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

CSU33012 Software Engineering
Measuring Engineering Report

Stephen Davis, Std# 18324401

November 14, 2021

Contents

1 Introduction

This report will discuss how software engineering can be measured, what platforms can be used to measure software engineering and what algorithms we can use to carry out such measuring. This report will then discuss the ethics of measuring software engineering, before finally concluding.

2 How Can Software Engineering be Measured?

There is much debate about whether software engineering can actually be measured. The question should not be whether software engineering can be measured, it should be how can software engineering be measured? It is worth noting that there are many ways we can measure software engineering, but not every way is accurate, insightful and fair. Most of the time the issue with these inaccurate measurements is that they lack context. It is worth noting that each measurement comes with its own advantages and disadvantages. Some examples of how to measure software engineering productivity which I will now discuss are:

- Lines of Code
- Number/Frequency of Commits
- Leadtime

2.1 Lines of Code

The most obvious method to measure software engineering is to measure the number of lines of code a software developer writes. This metric is considered to be "The Original Code Metric" (Harding, 2021). The advantages for this method are that it is very simple to measure and easy to understand this measurement. You do not need a technical background to understand how to analyse this metric.
Although these are some advantages to using Lines of Code (LOC) as a measurement, they are far out-weighed by the disadvantages. The most important disadvantage of LOC, is that it is "a very easy metric to game" (Pluralsight, 2016). Figure 1 and 2 below illustrate how one could gamify their code to appear more productive.

```
1) for(int i=0; i<6; i++) {
2)     System.out.println("Hello");
3) }
```

Figure 1: A basic for loop which executes 6 lines of output, but consists of 3 lines of code.

```
1) System.out.println("Hello");
2) System.out.println("Hello");
3) System.out.println("Hello");
4) System.out.println("Hello");
5) System.out.println("Hello");
6) System.out.println("Hello");
```

Figure 2: A manual version of the for loop seen in Figure 1. Both outputs are the same.

An engineer who writes code using a for loop such as that seen in figure 1, will produce less lines of code than an engineer who writes the code in figure 2. However, clearly, the code in figure 1 is more efficient and is better practice. This illustrates the point that LOC as a metric promotes inefficient, sloppy code and is far too

easy to game. This leads to inefficient code being rewarded, and those who practice efficient code to be punished. We must remember, "You don't pay a Michelangelo to make brush strokes, you pay him to be a genius" (Pluralsight, 2016).

2.2 Number/Frequency of Commits

At first, measuring the number of commits made by an engineer may appear to be beneficial as a metric for measuring productivity, however, just like Lines of Code (LOC), it is easily gamified. Our hope would be that software engineers commit code when they have made progress on a project, and by comparing the number of commits made by different developers, the higher the number, the more productive an engineer is. However, in theory, two engineers could essentially write the same code, except the first engineer commits the code when he has the functionality working, and let's say the second engineer commits every time he adds a new line of code. Now, of course, we do not want the latter to happen, but what is stopping this engineer, or any engineer, from doing so? Again, this illustrates how gamification can easily be utilised to improve your score when being analysed by the number of commits you make.
The disadvantages of using the number of commits made by an engineer is that it encourages insignificant and unnecessary commits. "You are incentivising them to make a commit every time they author a line of code" (Harding, 2021). The size and value of the commit is not taken into account, and this is why it is so easy to gamify. Perhaps one of the cruellest disadvantages of using the number/frequency of commits as a metric is that "if you're a hard-working developer that's striving to solve as many issues as possible, by simply saving their work more often, your lazier co-worker will shoot past you on the commit count leader board" (Harding, 2021). This metric rewards those who can game the system, whereas those who play fairly appear to be the software engineers falling behind in productivity.
One advantage of using the frequency of code commits as a metric is that if you notice it has been a while since an engineer has made a commit, this is "often a signal that they may be stuck" (Harding, 2021). However, it is quite clear that this one advantage is negligible when compared against all the disadvantages associated with this metric.
Again, as software engineers, we favour quality of code over quantity.

