25/11/20
CSU33012

Measuring Software Engineering

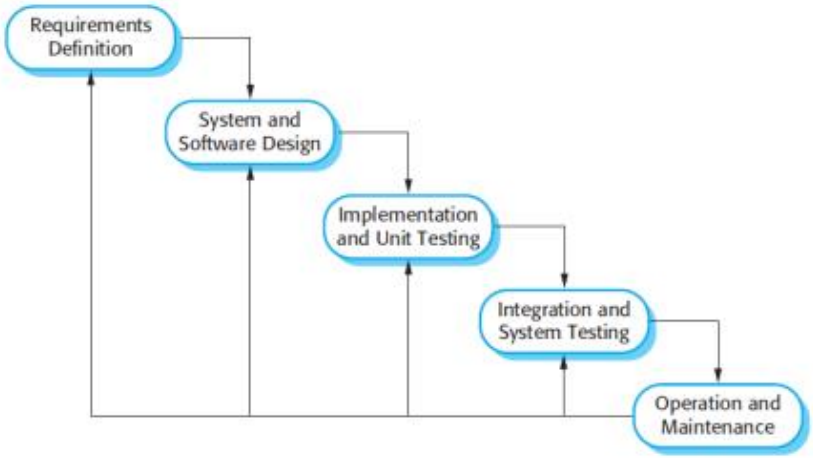
Harry O'Brien
18323075

Abstract

The purpose of this report is to provide a broad overview of the various ways in which processes in software engineering can be assessed and evaluated. I would answer this through the following topics: observable results, open computing platforms, accessible algorithmic methods and the ethical problems surrounding this method of research. A selection of scholarly material as well as the content addressed and cited in the CSU33012 lectures will be used to assemble this paper.

Introduction

Software Engineering is defined as "*the systematic application of engineering approaches to the development of software*" (Wikipedia, n.d.). It is basically the application of classical engineering procedures to the creation of software. The need emerged for a means of calculating and analyzing the software engineering process due to the exponential development of the industry. The method includes a sequence of activities leading to software creation, and the procedures below must be carried out independently of the project at hand.



This series of operations is usually referred to as the Software Development Life Cycle (SLDC). Each of these behaviors can be applied in several SLDC models, with waterfall, agile and iterative tending to be the most common. It may not be said the process is better than the others, but the suitability of a process would depend on the type of project (Osetskyi, 2017).

The creation method is linear and predefined in the waterfall model. This simplifies project management, but the software isn't finished until the very end. The agile model helps the client to view the results and at each step of development, approve/disapprove the program. By incorporating correction and technical criteria into the production phase, this strengthens software but raises the possibility of new specifications interfering with the current architecture. Iterative models create various system

components in stages, connecting each component to the previously formed components. The method is repetitive, causing every cycle to produce new versions of the product. It makes for better risk assessment but can be more difficult to manage than most procedures. As mentioned above, although procedures can vary slightly, the key steps above must be carried out.

Measuring Software Engineering requires obtaining information, reviewing the data and ultimately seeing the sum of output with a given amount of input received. More production with the same input quantity clearly implies higher productivity. Measuring progress in software engineering, especially in the early stages of development and for innovate products, is challenging.

In this report, I will try to breakdown the task of measuring software engineering into 4 chapters:

1. Measurable Data
2. Existing Solutions
3. Algorithmic Analysis
4. Ethical Concerns

Before I discuss *how* software engineering is measured, it's important to first clarify *why* software engineering is measured.

Why measure Software Engineering?

In reaction to the 'Software Conference' at the time, the first Software Engineering conference was organized and sponsored by NATO in 1968. The core subject of this conference was improved efficiency and usability in information applications, as the complexity and unreliability of software production was illustrated. Fast forward to now and the Software Engineering Institute (SEI) agency in the United States is working to boost the productivity of software engineering programs (WhatIs, 2020).

Figure 1: NATO Software Engineering Conference, 1968

The main reasoning behind the evaluation of information engineering has to do with market strategy. Managers want to be able to measure their software engineers' productivity and performance, but as we learned from Frederick P. Brooks, this is easier said than done, "Software entities are more complicated for their scale than maybe any other human construct since no two pieces are the same" (Brooks, 1987). In software engineering, the evaluation and development of software processes is a critical part of the business, because all these processes, whether technological, administrative, or quality, represent the foundation of software organizations. Based on their task orientation and process correlation to a reference model, the sophistication of these

software organizations can be calculated (Grambow, et al., 2012). A manager needs enough measurable data for this purpose in order to be able to assess the output of his employee and thereby become more efficient.

