CSU33012 Software Engineering

Measuring Software Engineering

Keira Gatt (19334557)

Table of Contents

1 Metrics

1.1 Lines of Code

The Lines of code (LOC) metric is basically an indication of size, which allows for the estimation of other variables such as effort, time scale, and total number of faults for the application. The LOC metric can be derived either with Physical Lines Of Code (PLOC) or Logical Lines Of Code (LLOC) where the difference between the two is best illustrated with an example –

Code Segment	Line #	Code
A	1	If(a == 1) b = 2;

B	1
	2
	3
		If(a == 1) {
		b = 2;
		}

Although these two snippets of C code are identical in terms of logic and functionality, they produce the same LLOC but different PLOC measurements. This is because unlike PLOC which measures the amount of lines of code within a program, LLOC measures the number of executable statements. A variation for both PLOC and LLOC is when they are combined with NLOC, which only considers the non-commented lines of code.

This metric is intuitive and easily automated but is not very useful for measuring an engineer's productivity and does not provide an insight of the engineer's skills. In fact, given that it is based on line counting, LOC encourages programmers to write overly-verbose, monolithic and repetitive code; an incentive that works against productivity and in particular when it comes to code reuse, maintenance and scalability. In addition, LOC counting is not considered a reliable method to determine functionality and complexity and is not adequate enough to address visual and non-procedural languages.

1.2 Code Churn

Code Churn is a natural part of a project cycle. Code churn occurs when a developer edits a piece of existing code. It shows where developers have been spending most of their time in the code base. It is normal for a programmer to go back to a piece of code to refine and test it. But a high level of code churn may indicate some deeper issues in the project such as a high risk code segment. It could be high risk for a number of reasons, e.g. lack of skills or knowledge in the team, writing code before understanding the aim and scope of the project or bad test coverage and practices. Code churn is a good metric to use to decide which piece of code should be investigated and fixed first.

1.3 Code Coverage

Code Coverage tools determine how much of the code is executed with your test suite. Code coverage can be broken down into the following categories:

i) Function coverage

Counts the number of functions called by the test suite.

ii) Statement coverage

Measures the percentage of code statements executed during the test run.

iii) Branches coverage

Determines how many branch decisions of control structures have been covered by the test suite.

iv) Condition coverage

Measures how many permutations of the boolean sub-expressions have been tested for.

Each of these sub-categories is usually represented as a percentage of *number of items tested / number of items found*.

Good code coverage helps avoid undetected bugs. But, high code coverage doesn't necessarily mean the code is robust enough for deployment. 100% coverage does not account for situations a user may come across that have not been tested for in the test suites. Aiming for full code coverage could be the downfall of a project; it could motivate the team to write tests that achieve 100% rather than tests that are based on the application requirements and users' perspective. Coverage can only tell the team so much; it cannot indicate if there is crucial functionality missing in the source code. Although achieving high code coverage is the basic idea; it must be paired with robust test suits that have the users' experience and system integrity in mind for it to be considered a useful metric.

1.4 Agile Velocity

Agile velocity measures the amount of work a single team completes during a software development iteration or Sprint. It is usually measured in Story Points and can be used to forecast how quickly the team can complete a task. The metric becomes more accurate over time; as it is measured per iteration. It is convenient to have a measure that can predict how quickly a team can get through a backlog but it must be used with caution.

Story points are subjective; they are agreed upon by an individual team. Therefore, comparing velocity across teams is a bad idea. One cannot determine if team A is more productive than team B solely on whether they have accumulated more Story points. Velocity is a measure unique to each individual team. Also, Story Points can be toyed with to give the illusion that a team is more productive. With this,

developers have the incentive to inflate their points and their full attention is no longer on the quality of the code they are writing.

Using velocity to estimate the delivery of a long-term project is another bad idea. A project changes too often and has too many complexities to be estimated with previous Sprints.

1.5 Mean Time Between Failures (MTBF)

This metric is typically used in conjunction with MTTR (Mean Time To Repair), MTTF (Mean Time To Failure) and MTTA (Mean Time To Acknowledge) to understand how often unexpected incidents and failures occur.

MTBF is the average time between repairable failures in a system. It is the total uptime of the system divided by number of failures. The metric is used to anticipate how reliable the system is and can be used to plan a schedule for preventative maintenance.