2.3 Leadtime

"Lead time quantifies how long it takes for ideas to be developed and delivered as software" (Altvater, 2017). To measure Leadtime, you "need to have a clear definition of when work begins and ends" (Lawrence, 2020). The idea behind using Leadtime as a metric is to measure how quickly engineers or a team of engineers can go from idea to production. If we notice one project idea has a significantly longer Leadtime than another, we may ask why? Perhaps a project idea was underspecified or there were permission requests which took a long time to come back. The advantages of this metric are that it incentivises software engineers to "improve how responsive they are to customers" (Altvater, 2017). Another advantage is that a company may realise from analysing this metric, that their Leadtime for releasing a fully-functional working version of the idea is too long, and their competitors may have beat them to the punch. By analysing the Leadtime, team leads may decide to "release little and often; you release each feature as it is ready, rather than waiting for an "all-in" big release" (Lawrence, 2020). This adapted approach would then keep the customers engaged and loyal to your company's product.

2.4 Conclusion

With any software engineering measurement, if you measure the wrong things, you risk pushing the quality of code down, which tends to reward mediocrity. Therefore, it is vital that if you are measuring software engineering, you choose a fair, insightful and well-thought-out metric, and consider the possible consequences of the metric you deploy.

3 What Platforms Can Be Used to Gather and Process Data?

Nowadays we have version control systems such as GitHub, which collect vast amounts of data. There is no doubt that this data exists, and that we have access to this data, but the question is how can we gather and process the specific data that we want. Thankfully, there are various platforms which provide such a service.

The examples I will discuss in this report are:

- Pluralisation Flow (Git Prime)
- WayDev
- Code Climate

3.1 Pluralisation Flow (Git Prime)

As previously mentioned, we can measure software engineering by the lines of code (LOC), the number of commits, Leadtime and various other metrics. These metrics individually are not much use and certainly not very insightful in the quest of measuring software engineering. However, by combining various different metrics, we can draw meaningful conclusions about our productivity as developers.
Pluralisation flow, previously called Git Prime, is a platform designed to collect various sources of information such as lines of code, pull requests, etc. from GitHub's wealth of data on software engineers, and analyse these metrics collectively, to produce meaningful insights.
Flow collects and configures historical git data into easy-to-understand insights and reports to assist software engineering teams in becoming more successful. Flow empowers you to be able to identify bottlenecks so you can remove them, compare trends and help your team reach their full potential. Based on the GitHub data flow collects and analyses from your team, it can then recommend skills which your team may need to develop.
"If we were not using Flow, it would be like going back to the Stone Age. We've gotten used to the metrics and the visibility Flow provides us" (Rob Teegarden, VPE, Dealersocket).
Pluralisation believes Git is more than version control. It is a record of how your team works. To align with this value, flow provides a powerful visualization into your teamwork dynamics. Questions such as what percentage of pull requests or commits get zero responses, what percentage of the team is involved in feedback and are senior engineers providing feedback and mentorship, are all answered by flow's powerful visualisation features.
A precarious position to find your team in is when your most informed, and up to speed developer(s), the central link in the team, is unavailable. Suddenly your team members may find themselves stuck, where no present members have the answer.

Figure 3: If the central link in your team becomes unavailable, the whole chain breaks down. This is where knowledge sharing reports come in handy.

Knowledge Sharing reports in Flow help you measure and improve how codebase knowledge is distributed across your team and empowers you to distribute value evenly across your engineering team.

(Pluralsight, 2021)

3.2 WayDev

The next platform on the list is WayDev. If you are looking to become more productive and happier with your work, then WayDev should be on the top of your list of platforms available to gather and process data. Earlier on in this report, we saw that a hard-working engineer would appear to be less productive than a lazy one, who knew how to game the system. However, with WayDev, we have justice. WayDev pride themselves on the fact that you can rest assured your achievements will be acknowledged. By being justly rewarded for all your hard work, you will be motivated to work harder, since you know your work is appreciated. This is a win-win situation, the productivity of the software engineer improves which is a win for the company, and the deserved engineer receives credit, acknowledgement and a sense of pride for his/her impactful efforts. Another feature WayDev provides is for you to ship your code faster, by equipping you with the tools necessary to visualise the codebase, pull requests, and tickets your team produce from sprint to sprint. WayDev also allows you to analyse how you spend your valuable time, how often you send code for review, and identifies those suggestive areas of improvement. Like Flow, WayDev unlocks the true value of data available to us on GitHub, and empowers us to learn, grow and improve, both individually and collectively, from this data.

(WayDev, 2021)

3.3 Code Climate

Code Climate advertises that you can "leverage [their] data-driven insights to build a culture of trust and high performance" (Climate, 2021). Code Climate understands the importance of meaningfully analysing data and puts emphasis on how one can "help every member of [their] team excel".