Measurable Data

In every sector, data is an integral aspect of quality improvement. Throughout the software engineering process, large volumes of data can be gathered, but the task is to ensure that this information can generate valuable information with clear business priorities in mind for process optimization.

Software engineering measurement is different than most other work disciplines. It is difficult to calculate efficiency or worth solely by the number of units produced or hours employed. In software engineering, software metrics are classified as tools and analysis used to assess performance. The word refers to numbers that describe software code properties, as well as models that help forecast the output of software and specifications for services. In enhancing how we track, forecast and optimize different qualities of the software engineering process, software metrics play a key role. To try to understand the positive and negative sides of each type of measurable data, I will evaluate a variety of approaches.

1. Lines of Code/Source Lines of Code (LOC/SLOC)

The most primitive and simplest measure of progress, SLOC is measured by the amount of lines of code written. There are 2 main measures of SLOC, physical and logical. Physical measures all lines of code written (including comments), and logical measures the number of statements. SLOC is a rough tool used to measure output, and in no way indicates the quality of code. Over-emphasis on SLOC, especially physical SLOC, can encourage software engineers to write unnecessarily long code, which is the opposite of what is desired, thus reducing the comprehensibility of the code and reducing the programmer's efficiency. Bill Gates famously compared the use of SLOC as a measure of progress to measuring the progress of plane construction by how heavy the plane is. Emanuel Weiss (2002) (Weiss, 2002/03) noted several of the flaws of SLOC outlined above, and we have seen a decline in the use of SLOC in the years since then.

2. Number of commits

Similar to SLOC, this method counts the number of commits by the developer over time. It may be used to measure the consistency of development, or to visualize code churn as seen in figure 2, but it bears no indication of the quality of the code.

3. Code Churn

Code churn measures the rate that code evolves at. There are multiple methods of measuring code churn, such as the number of commits over time, or the number of lines added & removed. It can be visualized as seen in figure 2, and is useful a useful metric for managers who want to track the rate of change of code overtime. If the SLOC removed is greater than SLOC added but functionality remains the same, it may

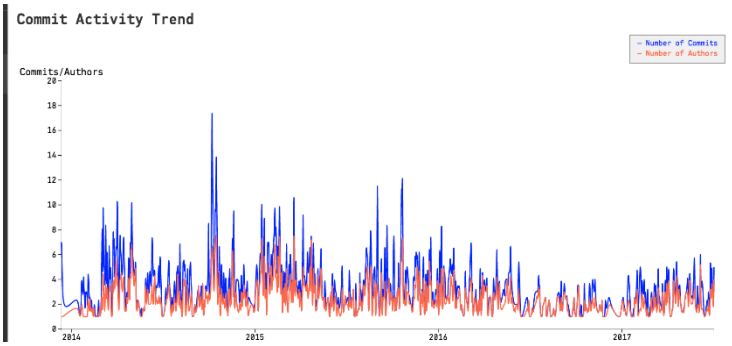


Figure 2: Code churn visualized using number of commits over time.

Source: https://codescene.io/docs/_images/CodeChurnPulse.png

indicate increasing efficiency of the code. Like SLOC, it doesn't indicate the quality of the code.

4. Code Coverage

Code coverage is used exclusively for tests, and is a percentage measure of how much of the original code is executed by the test file, i.e. if each statement/Boolean scenario is tested. It is a poor measure of quality because flawed tests contribute equally to code coverage as effective tests, but is a useful metric for measuring if enough tests have been written.

5. Observation/Keystroke tracking

The question of whether there is a connection between the speed of work of a person and the reliability of their code emerges from this insightful metric. The actions of a person can be monitored by tracking each activity they conduct on their computer down to each key, with this data gathered to evaluate the above query. I have read anecdotes online of peers who went in the programming process at a noticeably slow speed but created reliably high-quality, bug-free code. My experience would imply that the worth of one's work could outweigh that of a less skilled creator at any point in time, although there might be exceptions to this of course.

6. Self Assessment

Employees assess their own performance and identify their own "key performance indicators" is an increasingly prevalent practice, also adopted in some part by Google. There is an apparent drawback of this calculation in that it is arbitrary but can also be seen as an indirect indicator of employee happiness.