As with Agile Velocity, MTBF data can be skewed by selecting a favourable selection of time periods. For example, selecting a time period just after a failure and ending just before another failure. On a par with Velocity again, MTBF becomes more accurate over time; the more time periods there are to calculate for, the better the representation of the system's reliability.

1.6 Commit Count

Commit Count is a relatively easy metric to extract and calculate and is used by GitHub to record every commit made by a developer. Generally, it is a useful measure that indicates if a programmer is being productive; if a programmer has not made any commits in a couple of days, it is often a sign that he/she is faced with some difficulty.

But Commit Count must be used with caution; it is very susceptible to manipulation and abuse. This is because, commit counts are used as a "save point" by developers. Therefore, if developer A commits after they write a single line of code and developer B commits only once they've developed some new functionality or solved an issue; developer A will appear to be the more productive employee. From a developer's perspective it would seem like a good idea to save after every line written, even if it was not a useful line of code, to look like a valuable employee. This kind of metric encourages counterproductive behaviours and does not effectively assess an engineer's productivity.

2 Platforms

2.1 SeaLights

SeaLights offers products to gather and analyse all test code and coverage in a code base, regardless of testing frameworks and tools. Their Software Quality Intelligence Platform (SQIP) assesses the risks associated with each new piece of code and governs whether the code is sufficiently tested to be deployed into the organisation's pipeline.

SQIP measures tests coverage across every test stage and analyses the types of tests being used such as manual and automated testing. Since high test coverage does not necessarily mean that the code is error-free and that all crucial components are sufficiently tested, SQIP analyses all branches, paths and lines that are covered in the tests and provide information on any missing tests.

SQIP also aims to measure the functionality and attempts to determine the risk associated with a code segment. If it is low risk, it does not engage in the same level of testing as for areas of code that are considered crucial. This saves testing time and improves the team's productivity.

The platform can also evaluate the quality risks of new code by allowing or rejecting a code merge, based on the organisation's risk policies. This is a very important feature as it prevents non-compliant code from entering the code base pipeline.

2.2 Humanyze

Made by MIT Media Lab, Humanyze analyses existing corporate data from enterprise applications, communication tools, and location systems to improve the organisation's productivity and work environment.

The sociometric badge, one of Humanyze's products, analyses employees' conversations based on collected data that includes the parties that employees talk to, the duration of conversations, the employees' stress level based on heart rate and the voice inflection. This data is sent off to Humanyze, where it is visualised to showcase the social interactions in the workplace. Privacy issues are of concern, but Humanyze says that wearing the badges are at the employees' discretion and all data is anonymised, so that employers can only see a web of social interactions and not individual employee information.

Humanyze had an interesting case study of a large European Bank that wanted to understand why two very similar branches differed by almost 300% in their performance. From analysing the movements, work schedules and interactions of the employees in both branches, Humanyze found that the underperforming branch had less time to collaborate together and had a poor office design that restricted communication between teams. With this new information, the bank redesigned the underperforming branch to allow for more social interactions and in return, increased sales by 11%.

Although the data Humanyze collects seems intrusive, the badges are optional and the data is anonymous, and as can be seen from the case study, using the measured data improves company productivity and workplace relations.

2.3 Pluralsight

Pluralsight's product, Flow, makes use of Git repository data to improve team performance. It offers a number of services that include -

i) Get insights of the team's familiarity with programming languages

Determine the number of commits made per language, gauge the proficiency of a team based on language competency tests and get workflow recommendations.

ii) Evaluate the success of releases and compare sprints

Assess the team's most productive time periods and use the data to improve future sprints and avoid past bottlenecks.

iii) Assess the team's code review dynamics

Determine if the code reviews are useful, easily find non-reviewed pull requests and assess the extent of collaboration between the team, based on discussion in the comments.

iv) Increase knowledge distribution

Establish if there are any lead engineers not sharing their expertise with the team and make improvements in a way that expertise is distributed across the team.

2.4 CAST

The CAST Application Intelligence Platform (AIP) is an enterprise-grade software measurement and quality analysis platform used to assess applications both for code quality metrics and security vulnerabilities. It checks for compliance with architectural and coding standards by identifying shortcomings in a number of areas including performance, programming practices, architectural design, security and robustness.

AIP supports most source code languages, including scripting and interface languages, 3GLs, 4GLs, Web and mainframe technologies, across all layers of an application (UI, logic and data). Code and structural analysis showing accurate and timely metrics, is provided by means of dashboards and drill-down facilities allow managers to access and share multiple levels of information.