There are two main products offered by Code Climate:

1) Velocity
2) Quality

3.3.1 Velocity

Velocity is a powerful product offered by Code Climate, which has many overlapping features provided by Pluralsight and WayDev, such as insights into how you spend your time, recommendation on what skills your team can improve on etc. Since these features have already been discussed previously, I will move on to Quality, the second product offered by Code Climate.

3.3.2 Quality

Quality is a product offered by Code Climate which focuses on test coverage. The way this product works is it grades code on a scale from one to ten by running a suite of designed tests. The key benefit of these tests is that Code Climate will block a merge if tests fail, preventing problematic merges.

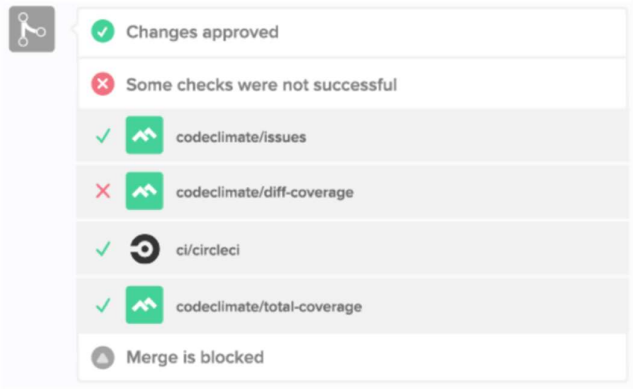

Figure 4: An attempted Merge is blocked by Quality's tests.

As mentioned in the beginning of this report, when being analysed under lines of code, software engineers can simply game the system. With Quality, this cheat-code is effectively dealt with, since the product neatly displays what files have changed, how much they have changed and how exactly they have changed.

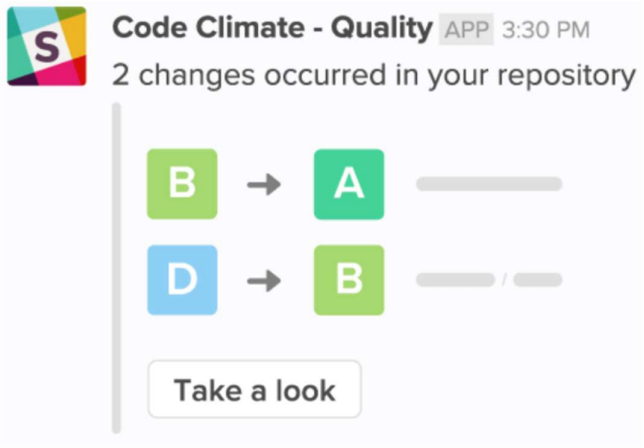

Figure 5: Quality highlights which files have changed, and provide functionality to analyse the changes in the various files.

If the same files appear to be changing again and again, perhaps this is an indication that a developer is in need of assistance. The power of Quality is that it saves an incredible amount of time that would have otherwise been spent manually reviewing code. Also, humans are susceptible to error, whereas machines running a suite of tests will not have room for human error. Quality prevents gamers from short-cutting their way to the top of the leader board, and ensures good programming practice and honesty is adhered to. (Climate, 2021)

4 What Algorithms Can We Use?

Once we have collected all the data that we need to measure software engineering, we then need to find some way to interpret this data in a meaningful way. Thankfully, there are numerous algorithms available in the software realm empowering us to do so. The algorithms which I will discuss in this report are:

- Halstead Complexity Measures
- Cyclomatic Complexity
- Maintainability Index

4.1 Halstead Complexity Measures

Maurice Howard Halstead introduced the algorithmic Halstead Complexity Measures in 1977 (Wikipedia, 2021). Halstead believed a computer program is considered to be a collection of tokens, which may be classified as either operators or operands, and that all software science metrics can be defined in terms of these basic symbols (javaTpoint, n.d.).
To begin using this algorithm, it is necessary to look at the basic measures which will be used in subsequent calculations. These are:

n1 = count of unique operators.
n2 = count of unique operands.
N1 = count of total occurrences of operators.
N2 = count of total occurrence of operands.

Figure 6: The Halstead complexity measures.

After establishing what the Halstead Complexity measures are, one can then progress towards looking at the Halstead metrics listed below.

Measure	Symbol	Formula
Program length	N	N= N1 + N2
Program vocabulary	n	n= n1 + n2
Volume	V	V= N *$\log_2$ (n)
Effort	E	E= V/2*n2
Halstead Program Length	H	H = n1*$\log_2$(n1) + n2*$\log_2$(n2)

Figure 7: The Halstead Complexity Metrics, alongside their corresponding symbols and formulae. Source: (ResearchGate, n.d.)