7. Technical Debt

Technical debt is essentially the opportunity cost of solving a problem. The extra effort that it takes to add new features or fix the problem is the technical debt. There will always be some element of technical debt in software engineering projects, but it should be kept to a minimum in order to maximise productivity. (Fowler, 2019)

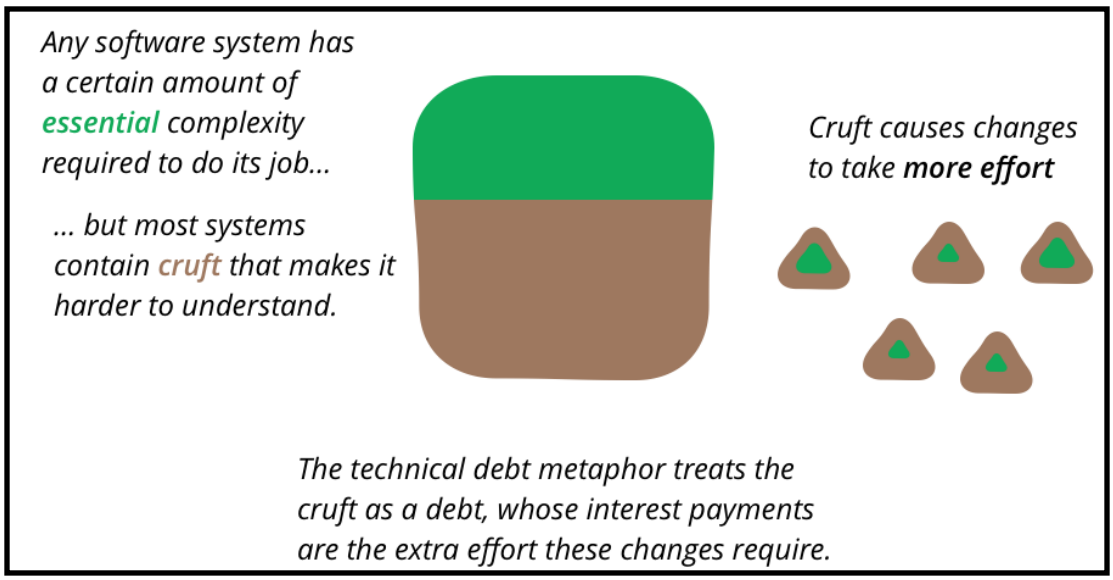


Figure 3: Simple graphical representation of technical debt.
Source: https://martinfowler.com/bliki/TechnicalDebt.html

8. Agile Metrics

There are dozens of agile metrics, and these can be highly accurate and are used extensively in software development. Different agile metrics require different data inputs. All require tasks to be predetermined, roughly of equal size, and tracked. Their ability to be displayed graphically allows developers to see their productivity in real time. Below is a table outlining different agile metrics, and what they measure.

Agile Metric	Description	Total tasks remaining	Tasks completed	Task completion length	Individual task importance
Sprint Burndown Report	Measures tasks remaining	Y	N	N	N
Velocity	Measures speed, quantity & importance of task completion	N	Y	Y	Y
Epic and Release Burndown	Measures tasks remaining and displays graphically	Y	N	N	N
Control Chart	Measures the length of time to complete tasks	N	N	Y	N
Cumulative Flow Diagram	Measures work completion	N	Y	N	N
Lead Time	Measures total length of delivery time for products	N	N	Y	N
Value Delivered	Measures quantity & importance of tasks done.	N	Y	N	Y
Work Item Age	Lagging indicator	N	N	Y	N
Throughput	Measures speed of task completion for individual or team.	N	Y	Y	N