Among the tangibles benefits of using AIP, CAST claims that their product delivers –

i) 10 - 50% reduction in source code defects early in the development cycle

ii) 10 - 20% efficiency gains by reusing components and frameworks and thus reducing developer rework

iii) 10 – 30% cut down in maintenance costs over time as a result of continuous technical quality improvements

3 Analytics

3.1 Code Complexity

3.1.1 Cyclomatic Complexity

Cyclomatic complexity is used to determine the structural complexity of a code module in order to limit its complexity, which can lead to a high number of defects and maintenance costs. It is also used to identify the minimum number of test paths to assure test coverage. The metric is essentially a measure of the number of control flows within a module, an approach based on the premise that the greater the number of paths through a module, the higher the complexity.

Cyclomatic complexity uses the cyclomatic number graph theory, which when applied to software measurement, indicates the number of linearly independent paths within a program where the program is represented as a strongly connected graph with unique entry and exit points. The cyclomatic number ***V(g)*** can be calculated either by counting the nodes and edges of the graph or by counting the number of binary decision points as outlined below -

i) $V(g) = e - n + 2$

where ***g*** is the control graph of the module, ***e*** is the number of edges, and ***n*** is the number of nodes.

ii) $V(g) = bd + 1$

where ***bd*** is the number of binary decisions in the control graph. Note that in the case of a *n-way case* or *select* statement, this is counted as *n - 1* binary decisions.

The main advantages and disadvantages of using the cyclomatic complexity metric are as follows –

Advantages	Disadvantages
The metric is easy to calculate, even when considering the complexity of several graphs that are treated as a group, which would be equal to the sum of the individual graphs' complexities	High complexity does not always mean the code has to be revised as some problems are complex by nature
A high complexity value can indicate that the code is overly complex	The metric is only useful for the control flow of a program and does not account for the complexity of sequential statements and data
The program graph shows the most complex code region where one should concentrate most of the testing efforts.	The metric does not distinguish between nested and non-nested loops and different kinds of control flow complexity such as loops versus IF-THEN-ELSE statements

3.1.2 Size

The size of source code is normally measured in Lines of Code (LOC) or Function Points (which are also converted to LOC after being factored with language gearing). The usage of LOC as a software measurement metric has already been discussed in *Section 1.1* so the interest in LOC here is its relationship with the defect density quality metric, i.e. the ratio between the total number of defects and the size of the module code.

Based on research by *Les Hatton*, *Yashwant K. Malaiya* and *Jason Denton*, the number of defects is based on the number of modules and the number of lines of code. For each module, there are faults caused by having a module rather than by having the code included in another module, such as initialisation defects and handling of global data. Therefore, for module-related faults, if a module is of size ***s***, its expected defect density $\boldsymbol{D_m}$ (in defects/LOC) is given by –

$D_m(s) = a / s$

where ***s*** must be greater than zero and ***a*** is an empirically derived constant. In terms of defect density, this factor decreases as module size grows.

Hatton, Malaiya and *Denton*, when correlating published defect density with module size, concluded that for very small modules, the defect density decreases linearly as the module size increases, until it hits a point somewhere between 200 and 400 LOC. At this point, it flattens out and then starts to increase linearly. They also arrived at another important conclusion when they discovered that language is not a large factor and that the fault density behaviour is consistent across languages.

3.1.3 Halstead's Metrics

Halstead makes a distinction between software science and computer science based on the premise of software science that any program is made up of tokens, which could either be operands or operators. Operands are tokens with a value, such as variables and constants whereas operators constitute everything else, including commas, parentheses, and arithmetic operators. The primitive measures of Halstead's software science are -

$\boldsymbol{n_1}$ - number of distinct operators in a program

$\boldsymbol{n_2}$ - number of distinct operands in a program

$\boldsymbol{N_1}$ - total number of operator occurrences

$\boldsymbol{N_2}$ - total number of operand occurrences

By using these primitive measures, Halstead developed a system of equations expressing the total vocabulary, the overall program length, the potential minimum volume for an algorithm, the actual volume (number of bits required to specify a program), the program level (a measure of software complexity), program difficulty, and other features such as development effort and the projected number of faults in the software.

