software development help identify what factors hinder the effectiveness of a team and eliminate those, which would ultimately lead to a happier, high-performing team. We can compare the costs and benefits of certain practices to determine which is worth the cost. Or we can benchmark two different practices to choose the better approach. There are many Platforms and Algorithmic Approaches to analysing this data, more on those in the following sections of the report.

3 Analytical Platforms

3.1 Personal Software Process

Watts Humphrey created this platform to apply the underlying principles of the Software Engineering Institute's (SEI) Capability Maturity Model (CMM) to the software development practises of a single developer. [4]

Personal Software Process (PSP), is a platform designed to help software engineers better understand and improve their own performance by using sound engineering practices. "PSP shows software developers how to plan and track their projects, use a measured and defined process, establish goals, and track their performance against these goals." [5] It shows developers how to manage the quality of their own products. PSP assists engineers in managing software quality throughout the whole development process, evaluating the results of each completed task and using the results to suggest improvements for the next one. PSP is designed for use with any programming language or design methodology and it can be used for most aspects of software work, including writing requirements, running tests, defining processes, and repairing defects.

The Personal Software Process is designed to help software engineers to:

- Manage the quality of their projects.
- Improve their estimating and planning skills.
- Make commitments and schedules they can keep and meet.
- Reduce the amount of defects in their projects.

PSP follows an evolutionary improvement approach: an engineer learning to integrate the PSP into their process begins at the lowest level – PSP0 – and progresses through the levels until the final level - PSP2.1 - is reached. Each level is different and has detailed scripts, checklists and templates to guide the engineer. Humphrey encourages engineers to personalise these scripts as they get a better understanding of their individual strengths and weaknesses. [4]

PSP0, PSP0.1

- PSP0 has 3 phases: planning, development and a post mortem. The following are measured: time spent on programming, faults injected/removed, size of a program. In a post mortem, the engineer ensures all data for the projects has been properly recorded and analysed.
- PSP0.1 advances the process by adding a coding standard, a size measurement and the development of a personal process improvement plan (PIP). In the PIP, the engineer records ideas for improving his own process.

PSP1, PSP1.1
•PSP1 - Based upon the baseline data collected in PSP0 and PSP0.1, the engineer estimates how large a new program will be and prepares a test report.
•PSP1.1 - Accumulated data from previous projects is used to estimate the total time. Each new project will record the actual time spent. This information is used for task and schedule planning and estimation.

PSP2, PSP2.1
•PSP2 adds two new phases: design review and code review. Defect prevention and removal of them are the focus at the PSP2. Engineers learn to evaluate and improve their process by measuring how long tasks take and the number of defects they inject and remove in each phase of development. Engineers construct and use checklists for design and code reviews.
•PSP2.1 introduces design specification and analysis techniques.

To maximise the effectiveness of PSP, engineers should plan their work and base these plans on their personal data. To improve their performance, software engineers should personally use regular and well-defined processes.

Software engineers should feel personally responsible for the quality of the products they are making. For engineers to understand their performance, they must measure time spent on each step of a project, defects injected and removed, and the size of the software products they produce. PSP provides software engineers with a self-directed and disciplined personal framework for doing quality work.

3.2 Hackystat

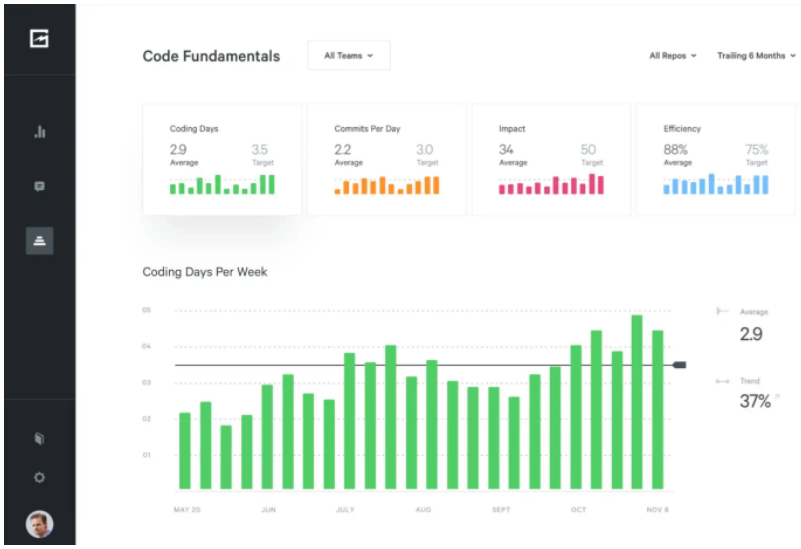
Hackystat is a framework for collection, analysis, visualization, interpretation, annotation, and dissemination of software development processes and product data. [6] It is no longer in active development but it is open source, with it's code being widely available on GitHub. [7] The aim of the platform is to provide a mechanism that is extendable which can reduce the overhead associated with collection of data about software engineering. Hackystat implements a service oriented architecture in which sensors attached to development tools gather process and product data and send it to a server, which other services can query to build higher-level analyses. [8]