An example of a codebase analysed by the Halstead Complexity Measures is seen below.

```
int sort (int x[ ], int n)

{
    int i, j, save, im1;
    /*This function sorts array x in ascending order */
    If (n< 2) return 1;
    for (i=2; i< =n; i++)
    {
        im1=i-1;
        for (j=1; j< =im1; j++)
            if (x[i] < x[j])
            {
                Save = x[i];
                x[i] = x[j];
                x[j] = save;
            }
    }
    return 0;
}
```

Figure 8: Sample codebase used for illustrating Halstead Complexity Measures.

operators	occurrences	operands	occurrences
int	4	sort	1
()	5	x	7
,	4	n	3
[]	7	i	8
if	2	j	7
<	2	save	3
;	11	im1	3
for	2	2	2
=	6	1	3
-	1	0	1
<=	2	-	-
++	2	-	-
return	2	-	-
{}	3	-	-
n1=14	N1=53	n2=10	N2=38

Figure 10: Illustration of operators and operands in the above codebase, used to calculate Halstead Complexity Measures n1, N1, n2, N2.

```
Therefore,
N = 91
n = 24
V = 417.23 bits
```

Figure 9: Sample calculation of Volume (V) Halstead Complexity Metric using n1, N1, n2, N2 measures.

By applying the corresponding formula for each metric as seen in figure 7 above, we can then compare metrics from one project against metrics of another project. We can then inquire as to why one project had a greater program length compared to another, for example.

(javaTpoint, n.d.)

4.2 Cyclomatic Complexity

Developed by Thomas J. McCabe, Sr, in 1976, "cyclomatic complexity is a software metric used to indicate the complexity of a program" (Wikipedia, 2021). What cyclomatic complexity of a section of source code measures is the number of linearly independent paths within this section of code.
The formula for calculating the cyclomatic complexity of a section of code, which we represent with the letter M, is:

M = E − N + 2P, where

E = the number of edges of the graph.
N = the number of nodes of the graph.
P = the number of connected components.

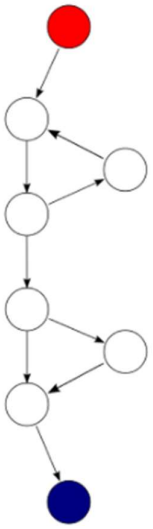

Figure 11: Sample graph illustrating the factors necessary to calculate the cyclomatic complexity. The graph has 9 edges (E), 8 nodes (N), and 1 connected component (P).

The logic behind cyclomatic complexity is that the more decisions that have to be made in code, the more complex it is. The effect of various constructs contained in code on the cyclomatic complexity (CC) can be seen in Figure 12 below.

Construct	Effect on CC	Reasoning
if	+1	An `if` statement is a single decision.
elif	+1	The `elif` statement adds another decision.
else	+0	The `else` statement does not cause a new decision. The decision is at the `if`.
for	+1	There is a decision at the start of the loop.
while	+1	There is a decision at the `while` statement.
except	+1	Each `except` branch adds a new conditional path of execution.
finally	+0	The finally block is unconditionally executed.
with	+1	The `with` statement roughly corresponds to a try/except block (see PEP 343 for details).
assert	+1	The `assert` statement internally roughly equals a conditional statement.
Comprehension	+1	A list/set/dict comprehension of generator expression is equivalent to a for loop.
Boolean Operator	+1	Every boolean operator (and, or) adds a decision point.

Figure 12: A table illustrating the effect various constructs have on the cyclomatic complexity number assigned to a section of code. Source: (Radon, n.d.)

At the beginning of this report, lines of code (LOC) was discussed as a metric, and pointed out to be easily gamed. LOC was linked to bad programming practice, where developers simply write more lines of code, and in the software engineering world, a popular opinion is that the more lines of code a function has, the more likely it is to have errors. However, when you combine cyclomatic complexity with lines of code, then you have a much clearer picture of the potential for errors. Just because you have more lines of code, it does not necessarily mean you are more prone to having errors in your code.