The most significant metrics among Halstead's many equations are the following -

i) Vocabulary (*n*)

$n = n_1 + n_2$

ii) Length (*N*)

$N = N_1 + N_2 = n_1 \log_2 n_1 + n_2 \log_2 n_2$

iii) Volume (*V*)

$V = N \log_2(n) = N \log_2(n_1 + n_2)$

iv) Level (*L*)

$L = V^* / V = (2 / n_1) (n_2 / N_2)$

where ***V**** is defined as the minimum volume represented by a built-in function performing the task of the entire program.

v) Difficulty (*D*)

$D = V / V^* = (n_1 / 2) (N_2 / n_2)$

Note that program difficulty ***D*** is the inverse of program level ***L***, which is a measure of program complexity.

vi) Effort (*E*)

$E = V / L = DV$

vii) Faults (*B*)

$B = V / S^*$

where ***S**** is defined as the average number of decisions between errors. According to Halstead, the value of ***S**** is 3000.

Notwithstanding the significant impact of Halstead's work in the field of software measurement, the metrics are generally derived once the code is written and therefore have limited uses in projections and forward planning. For instance, to predict the program length *N*, data on N_1 and N_2 must be available, and by the time these two variables can be determined, the program should be completed or near completion.

3.1.4 Maintainability Index

The Maintainability Index (MI) is a composite metric that aims to determine the effects of code maintenance on code maintainability. It is expressed as a polynomial consisting of *Lines Of Code*, *Cyclomatic Complexity* and *Halstead Volume* and can be evaluated as –

$$MI = 171 - 5.2\ln(aV) - 0.23aV(g') - 16.2\ln(aLOC) + 50\sin(2.4perCM)^{1/2}$$

where :

aV = average Halstead Volume (***V***) per module

aV (g') = average extended Cyclomatic Complexity per module which takes into account the added complexity resulting from compound conditional expressions. A compound conditional expression is defined as an expression composed of multiple conditions separated by a logical operators such as OR or AND. If an OR or AND condition is used in a control construct, the level of complexity is increased by 1

aLOC = average count of lines of code (LOC) per module

perCM = (optional parameter) average percent of lines of comments per module

The following coefficients, which were empirically derived from actual data, classify systems using the MI metric as follows –

MI Score	Maintainability
MI > 85	Highly maintainable
65 < MI ≤ 85	Moderately maintainable
MI ≤ 65	Difficult to maintain

Poor maintainability could be a cause of inefficient, large or uncommented code. Although MI is useful to indicate when code should be revised, it does have some disadvantages. For instance, it does not indicate exactly which piece of code should be improved. Also, a programmer cannot predict how code changes will affect the MI. For example, to improve the Cyclomatic Complexity, one might remove overcomplicated nested loops and conditional statements, but in doing so the LOC will normally go up. As every code change influences all three of the MI's mandatory parameters, it is difficult to determine how codes changes will affect the MI.

3.2 Machine Learning

3.2.1 Supervised Learning

This model of machine learning employs sample inputs consisting of predictor measurements with their associated responses for algorithm training. From these pairs, it creates a model that maps the response to the predictors, which it will then use to predict the response of future observations or to infer the relationship between the response and predictors.

For example, if one wishes to predict a person's height based on their weight and age, they must collect sample data that contains people's weight, age and height. With this data, the algorithm creates its model which, given someone's weight and age can be used to predict their height.

The main advantage of supervised learning is the high degree of control one has of the training process. It is easy to control what the machine learns and what inferences it creates, since the developer has to supply the algorithm with all the sample input and output. But, gathering and labelling all the sample data is expensive and tedious and as the engineer has to supply the data, it is susceptible to errors. Because the algorithm has no way of detecting these errors, it can get confused and distort the predictions of future observations.

There are multiple applications of Supervised Learning such as *Support-Vector Machines*, *Linear Regression, Logistic Regression* and *Decision Trees*, which shall be discussed in more detail in the following section.

Decision Trees

The tree consists of data that is continuously split according to a certain parameter and is represented as decision nodes and leaves, where the nodes are where that data is split and the leaves are the decisions. There are numerous variations on Decision Trees and one of the better known algorithms is the *Iterative Dichotomiser 3 (ID3).*

ID3 uses a top-down greedy search to build trees. Note, a greedy algorithm chooses the option that seems best at that time. During each iteration, ID3 calculates the entropy and information gain for every attribute in the data set. It then chooses the attribute with the lowest entropy or highest information gain and splits the data based on this selected attribute. ID3 repeats this process, only considering the unused attributes.