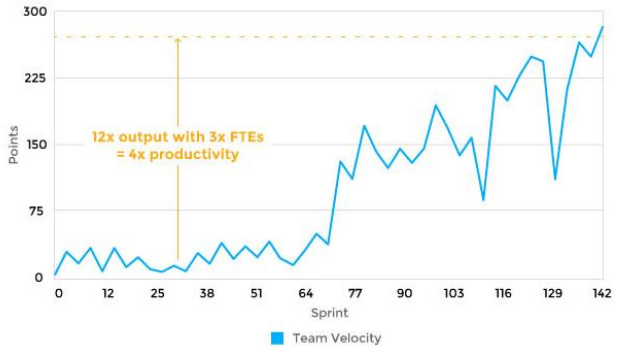


Figure 5: Velocity chart

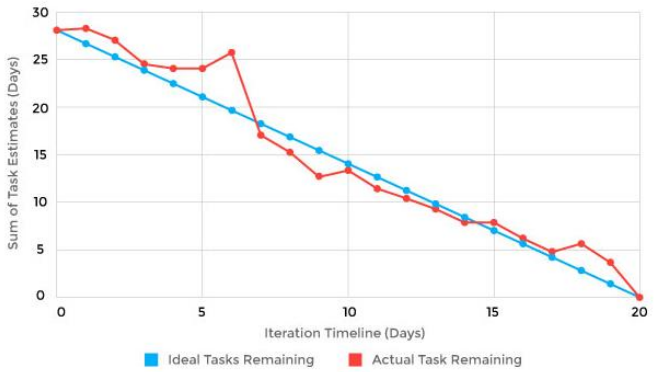


Figure 4: Sprint Burndown Chart

9. Bugs fixed

Management need not really be involved in how many bugs exist in the code of the creator, but rather in the magnitude and how they treat these bugs. The time spent on bug fixes may be used as a measure, but the time spent would vary from developer to developer regarding bug severity/quantity.

Limitations of software engineering measurement

For managers, it may be very easy to glance at a metric and leap to a conclusion. We may not use metrics to infer precise reasons, however. Until determining if the metric is a cause for concern, contact with the team and inquiry into any troubling metrics is important, and metrics need to be considered in background. When we look at the total number of defects and the density of defects, which is the number of defects per new or changed line of code, an example of this is. Intuitively, more errors are predicted to have longer coding. Our priority is the number of post-delivery errors in our applications, but hypothesis tests conducted on the metrics detailed in academia show that complexity indices are weak predictors of post-release fault density. (Norman E. Fenton, 1999)

Existing Solutions

Tools are a critical part of the software engineering measurement process. Once management has selected the metrics, they can use to gather the relevant information, they must then determine where to measure it in order to achieve the data. In recent years, the field of computing observable data has undergone significant advances, with several businesses providing methods to calculate and analyze machine metrics.

i. Personal Software Process (PSP)

The Personal Software Process (PSP) is a software development process that has been developed to direct engineers to consider their own results, thus reducing software errors and increasing the accuracy of their development time estimates. The concept behind it is that developers should try to write high quality code from the start instead of depending on evaluations.

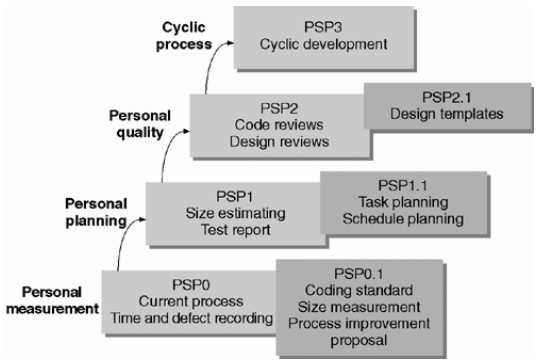


Figure 6: The PSP represented graphically

The PSP definition was first proposed by Watts Humphrey in the 1990's (Humphrey, 2000). He assumed

that the engineer in question would be able to streamline their operation in a more productive way through a systematic implementation process as well as disciplined success monitoring.

The PSP recognizes that every software engineer is unique and must thus evaluate their own performance in their own way. It would not work if there was a universal format, since developers need to personalize and plan their own evaluation. They could deliver higher quality finished goods if engineers are responsible for their own failures. The PSP also stresses the importance of discovering problems sooner, making correcting them easier.

It starts with project preparation by developers drafting a concept paper, and this design must be reviewed before going forward. The next step is the coding, which must also be reviewed before the testing is carried out. Finally, to explain how the project went, a review must be performed, meaning the creator must continually keep track of their own results, even items like missing deadlines or code defects. Items such as lines of code or tasks completed must also be used to document the scale of the project.

In a system similar to Maslow's Hierarchy of Needs (Maslow, 1943), if a certain degree of utility or in this context, improved output has been reached, engineers must aim to develop their processes again. A strong disadvantage of the PSP is that it is vulnerable to human error, and the creation of the Leap toolkit was prompted by this inefficiency. (Johnson, 2013)

