Project Report

For

Deliverable 2

METRICSTICS

Submitted By

Amrinderpreet Singh
Anant Bir Singh
Gaganpreet Singh
Prabhjot Singh
Rahul Singh
Piyush Singla

Submitted to

Prof. Pankaj Kamthan

SOEN 6611 - Software Measurement

Fall 2023

Concordia University, Montreal, QC

Table of Contents

1 Introduction

This report delves into the creation and evaluation of METRICSTICS, a comprehensive system designed for the calculation and analysis of various descriptive statistics related to software projects. In this major deliverable, we address key challenges and implement solutions to fulfill the objectives set forth for METRICSTICS.

1.1 Motivation

Accurate metrics provide insights into the characteristics and performance of software, aiding in decision-making, resource allocation, and project management. By facilitating the analysis of datasets through measures like mean absolute deviation, standard deviation, and cyclomatic complexity, METRICSTICS equips users with valuable insights into the characteristics and complexities of their software projects. This, in turn, contributes to informed decision-making, efficient project management, and the overall improvement of software quality.

1.2 Objective

The primary objective of this report is to document the development and assessment of METRICSTICS. We aim to provide a detailed account of the methodologies employed and outcomes achieved in the course of implementing a software system that addresses various facets of software metrics. Through METRICSTICS, we seek to contribute to the advancement of software measurement practices.

2 PROBLEM 3: Project Effort Estimation

2.1 Effort Estimation using Use Case Point (UCP) Approach

Effort Estimate using use case points is given by:

$$\text{EffortEstimate} = UCP \cdot PF$$

where:

- UCP is Use Case Points
- PF is Productivity Factor

UCP is calculated as:

$$UCP = UUCP \cdot TCF \cdot ECF$$

where:

- $UUCP$ is Unadjusted Use Case Points
- TCF is Technical Complexity Factor
- ECF is Environmental Complexity Factor

$UUCP$ is calculated as:

$$UUCP = UAW + UUCW$$

where:

- UAW is Unadjusted Actor Weight
- $UUCW$ is Unadjusted Use Case Weight

Given values:

$$UAW = 3$$
$$UUCW = 10$$

$$UUCP = UAW + UUCW = 3 + 10 = 13$$

Technical Complexity Factor (TCF)
TCF considers technical concerns that can impact the software project. There are 13 different Technical Complexity Factors with their respective weights:

TCF Type	Description	Weight
$T1$	Distributed System	2
$T2$	Performance	3
$T3$	End User Efficiency	2
$T4$	Complex Internal Processing	3
$T5$	Reusability	1
$T6$	Easy to Install	0.5
$T7$	Easy to Use	0.5
$T8$	Portability	2
$T9$	Easy to Change	1
$T10$	Concurrency	1
$T11$	Special Security Features	1
$T12$	Provides Direct Access for Third Parties	1
$T13$	Special User Training Facilities are Required	1

The values for influence are:

$$\begin{aligned} \text{Value of no influence} &= 0 \\ \text{Value of average influence} &= 3 \\ \text{Value of strong influence} &= 5 \end{aligned}$$

TCF is calculated as:

$$TCF = C1 + (C2 \cdot \sum_{i=1}^{13} (WT_i \cdot Fi))$$

where:

$$\begin{aligned} C1 &= 0.6 \\ C2 &= 0.01 \end{aligned}$$

$$\begin{aligned} TCF = 0.6 &+ (0.01 \cdot ((2 \cdot 0) + (3 \cdot 3) + (2 \cdot 3) + (3 \cdot 3) + (1 \cdot 5) + (0.5 \cdot 5) \\ &+ (0.5 \cdot 5) + (2 \cdot 5) + (1 \cdot 5) + (1 \cdot 0) + (1 \cdot 0) + (1 \cdot 0) + (1 \cdot 0))) \\ &= 0.6 + (0.01 \cdot 64) = 1.24 \end{aligned} \quad (1)$$