To split the data set, ID3 uses the following two calculated criteria -

i) Entropy

Entropy, ***H(S)***, is the measure of randomness in a data set. The higher the entropy, the more random the data is and hence, the harder it is to draw conclusions from the data set. Entropy is at a maximum when the probability is 0.5 because it projects perfect randomness in the data.

The formula for entropy is -

$H(S) = \sum_x p(x)\, log_2 \frac{1}{p(x)}$

ID3 uses this attribute to decide whether the data needs to be further sub-categorised, i.e. if the branch does not have an entropy of zero, the data must be split again.

ii) Information Gain

Information Gain, ***IG(S, A)***, for a set ***S***, is the change in entropy after choosing an attribute. The goal of a decision tree is to find the attribute that gives the lowest entropy and the highest information gain. It can be expressed as -

IG(S, A) = H(S) – H(S, A)

Decision trees can handle both numerical and categorical data and do not require data normalisation. They use an open-box model, meaning a result from the tree can be easily explained using Boolean logic. But, they are not very robust; any change in the data set can result in a significant change in the tree structure. The main issue with trees is over-fitting which happens when the tree branches are too specific and do not generalise the data. This issue can generally be avoided by using methods such as pruning.

3.2.2 Unsupervised Learning

Unsupervised Learning algorithms group unsorted data based on similarities and differences without human intervention. They are more complex than the ones used for Supervised Learning as they must identify patterns within the data set themselves. Because of this, unsupervised learning training sets are easy to prepare as there is no need to categorise the data. But they do require large data sets to help them spot patterns and are susceptible to inconsistencies in the data. However, Unsupervised Learning algorithms are particularly useful for detecting previously unidentified patterns.

Clustering is a category of this machine learning approach. With this algorithm, the training data is grouped into categories with similarities. Similarity measures used in clustering include *Euclidean, Cosine*, *Manhattan* and *Jaccard Distance*. Clustering can be further subdivided into five distinct methods, i.e. Partitioning, Density-Based, Distribution Model-Based, Fuzzy and Hierarchical, which is discussed in more detail in the next section.

Hierarchical Clustering

Hierarchical analysis uses a tree approach where the tree representation is known as a *dendrogram*. The dendrogram is a visual representation of the connections between items and is used to divide the final cluster into multiple clusters depending on the problem at hand.

The two types of algorithms used in Hierarchical Clustering are as follows –

i) Agglomerative

This form of clustering uses a bottom-up, greedy strategy. Each data point begins as a cluster, and as one moves up the hierarchy, pairs of clusters are combined. This process is repeated until all data points are in one cluster at which point the dendrogram is then used to sub-divide the cluster.

ii) Divisive

Essentially, divisive clustering uses the opposite approach to agglomerative as it is a top-down procedure. Every data point is lumped together into one cluster and then recursively split, based on small differences between the points, as one goes down the hierarchy. This iterative process continues until the desired number of clusters is obtained.

Hierarchical clustering has many applications such as identifying fake news, tracking sources of viruses and identifying criminal activity. However, it should be noted that it requires significant computational power, which is not ideal for large data sets. It is also sensitive to outlier data which can produce inaccurate results.

3.2.3 Semi-Supervised Learning

In Semi-Supervised Learning we are given a small number of labelled examples and a large number of unlabeled examples. Consequently, this approach falls between supervised learning and unsupervised learning, where neither of the two types of algorithms is effective in the mixed data set. A Semi-Supervised algorithm can be constructed by combining clustering and classification methods.

For example, say we have a large data set of unlabeled images of flowers. We can use a form of Hierarchical Clustering discussed earlier, to find the similarities between the flowers and divide them into an appropriate number of clusters. From these clusters, we choose the picture that represents the cluster the best and label it. Now, with our labelled data set, we can use a classifier algorithm such as the ID3 decision tree to train our machine.

Semi-Supervised algorithms are most useful when there are more unlabeled than labelled data points and are very practical when working with data that is tedious and expensive to label. This often happens when obtaining data is cheap and easy, such as with speech recognition where recording a speech is inexpensive but labelling it requires someone to type a transcript. In many cases semi-supervised algorithms are more beneficial than supervised ones since we can produce a more accurate prediction by using the unlabeled data points. However, it is very important that these unlabeled data points be relevant for the classification problem. If this is not the case, we could actually reduce the accuracy of our predictions due to these incorrect inferences made by the algorithm.