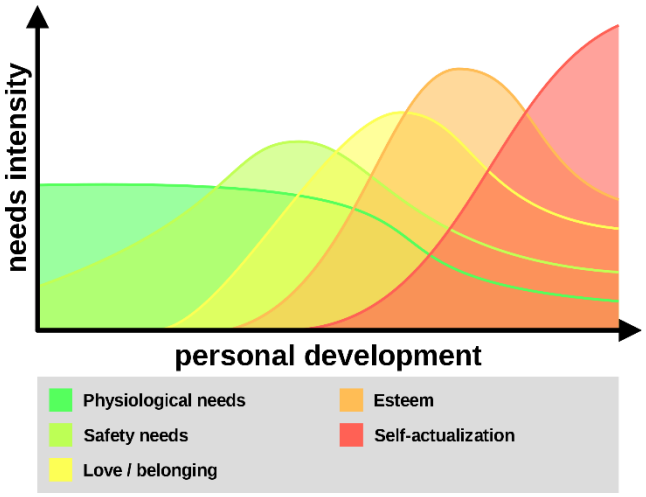


Figure 7: How an individual's hierarchy of needs change as they develop

ii. Leap Toolkit

Through automating and normalizing data processing, the Leap toolkit aims to resolve the issues contained in the PSP. While manual developer feedback is still needed for the toolkit, it then automates subsequent PSP analysis and offers additional resources, such as regression analysis. To use a metaphor: "The Leap toolkit substitutes a campfire for the PSP candle." The storability of the Leap Toolkit means developers can carry it across projects and organizations. Since the development of the Leap toolkit and its industry-wide continuity in use, it was agreed that user feedback would be needed in some way and it was not feasible to completely eliminate human error. A way around manual feedback was needed to carry out impartial analysis, which inspired the development of more digital methods, including a decade-long research project named Hackystat. (Johnson, 2013)

iii. Hackystat

Hackystat is an open source framework for collection, analysis, visualisation and interpretation of the software engineering process. It is possible for scholars, developers and more to use Hackystat. The platform reverses the traditional method of identifying high-level priorities and thereby deciding what interpretation of data collection is necessary to accomplish them.

Hackystat enables users to connect 'sensor' software to their development tools that unobtrusively capture and transmit 'raw' development information to a web service called the Hackystat SensorBase which can be alter queried by other web services to form higher level abstractions of this raw data (Hackystat, 2018). The objective of Hackystat is to facilitate ‘collective intelligence’ in software development, giving us an insight and knowledge about the raw data it collects.

However, this may lead to disputes with developers who may be uncomfortable with the concept of their data being constantly measured, and their productivity constantly measured and scrutinized (Johnson, 2013). Since they can conveniently combine the repositories with Hackystat and obtain useful knowledge, it is ideal for any enterprise using GitHub or another git derivative.

iv. Code Climate

Established in 2011, Code Environment is a privately owned tech corporation. Code Climate is an automatic code analysis mechanism that before recommending what should be changed in the software, analyzes code at many different stages. Every day, Code Climate analyzes massive volumes of code, which is why it is the most popular platform for developers currently available. It provides a service in collaboration with Github, which shows a customer their code coverage, technological debt and progress report. It enables various languages to be automatically evaluated, including Python, PHP and JavaScript. In addition, it is known for its reliability, consistency and test coverage, while also allowing team sharing, which ensures that it can provide a whole production team with direct access. However, when compared to its competitors, Code Climate is quite costly.

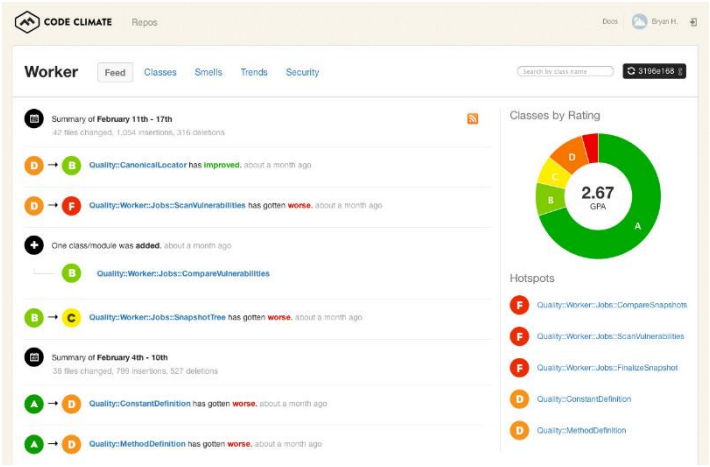


Figure 8: Code Climate dashboard

v. Humanyze

If it is not enough to calculate the software engineering process alone, Humanyze could act as an alternative for businesses. Humanyze offers sensors for businesses to watch all their workers do all day long. Their credit card-sized badges feature a voice and conversation monitoring microphone, a motion detection accelerometer, infrared to identify who one is talking to, and Bluetooth to monitor location. (Heath, 2016)

Humanyze has now developed itself, partnering with over 50 Fortune 500 firms across a wide variety of pharma, oil and technology sectors. Employees may feel uncomfortable about the amount of data gathered, but Humanyze emphasizes that the system is designed to anonymize details, measuring the efficiency of the entire enterprise rather than individual employees.