Hackystat does both client and server-side data collection. Another feature is that the data collection is unobtrusive. It doesn't make assumptions about connectivity and Hackystat client-side instrumentation locally caches any data collected while a developer works offline. Hackystat also supports group measurements and can track when multiple engineers are working on the same project.

3.3 GitPrime

GitPrime is a platform which focuses on the analysis of source control, specifically Git. It allows you to "draw insights out of your git repos and measure your team's success." [9] With GitPrime you can spot and better understand trends in your team's development process, It is a software solution for measuring software. GitPrime aggregates historical git data into easily understandable insights and reports.

The main incentive of GitPrime is how easily understandable the data is. It visualises things such as pull requests, commits, team dynamic graphs, team's contributions and work habits. You can clearly see all the metrics and understand how the team is performing. Past performance, predicts future performance, and with the information it provides, GitPrime allows for data driven development.



The platform gets rid of a lot of ambiguity and allows you to spot bad trends and bottlenecks early. You can tell exactly who is contributing to the project and how they're doing, which allows you to plan better sprints and reduce cycle time. You can tell if code reviews are positive or negative, observe patterns in the code review process and see an overview of each pull request.

GitPrime has different metrics for engineers reviewing code and submitting it. [10] The reviewers are measured on: Responsiveness, Comments Addressed, Receptiveness and Unreviewed PRs. Subitters are measured on: Time to First Comment, Influence, Review Coverage and Reaction Time. Engineers are also measured on metrics such as: Average Time to Resolve, Recent Ticket Activity Level, Average Number of Follow-on Commits, Coding Days, Commits Per Day, Impact and Efficency. All of the measurements are visualised and easily accessible.

4 Algorithmic Approaches

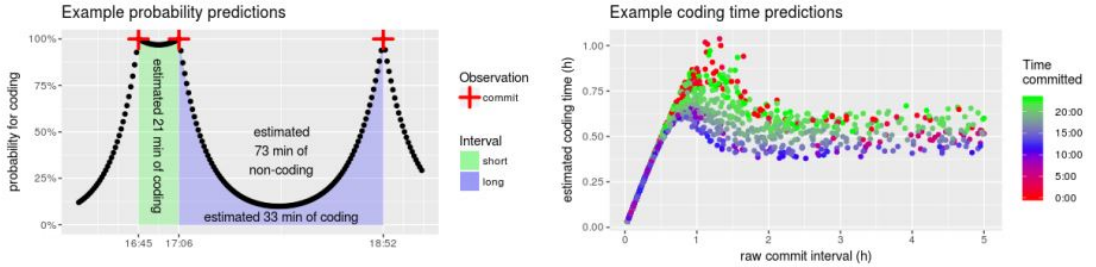
4.1 Neural Hidden Markov Model

We can train a Neural Hidden Markov Model to predict the probability that an individual developer is currently coding for each 1 minute of their recent history. We can hence compare the expected to actual coding time between two successive commits. [11]

There are two states in the model: coding and non-coding. We observe the timestamped sequence of commit events - a commit can only happen during coding (with a trained probability C per minute). Developers end coding within the next minute with probability E(t), and non-coding developers start with a probability S(t). In contrast to classic Markov models, the probabilities E and S vary over time and are affected by trends and

habits. A simple neural network (NN) with 5 inputs – the sine and cosine of the angle of an imaginary dial on a day-long and week-long clock, and normed overall time – supplies the probability values. These features can encode daily or weekly patterns that may shift over time. The NN is then trained by back propagating likelihood gradients through the HMM part.