4 Ethics

The software development life cycle (SDLC) is essentially a long chain of human activity, involving a myriad of skilled professionals entrusted with the development of software products that meet a predefined set of requirements. Given that humans are intrinsically wary of any form of monitoring, the introduction of metrics, data collection and analytics as a means to measure the software engineering activity will invariably cause concern among those affected, unless the process is properly planned, communicated and managed. This is a management responsibility that is often overlooked and where the organisation is more focused on the tools and metrics and very little attention is given to the cultural change that software measurements bring to those involved in the SDLC.

I think that the notion of assessing people through metrics is always going to be a controversial topic and does not line up well with good management practices that promote motivated, self-managed and effective teams. There is no denying that the potential for an organisation to use software engineering metrics for purposes other than those intended does exist. And regardless of whether this is supported by corporate policy or carried out by stealth, the introduction of software engineering activity monitoring without proper consultation and the full involvement of the "monitored subjects" can lead to a disincentivised workforce and a system that is ultimately counterproductive.

Unlike many traditional disciplines, software engineering is largely a creative endeavour where engineers are tasked to solve problems through the implementation of concepts and formal methods. Typically, the engineer's cognitive process occupies much of his / her time (which is not always aligned with office hours), improvising and testing ideas before any productive code is actually written. More often than not, solutions need innovation and innovation requires approaching problems from different angles, which is all part of the learning curve involved in the software development process. I am not aware of any methodology that can reliably gauge this particular aspect of the software engineering activity and for this reason, I believe that evaluating an engineer's performance on the basis of a daily-measured quantitative metric such as Lines of Code or Function Points demonstrates a poor understanding of the software engineering profession.

On the other hand, one can also recognise the need for a systematic approach to software development projects and the importance of data-driven methods for decision making, planning, costing, scheduling, tracking and overall project control. At the technical level, software measurements assist in determining the quantitative and qualitative attributes of the software product including scale, complexity and reliability and can serve as the basis for benchmarking and for establishing baselines. Measured data can also play a role in post-project analysis where forecasts are compared with actuals, gaps are identified and recommendations are made towards process and technology improvements.

Therefore, when taking into account the sensitivity and the benefits of introducing a software measurement infrastructure, it is crucial for management to remain pragmatic and to aim at striking a balance between the corporate needs and the wellbeing of the employees. The path chosen by management to implement such a system can go a long way in assuaging the real and perceived fears of the workforce; something that can be readily achieved with open and honest communications and by allowing the stakeholders to actively participate in the process. Other considerations that should be factored in for a smoother on-boarding process are the following -

i) The entire software measurement programme must comply with information management policies and any statutory legal and regulatory requirements, including the *Freedom of Information Act 2014* and the *General Data Protection Regulation 2018*.

ii) Metrics should be goal-oriented and realistic, taking into account the limitations of the organisation such as availability of resources, skill sets, training, etc. and also the nature of the project in terms of scale, complexity, programming language, platform, IDE, etc.

iii) Each chosen metric should have a precise interpretation. Metrics that yield inconsistent or ambiguous values can be misconstrued and lead to a poor understanding of the software engineering activity and the process deliverables.

iv) When establishing metrics, one needs to be mindful of the fact that quality shortcomings such as incompleteness, defects, etc. are propagated across the entire development process. The emphasis here is that a lack of quality control at one stage of the SDLC could have a negative impact on the performance (and therefore the metrics) of successive stages.

v) The scope, volume and frequency of data collection need to be judicious and should be limited to the extent required by the software measurement infrastructure in order to perform its functions. As far as possible, data should be anonymised either directly by the source or through obfuscation and should not be used for any other purpose except as delineated by the requirements of the software measurement activity. All monitoring data must be protected when in-flight and at-rest and access / authorisation controls need to be strictly enforced. The online and offline data storage facilities must be properly managed and all data should be maintained in accordance to an established data archiving and retention policy.

vi) In the event that the collected data carries labels or signatures that could possibly identify the human sources, there must be a formal process that allows an employee to view and examine the data for correctness and completeness. Ideally, the monitored subject should be sent a copy of the data set (in full or a summarised version) automatically once it is collected. The process must also give the source an opportunity to rectify any irregularities in his/her data sets whenever these are justified.