Environmental Complexity Factor (ECF)
ECF accounts for the development team's personal traits and experience. There are 8 Environmental Complexity Factors with their respective weights:

ECF Type	Description	Weight
$E1$	Familiarity with Use Case Domain	1.5
$E2$	Part-Time Workers	-1
$E3$	Analyst Capability	0.5
$E4$	Application Experience	0.5
$E5$	Object-Oriented Experience	1
$E6$	Motivation	1
$E7$	Difficult Programming Language	-1
$E8$	Stable Requirements	2

The values for influence are:

Case: E1, E3, E4, E5, E6, and E8

0 : No influence
1 : Strong, Negative influence
3 : Average influence
5 : Strong, Positive influence

Case: E2, E7

0 : No influence
1 : Strong, Favorable influence
3 : Average influence
5 : Strong, Unfavorable influence

ECF is calculated as:

$$ECF = C1 + (C2 \cdot \sum_{i=1}^{8} (WE_i \cdot Fi))$$

where:

$$C1 = 0.14$$
$$C2 = -0.03$$

$$\begin{aligned} ECF &= 1.4 + (-0.03 \cdot ((1.5 \cdot 3) + (-1 \cdot 0) + (0.5 \cdot 5) + (0.5 \cdot 5) \\ &\quad + (1 \cdot 5) + (1 \cdot 5) + (-1 \cdot 1) + (2 \cdot 5))) \\ &= 1.4 + (-0.03 \cdot 28.5) = 0.145 \end{aligned}$$

Calculating UCP and Effort Estimate
Finally, UCP and Effort Estimate are calculated as:

$$UCP = UUCP \cdot TCF \cdot ECF = 13 \cdot 1.24 \cdot 0.145 = 2.3876$$

$$\text{Effort Estimate} = UCP \cdot PF = 2.3876 \cdot 20 = 47.752$$

2.2 Effort Estimation using Basic COCOMO 81

The Basic COCOMO 81 model is used to estimate effort for a software project based on the size of the project in Kilo Lines of Code (KLOC) and its complexity type. The formula for estimating effort is:

$$\text{Effort (E)} = a \cdot (KLOC)^b$$

Where:

- E is the effort.
- $KLOC$ is the size of the project in Kilo Lines of Code (1 KLOC = 1000 lines of code).
- a and b are coefficients that depend on the project type and complexity.

The COCOMO 81 model classifies projects into three types:

1. Organic: Simple, well-understood projects.
2. Semi-detached: Intermediate projects with some degree of complexity.
3. Embedded: Complex, real-time projects.

The values of a and b for the organic project type are as follows:

- Organic:

$$a = 2.4$$
$$b = 1.05$$

To estimate the effort for our project, you need to determine the size of our project in KLOC and classify it as organic based on its simplicity. Then, use the values of a and b to calculate the effort.

For example, if our project is 50 KLOC and you classify it as "organic," you can calculate the effort as follows:

$$E = 2.4 \cdot (0.542)^{1.05}$$

Now, plug in the values to get the estimated effort:

$$E = 2.4 \cdot 0.525 \approx 1.262$$

So, the estimated effort for our project using the Basic COCOMO 81 model is approximately 1.262.

2.3 Comparison of Estimates and Actual Effort

In this section, we will compare the effort estimates obtained using the Use Case Point (UCP) approach and the Basic COCOMO 81 model with the actual effort towards the project.

Estimates using UCP Approach
Using the UCP approach, we estimated the effort required for the project to be approximately 47.752.

Estimates using Basic COCOMO 81
Using the Basic COCOMO 81 model with the project classified as "organic," we estimated the effort required for the project to be approximately 1.262.

Actual Effort
Now, let's consider the actual effort that was required to complete the project. After completing the project, the actual effort spent on the project was measured to be around 3 person days.