vi. Codebeat

Thanks to the fact that it is easy, free, and open source, Codebeat is another commonly used tool for measuring software development. In recent years, it has evolved dramatically and supports numerous programming languages, including Python, Ruby, Java and more. The features of Codebeat have recently advanced, but it is still most

common for user input to be taken into account. Codebeat develops its own algorithms from scratch, with knowledge that allows developers to concentrate on core areas of code analysis, such as design and business logic. Despite this, there are still some missing features in Codebeat, namely the lack of security checks, and no support for open source tools or CSS (Ozimek, 2017).

vii. Other Platforms

For executives to evaluate digital innovation data and metrics, several more tools exist, such as GitPrime, Waydev, Codacy, and Velocity, etc. The vast variety of available platforms is representative of the ever-increasing prevalence and sheer value of measuring software engineering.

Figure 9: Logos for other platforms

Algorithmic Analysis

Data analytics is obviously an integral aspect of evaluating the process of software engineering. Manual data processing is increasingly being replaced by algorithms that automate the process, and the aforementioned computing platforms have a variety of algorithms running in the background. In a short time frame, algorithms can generate a vast volume of information, as well as recognize related systems, patterns and trends. Big software firms such as Google have patented algorithms used to measure the success of their workers. I will focus my work on this report on the underlying technology and methods behind the software.

Machine learning is “a method of data analysis that automates analytical model building. It is a branch of artificial intelligence based on the idea that systems can learn from data, identify patterns and make decisions with minimal human intervention” (SAS, 2020). All machine learning (ML) fall into one of 3 categories (Brownlee, 2020).

1. Supervised Learning
2. Unsupervised Learning
3. Semi-Supervised Learning

Supervised Learning

This type of ML algorithm produces a function that maps inputs to ideal outputs. As the data scientist serves as a guide to show the algorithm what inference it can draw from the data, it is "supervised". A "training dataset" where there are a variety of inputs and corresponding outputs is used to "teach" the results. Iteratively, the algorithm makes predictions based on the knowledge and corrects any outlying predictions. The procedure is then repeated until it achieves enough precision.

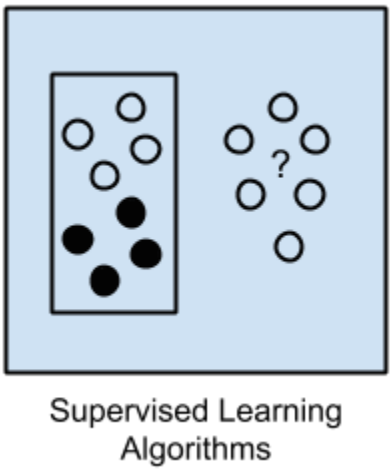


Figure 10: Graphical representation of supervised ML (Brownlee, 2020).

The following are the forms of supervised learning algorithms:

A. Linear Discriminate Analysis

Linear discriminant analysis is a classification technique that allows us to place multivariate data into classification groups. A classification judgment bound by this algorithm identifies and classifies several instances of this data by the boundary. Within the ML and AI profession, the procedure is widely used (White, 2020).

B. Decision Tree Analysis

An illustration of supervised machine learning is decision tree analysis. It is a graphical representation of different possible solutions to solve a problem that are available. By answering a series of questions after each other, a final response is obtained by providing positive or negative responses (Mulder, 2017).

C. K-Nearest Neighbour

K-Nearest Neighbours is a non-parametric method of grouping. Within each class, the algorithm makes no assumptions about the spread of results. In order to make a guess, the algorithm is applied by obtaining a point and classifying it using the data we already have. It determines the number of K-Nearest Neighbours to help identify the point, e.g. k = 2, and then the computer chooses the vector that dominates the choice of the 2 closest observations to the new point (White, 2020).

Unsupervised Learning

Machine learning without supervision is similar to supervised machine learning, except that you only have input data and no associated output variable in this situation. Some believe unsupervised learning to be more synonymous with machine learning, since the goal is to make the machine learn how to do things without asking us how. Without human input, the aim is for the algorithm to learn to recognize complicated processes and patterns. Input information is supplied, but the output is uncertain. The following are the forms of unsupervised learning algorithms:

A. Principal Component Analysis

Principal Component Analysis (PCA) is a data reduction technique that highlights variance and highlights otherwise unclear patterns in a dataset. It is also used to make it easy to investigate and visualize data. It does this by converting a set of correlated variables into a reduced set of uncorrelated variables, by altering the eigen vector of the plane, thus transforming the xy plane in a way that highlights the differences in data.