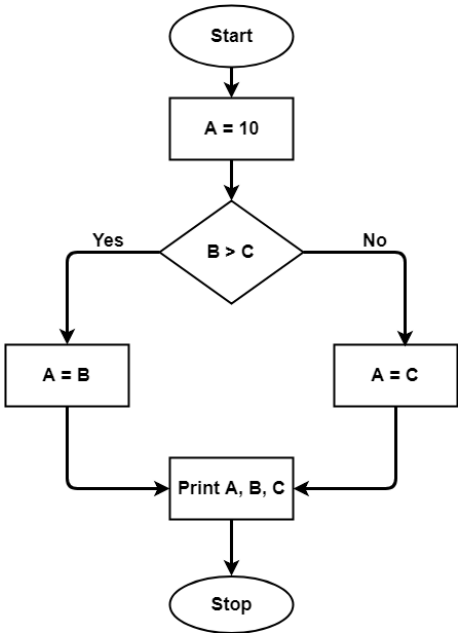
The neural HMM is a flexible solution to a general problem: for any process that alternates between active and inactive phases, it infers the probability of being active at any time from an observed rhythm of potentially infrequent 'life signs' (in this case commit events). The neural HMM can discover night-time coders, or developers that code only on weekdays, or weekends etc. The image below is a model that maps commit intervals to estimates of coding time.



4.2 Cyclomatic Complexity

Cyclomatic complexity of code is the quantitative measure of the number of linearly independent paths in it. It is a software metric used to indicate the complexity of a program. It's computed using the Control Flow Graph of the program. The nodes in the graph indicate the smallest group of commands of a program, and a directed edge in it connects the two nodes i.e. if second command might immediately follow the first command.

A sample control flow graph may look like this:



Mathematically, the formula is [12]:

$$\mathbf{M = E - N + 2P}$$

where,

M = cyclomatic complexity
E = the number of edges in the control flow graph
N = the number of nodes in the control flow graph
P = the number of connected components

Measuring complexity of a program is not a trivial task, but cyclomatic complexity is generally a good approach. Why do we even bother measuring code complexity? Well, it is a very valuable metric which can provide a lot of context to other metrics such as LOC or commit frequency - a more complex problem might take longer to solve, or perhaps a really short bit of code is in fact extremely complex. Without context these measurements would be skewed and not represent the reality of the situation. Understanding data correctly is as important as the measurements themselves.

4.3 Computational Intelligence

"Computational Intelligence (CI) is the theory, design, application and development of biologically and linguistically motivated computational paradigms." [13] Though CI is seen as an umbrella under which more and more methods are slowly being added, and as such, a concrete definition of Computational Intelligence is impossible. With the recent spike of interest in Deep Learning and Artificial Intelligence (AI), many companies are looking at CI to see if they can implement a system which will automatically make decisions and give feedback based on some input metrics. CI is said to be the "devoted to solution of non-alogrithmizable problems." [14] CI experts focus on problems that are difficult to solve using artificial systems, but are solved by humans, problems requiring intelligence.

CI is a subset of Artificial Intelligence (AI) which mainly consists of three pillars [15]:

• Neural Networks
Neural Networks are massively parallel distributed networks that have the ability to learn and generalize from examples. They are used for data analysis and classification, associative memory, clustering generation of patterns and control. Neural Network techniques share with the fuzzy logic ones the advantage of enabling data clustering.

• Fuzzy Systems
Fuzzy Logic can face incompleteness, and most importantly ignorance of data in a process model, in contrast to AI, which requires exact knowledge and values. Fuzzy Systems solve uncertain problems based on a generalization of traditional logic, which enables us to perform approximate reasoning.

• Evolutionary Computation
Using the biological evolution as a source of inspiration, Evolutionary Computation (EC) solves optimization problems by generating, evaluating and modifying a population of possible solutions. EC includes genetic algorithms, evolutionary programming, evolution strategies, swarm intelligence and more.

5 Ethical Considerations

There are many ethical concerns around the topic of measuring software engineers. Personally, I think the biggest one is that few enjoy the thought of their productivity being tracked; and, indeed, these metrics may be abused and misinterpreted. Are the software engineers aware that they are being measured? If so, to what extent? Are they told about every single metric that's being kept track of or are some kept hidden? Is the collection of data unobtrusive? These are all questions which should be considered when measuring engineers as they can all affect their behaviour.