Reasons for Variation
There can be several reasons for the variation between the estimated effort and the actual effort for a software project. Some of the common reasons which we think are:

- **Team Experience:** The experience and expertise of the development team can significantly impact the actual effort required. A more experienced team may complete the project more efficiently.
- **Scope Changes:** Changes in project requirements or scope during development can lead to variations in effort. Additional features or modifications can increase effort.
- **Technology Challenges:** Unforeseen technical challenges or issues can arise during development, requiring additional effort to resolve.
- **Project Management:** Effective project management and communication can mitigate variations, while poor management can lead to increased effort.
- **External Factors:** External factors such as market changes, regulatory requirements, or unexpected events can impact project timelines and effort.

It's essential to regularly monitor and adapt project plans to address these variations and ensure successful project completion.

3 PROBLEM 4: Implementation of METRICSTICS

We have implemented METRICSTICS using object-orientation as the programming paradigm and Python as the programming language. The corresponding code is accessible via our GitHub link.

You can access the GitHub repository at:
https://github.com/theOGCodeWitcher/SOEN-6611-METRICSTICS

3.1 Implementation Details and Design Choices

Our application provides users with two choices: they can either upload their own data file or opt to generate random data for metric calculations.

- **Programming Style:** The overall structure of the code adheres to **PEP 8** style guide recommendations
- **Modularity:** The code is organized into multiple files (main.py, data_processor.py, and helper.py), each serving a specific purpose. The main functionality is encapsulated within the DataProcessor class and the use of helper methods within the Helper class further supports modularity by separating distinct functionalities.
- **Internal Reuse** The generate_random_data method in the Helper class is reusable for generating random data, promoting a modular and reusable approach.

3.2 Algorithm Selection and Rationale

We have used the Merge sort, a "divide and conquer" algorithm that efficiently handles the sorting of data by breaking the input into smaller segments, sorting each segment independently, and then merging them back together.

Rationale for selection

- Merge sort has a consistent time complexity of O(n log n) in the worst case. This performance is reliable and efficient, regardless of the specific distribution or characteristics of the input data. For a fixed size of 1000 elements, this performance remains reasonable.
- Merge sort has a straightforward and modular implementation. The divide-and-conquer approach makes the algorithm easier to understand, maintain, and modify. This is beneficial for code readability and future development efforts.

4 PROBLEM 5: Cyclomatic Complexity

Cyclomatic complexity is a measure of the number of linearly independent paths through a program's source code. It provides an indication of the program's complexity and potential difficulty of maintenance.

METRICSTICS is organized into three directories: main.py contains the main

graphical user interface (GUI) implemented using Tkinter, handling user interactions and displaying descriptive statistics. data_processor.py includes the DataProcessor class responsible for processing input data, calculating statistics such as mean, median, mode, and managing file reading or random data generation. The helper.py module houses the Helper class, providing utility functions for reading different file types, performing arithmetic operations, and aiding in sorting algorithms.

4.1 Calculation

Radon is a Python library designed for code analysis and complexity measurement. Leveraging its capabilities, we utilized Radon to calculate cyclomatic complexity of METRICSTICS. Fig.1 illustrates the results obtained. The first letter signifies Block type, Blocks are also classified into three types: functions, methods and classes (see Table 1) followed by block position and name. The last column signifies Cyclomatic number of a block. Every block is ranked from A (best complexity score) to F (worst one). Ranks corresponds to complexity scores as shown in table 2.