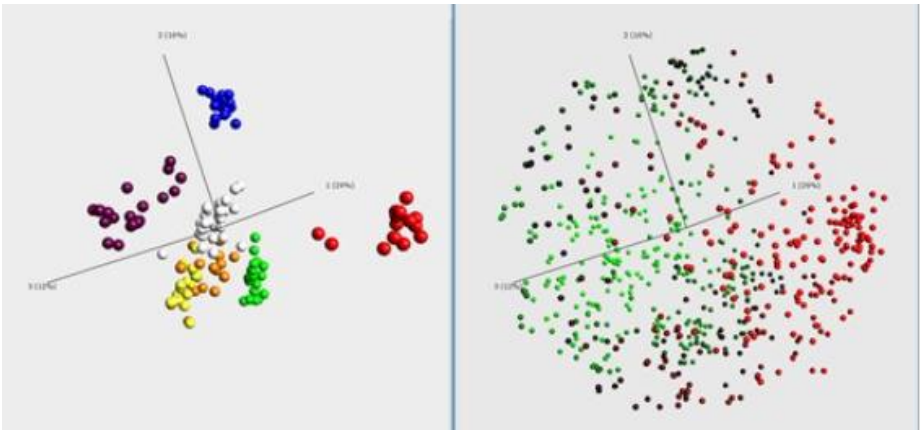


Figure 11: Visual representation of PCA in action. Notice how much clearer the difference is between the groups of data in the model on the left, which has had PCA applied to it.

B. K-Means Clustering

In data analytics, clustering algorithms are very useful since any dataset can be grouped according to its various features into a series of clusters, thereby defining a category structure in the dataset. We can, for instance, analyze the characteristics of such clusters if there are distinct bands of fast-working and slow-working engineers. This algorithm is an iterative approach that separates the data into various groups such that observations are in the same group with similar characteristics, while observations are different within groups (White, 2020). Mean-shift clustering and hierarchical clustering are other ways of clustering.

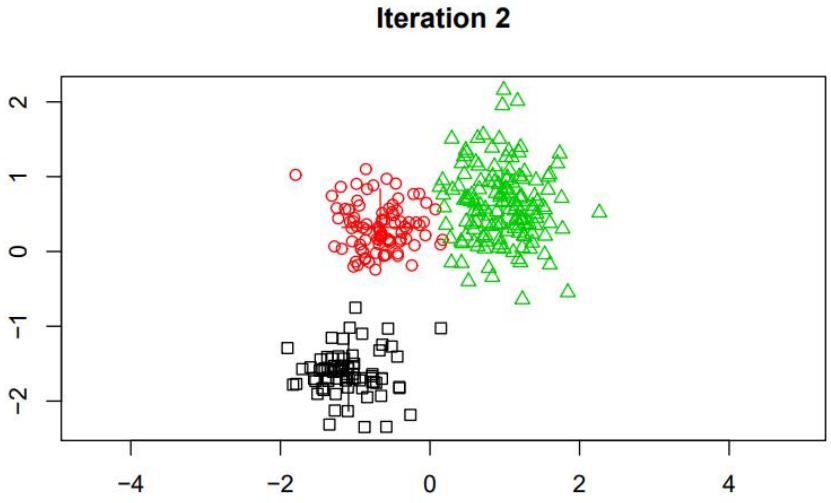


Figure 12: k-means clustering where k = 3 (White, 2020).

Semi-Supervised Learning

Semi-supervised learning is a machine learning approach that during testing, mixes a small amount of labeled data with a significant amount of unlabeled data. Semi-supervised learning falls somewhere between unsupervised learning and supervised learning.

When used in combination with a limited volume of labeled data, unlabeled data can yield significantly more accurate results than simply using an unsupervised algorithm with only unlabeled data. This is because the algorithm can learn and extrapolate its results from the labeled data to infinite amounts of unlabeled data. Therefore, the costs associated with the labeling process will make massive, completely labeled training sets unfeasible, whereas it is relatively inexpensive to procure unlabeled results. Semi-supervised learning can be of considerable functional benefit in such cases. In machine learning and as a paradigm for human learning, semi-supervised learning is also of interest.

The following are examples of semi-supervised algorithms:

A. Semi-Supervised Clustering

"Semi-supervised clustering uses a small amount of labeled data to aid and bias the clustering of unlabeled data" (Basu, et al., 2002)

There are multiple approaches to semi-supervised clustering. In the seeded approach, the labeled data helps initialize clusters in a process known as 'seeding'. In the constrained approach, the labeled data constrains groups, which the unlabeled data then fills. In a feedback-based approach, regular clustering is first run, followed by an adjustment of the clusters based on labeled data. This benefits from user feedback and can be adjusted iteratively.