If the software engineers are fully aware of the fact that they are being tracked, this may well affect their behaviour, and create a more toxic environment where the metrics are being abused. People act differently when they are being measured, and this was famously exposed in the Stanford Prison Experiment. [16] It isn't certain that the engineers would be as affected as the students in the experiment, but it is extremely likely that their behaviour would change - possibly for the better or worse. The metrics could create a competitive atmosphere where engineers are competing with each other to see who can get the best performance, but personally I think this would only create a toxic environment in the work-space. There is a very high probability that the engineers behaviour would change in order to fit the metrics better. This could be very easily abused if the metrics are taken as an average of the team. Imagine an engineer who has extremely good statistics on a very average team, would this engineer be artificially inflating the performance of the others? Would it be ethical for this engineer to artificially decrease his productivity so that he can fit in more with the average without any punishment?

This brings up another issue, specifically, crab mentality. This mentality could be best described with the phrase "if I can't have it, neither can you". [17] What's the incentive for an engineer to improve their performance, if they can harm someone else's and bring down the average so that they aren't exposed. Not only that, if we take the wrong metrics, engineers can pamper to those rather than actually increasing their productivity. For example, if we take a look at the Lines Of Source Code as a metric in isolation, the more lines a developer has the better. This of course isn't always the case, and this kind of measurement only motivates the developer to write poorly optimised and long code in order to boost their statistics. Another example could be Bugs Fixed per commit, an engineer could intentionally put bugs into the code only so that they could fix them in the following commit and get a boost in their statistics. Clearly this is a big problem, and if it's possible, it will happen. Therefore we need to define some sort of average which can't be affected by artificial inflation / deflation of certain metrics of other engineers, and we also need to do significant research into what metrics are useful and worthwhile. Not only that, we need to make sure the measurements are looked at with context and not in isolation, as raw data may not always tell the full story. As I said previously, understanding the data is as important as the measurements themselves.

We also need to keep in mind who has access to the data. Is it only available to the individual engineer? To their manager? To everyone on the team? How can we trust whoever has access to the data to interpret it correctly and use it beneficially? Even if we are measuring the correct things, they can still be interpreted incorrectly, not only by whoever is looking at the data, but what if the algorithms used to analyse the data are biased in some sense? Is it ethical to trust Artificial Intelligence to judge our perfor-

mance at work and make decisions over who may or may not be fired. Another thing we need to consider is where to draw the line, what's considered too far? It's obvious that things external to the work-space can affect an engineer's performance. Perhaps a loved one is sick, or something private is going on at home. Clearly, these things will affect an engineers mood and productivity, but we shouldn't be monitoring people's personal life only so that they can be more productive at work and make the company more money. We need to define limits,and personally I think measuring people's personal lives is too far.

Ultimately is it even ethical to track someone like this or should employees be entitled to having some personal freedom in the office itself. Does this sort of measurement have any benefits to the work-life of the engineers? Personally I think it could, if done right. In an ideal situation the measurements would only benefit the engineer, they would be accurately told what they're doing right and what they need to improve on. This would maximise individual performance and help developers reach their fullest potential. If handled correctly there would be no internal toxicity between team members to try and better each other just to appeal to the algorithms more. However this is all hypothetical, in a real world environment things will not be as perfect, and there will be issues. I don't have the answers on how to fix these issues, but based of current research being done and the platforms available, it looks like we could be heading in a good direction.

6 Sources

[1] Top 10 software development metrics
[2] Measuring employee performance
[3] 9 Software Metrics
[4] Personal Software Process
[5] Personal Software Process
[6] Hackystat
[7] Hackystat GitHub
[8] Hackystat
[9] GitPrime
[10] GitPrime Metrics
[11] Neural Hidden Markov Model
[12] Cyclomatic Complexity
[13] Computational Intelligence
[14] Computational Intelligence
[15] Computational Intelligence
[16] Stanford Prison Experiment
[17] Crab Mentality

Background Reading:

- Software Metrics
- McKinsey on Impact social technologies
- Turning Software Development Into a Game
- Useful Software Analytics
- Machine Learning Approach
- Algorithmic Approaches