As an algorithmic approach, cyclomatic complexity can be used to "get a sense of how hard any given code may be to test, maintain, or troubleshoot as well as an indication of how likely the code will be to produce errors" (Microsoft, 2021). The value in cyclomatic complexity as an algorithmic approach is that it assigns a

complexity number, or rating, to the code, where the higher the complexity number, the greater the probability of errors. Naturally, the more errors code contains, the more time and effort it takes to maintain and troubleshoot this codebase. Implementing the cyclomatic complexity algorithm on your work will guide you as to where you can improve your code. After establishing any functions with a high complexity value, you can then try to refactor them to reduce their complexity.

4.3 Maintainability Index

The last algorithm this report will discuss is the Maintainability Index. One could argue that the logic behind this algorithm is the simplest to understand, especially for somebody who does not come from a technical background. The maintainability index simply measures how maintainable (easy to support and change) source code is, assigning the code an index value between 0 and 100. The higher the index value, the more maintainable the code is. It is that simple. The maintainability index is calculated as a factored formula consisting of Lines of Code, Cyclomatic Complexity and Halstead volume, all of which have already been discussed in this report. Since the Maintainability Index formula consists of these figures, the first step in using this algorithm is to measure these aforementioned metrics. Once we have these metrics, we can then use the Maintainability Index formula.

The original formula for calculating the Maintainability Index (MI) of code is:

$$MI = 171 - 5.2 \ln V - 0.23G - 16.2 \ln L$$

Figure 13: The original formula for calculating Maintainability Index (MI) of code. Source: (Radon, n.d.)

Many companies have adopted this algorithmic metric and some have tweaked the original formula slightly, to get their own preferred variant. One such example is Microsoft Visual Studio 2010, whose altered formula is listed below.

$$MI = \max\left[0, 100\,\frac{171 - 5.2 \ln V - 0.23G - 16.2 \ln L}{171}\right].$$

Figure 14: Microsoft Visual Studio 2010's altered version of the Maintainability Index formula. Source: (Radon, n.d.)

(CodeGrip, n.d.)

The key for reading these formulas is listed here:

Where:

- V is the Halstead Volume (see below);
- G is the total Cyclomatic Complexity;
- L is the number of Source Lines of Code (SLOC);

Figure 15: Key for understanding Maintainability Index formulas listed above. Source: (Radon, n.d.) Note: Source Lines of Code (SLOC) is the same thing as Lines of Code (LOC).

It is clear from the above formulas that the most influential factor in the calculation of the Maintainability Index is the Lines of Code. This is because the coefficient of the Lines of Code, 16.2, far exceeds the coefficient of the Halstead's Volume, of 5.2ln, and of the Cyclomatic Complexity, which is 0.23.

4.4 Conclusion

The most important point to take from discussing these algorithmic approaches is not which approach to use, it is to highlight the effectiveness of using any one of these algorithms. The point is, that by utilising any one of, or combination of, these algorithms, developers can understand which parts of their code should be reworked or more thoroughly tested. Development teams can identify potential risks, understand the current state of a project, and track progress during software development (Microsoft, 2021)

5 Is This Ethical?

A very sensitive topic relating to measuring software engineering is whether it is ethical. Up to now, this report has highlighted the benefits of measuring software engineering, such as increasing awareness of which parts of the codebase need to be reworked. However, one must consider the potential invasion of privacy on those being measured, as well as various other moral issues. There are many different reasons for and against measuring software engineering, and hence, different people form different opinions. The view taken in this report is only my personal view, one of hundreds of thousands, and it is important we respect each and every one of these views. The key topics this report will focus on for our discussion on ethics are:

- Moral Issues – Privacy
- Benefits of Measuring Software Engineering
- Do Benefits Outweigh the Cons?

5.1 Privacy

A big concern in measuring software engineering is the feeling of an invasion of privacy. A Netflix documentary called "The Great Hack" focuses on a company called Cambridge Analytica, who used personal data of Facebook users and their friend connections on the platform, to manufacture specific speeches for Donald Trump's campaign. The popularity of this documentary exploded on Netflix because viewers resonated with the topic. All of us, including the viewers of this documentary, suddenly felt their privacy was under threat, and could easily be violated, without them even knowing. This topic applies to measuring software engineering, because although the data being measured is different, the underlying concept is the same – a person's data is being analysed, sometimes without them even knowing.
There are several ways to resolve this invasion of privacy conflict, such as informing those that are being measured, and only measuring those that give their consent. In addition to this, participation in being measured should be completely optional. Once the above criteria are met, one cannot claim that there is an invasion of their privacy when they are being measured. They have been informed, and asked whether it is ok, before measuring their data. The key take-away here is to get consent before commencing the measuring.