B. Expectation-Maximization (EM)

The EM algorithm is generally considered an unsupervised ML algorithm, however it has applications in a semi-supervised setting (He & Jiang, 2020). It works by first receiving some input data, aka labeled data. It uses this to calculate the possibilities of all of the possible answers for the variable, and uses the expectation with the highest probability, thus maximizing the expectation as the name implies. One of the strengths of EM is its ability to deal with unidentified or latent variables. This allows it to deal with large amounts of unlabeled data and is frequently used in Natural Language Processing (NLP) among other fields for this reason. For example, by labeling a small pool of documents, a semi-supervised EM algorithm can 'read' virtually limitless amounts of unlabeled documents and classify the data with precision close to that of human cognition (Nigam, et al., 2000).

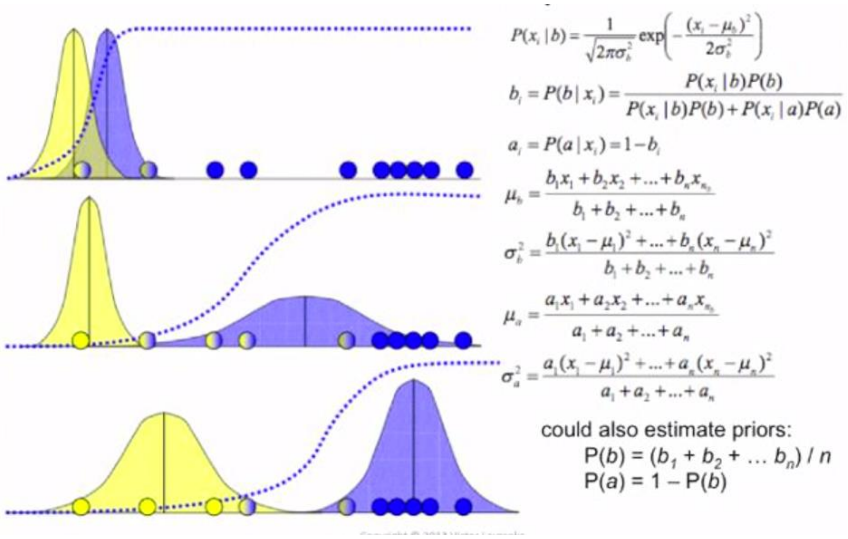


Figure 13: 2D graphical representation of EM algorithm, accompanied by relevant mathematic formulas

Ethics

The power of data analysis has grown exponentially since the invention of the computer. This has brought with it many great benefits to humankind and revolutionized our world. It has also brought an immense responsibility upon data analysts to consider the ethics of their actions, as their actions may have immense consequences on humans all over the globe. A recent, well-documented case of data analysis software created without enough consideration for ethics was the Facebook – Cambridge Analytica 2016 scandal, where the data of millions of Facebook users was harvested without their consent by Cambridge Analytica, who under the patronage of Ted Cruz and later Donald Trump, targeted individual swing state voters with highly personalized targeted online advertising, swaying the outcome in the neck-and-neck 2016 U.S. Presidential Election. Such interference is in an ethically dubious area, and warrants further discussion and research before such a tool is deployed on the population again (Boldyreva, et al., 2018). I will explore the ethics of measuring software engineering under 2 headings:

1. Data Collection
2. Data Rights

1. Data Collection

The way information is collected can pose a range of ethical questions, especially when information is gathered unobtrusively. In the case of Hackystat, if data is being gathered without the developer's explicit permission, developers may feel uncomfortable. As discussed earlier, Humanyze technology also monitors the single step and contact of workers, processing it in real time. This will impair competitiveness and morale, and therefore job efficiency, because developers are not going to work openly. This belief is supported by academic literature, which shows that increased anxiety causes employees

to choose less creative solutions and more unethical (McCarthy & Cheng, 2016). A famous example of competitiveness eliciting unethical behavior from staff was the Wells Fargo account fraud scandal, when employees created millions of fraudulent savings and checking accounts on behalf of Wells Fargo clients without their consent, which were opened by employees trying to hit their sales goals. This same research also shows that if competition elicits excitement instead of fear, the opposite occurs, and employees are more creative and less unethical. This displays the importance of creating a work environment where employees feel safe, secure, included, and excited to work.