5.2 Benefits of Measuring Software Engineering

A big benefit of measuring software engineering, is that the insights provided from such measurement helps developers grow – "by understanding why their code is complex. developers learn to improve their code in the future." More examples of how measuring software engineering helps individual engineers and software engineering teams is that they can identify areas of improvement, manage workloads (automated testing) and increase return on investment by reducing Leadtime. An often-overlooked benefit of measuring software engineering is that the measurements carried out on the codebase will identify errors and issues in the codebase in advance, before they get out of hand. The idea here is that prevention is the best form of treatment.
Remember, what gets measured, gets managed (Pluralsight, 2021).

5.3 Do Benefits Outweigh the Cons?

The trouble with deciding whether measuring software engineering is ethical is that we have no metric to use to compare the invasion of privacy against the increase in productivity which arises from measuring software engineering. We have no statistical figure or common metric that can be applied to both, to see which one outweighs the other.

However, we can take certain steps to make the practice of measuring software engineering ethical, or at least somewhat more ethical, or less unethical. To do this, one must acknowledge the potential downsides, and incorporate a solution to these downsides into the overall process. As discussed, the most obvious reason for being against measuring software engineering is the feeling of an invasion of privacy, and the possible solutions to this are to ask for consent, notify the engineers being measured that they are in fact being measured, and to make participation optional. By doing so, instead of trying to decide whether the benefits outweigh the cons, we can simply reduce the effect of the cons, ultimately, and hopefully, leading to a majority consensus view that measuring software engineering should be acceptable, and ethical. The benefits of measuring software engineering are extremely valuable, and once the measurement process is carried out in the aforementioned manner, there is no reason not to measure software engineering. We have so much to gain, and very little, if anything, to lose.

6 Conclusion

There are many metrics we can use to measure software engineering such as Lines of Code (LOC), number of commits and Leadtime. However, these metrics alone are insufficient to measure software engineering and are simply too easy to game, promoting inefficient, sloppy practice. The collection of data using the aforementioned metrics can be carried out by platforms such as Pluralisation, WayDev and Code Climate. Algorithmic approaches such as Halstead Complexity Measures, Cyclomatic Complexity and the Maintainability Index can be used to analyse the complexity and maintainability of code, providing accurate insights into the work of the software engineers being measured. Ethics is a constant heated discussion in the software world, with developers and customers alike debating whether or not software engineering should be measured. The ultimate conclusion seen in this report being that provided certain steps are taken to reduce the feeling of an invasion of privacy, software engineering should be measured, because we have so much to gain from doing so, and so little to lose.

7 References

Altvater, A., 2017. *Stackify.* [Online]
Available at: https://stackify.com/track-software-metrics/
Climate, C., 2021. *Code Climate.* [Online]
Available at: https://codeclimate.com/
CodeGrip, n.d. *CodeGrip.* [Online]
Available at: https://www.codegrip.tech/productivity/a-simple-understanding-of-code-complexity/
Harding, B., 2021. The 4 Worst Software Metrics Agitating Developers. *GitClear.*
javaTpoint, n.d. *javeTpoint.* [Online]
Available at: https://www.javatpoint.com/software-engineering-halsteads-software-metrics
Lawrence, C., 2020. *Humanitec.* [Online]
Available at: https://humanitec.com/blog/lead-time-a-key-metric-in-devops
Microsoft, 2021. *Microsoft.* [Online]
Available at: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-cyclomatic-complexity?view=vs-2022
Microsoft, 2021. *Microsoft.* [Online]
Available at: https://docs.microsoft.com/en-us/visualstudio/code-quality/code-metrics-values?view=vs-2022
Pluralsight, 2016. Lines of code is a worthless metric. Except when it isn't. *Pluralsight,* 4 May.
Pluralsight, 2021. *Pluralsight.* [Online]
Available at: https://www.pluralsight.com/product/flow
Radon, n.d. *Radon.* [Online]
Available at: https://radon.readthedocs.io/en/latest/intro.html
ResearchGate, n.d. *ResearchGate.* [Online]
Available at: https://www.researchgate.net/figure/Halstead-Complexity-Metrics_tbl1_319481865
WayDev, 2021. *WayDev.* [Online]
Available at: https://waydev.co/software-engineers/
Wikipedia, 2021. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Halstead_complexity_measures
Wikipedia, 2021. *Wikipedia.* [Online]
Available at: https://en.wikipedia.org/wiki/Cyclomatic_complexity