Table 1. Block representation

Block Type	Letter
Function	**F**
Method	**M**
Class	**C**

Table 2. Complexity scores and Ranks

CC Score	Rank	Risk
1 - 5	**A**	Low - simple block
6 - 10	**B**	Low - well structured and stable block
11 -20	**C**	Moderate - slightly complex block
21 - 30	**D**	More than moderate - more complex block
31 - 40	**E**	High - complex block, alarming
41+	**F**	Very High - error-prone, unstable block

The Average Cyclomatic complexity of METRICSTICS is calculated as **3.129.**

4.2 Qualitative Conclusions

Average Cyclomatic Complexity:
The average complexity of the functions/methods in METRICSTICS is around

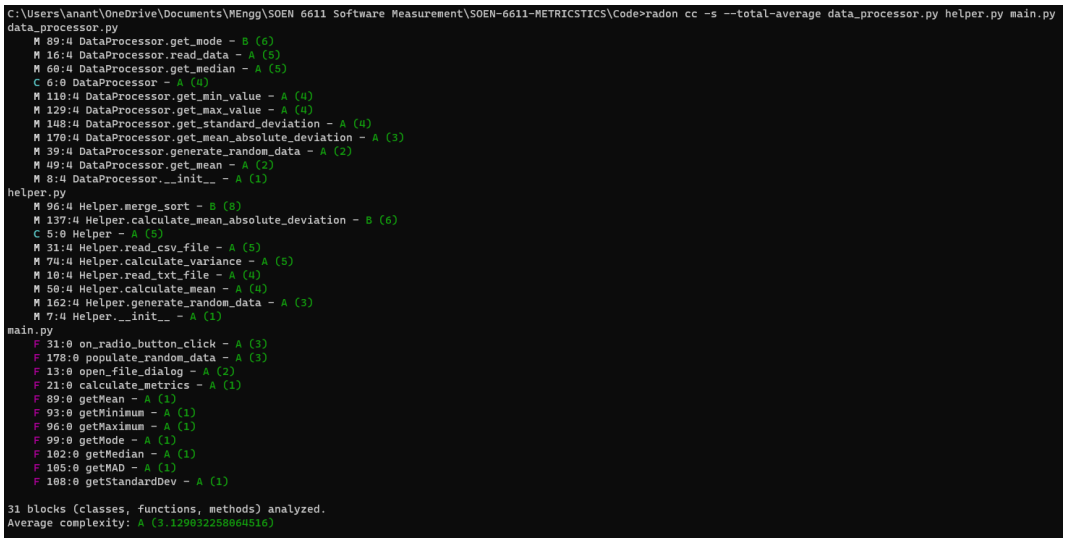

Fig. 1.
Cyclomatic complexity of various blocks of METRICSTICS

3.13, falling within the low complexity range. This suggests that, on average, the functions/methods are relatively simple and straightforward, promoting maintainability and readability of the codebase.

Functions/Methods Analysis:
Most of the individual functions/methods have low cyclomatic complexity values, which is generally positive for maintainability and readability. Examining specific functions/methods further supports this observation. For instance, functions like calculate_mean, calculate_median, and calculate_mode demonstrate low individual cyclomatic complexity values, contributing to the overall low average. The simplicity of these functions enhances the codebase's overall comprehensibility and reduces the likelihood of errors during maintenance or future development.

5 PROBLEM 6: Object-oriented metrics

The object-oriented metrics, WMC, CF, and LCOM*, for both the classes in METRICSTICS are calculated below.

5.1 DataProcessor class

Weighted Methods per Class (WMC):
"Weighted Method per Class" (WMC) is a metric used in software engineering to

measure the complexity of a class in object-oriented programming. It is a way of quantifying the complexity of a class by assigning weights to different methods based on their characteristics.

The WMC metric is often calculated by summing up the complexity values assigned to each method within a class. The complexity of a method can be assessed based on factors such as the number of control structures (e.g., loops, conditionals), the number of parameters, and other structural attributes.

We already calculated values of complexity for each method in problem 5 so we'll be directly taking the values from there.

The DataProcessor class has the following methods:

1. __init__ → Complexity : 1
2. read_data → Complexity : 5
3. generate_random_data → Complexity : 2
4. get_mean → Complexity : 2
5. get_median → Complexity : 5
6. get_mode → Complexity : 6
7. get_min_value → Complexity : 4
8. get_max_value → Complexity : 4
9. get_standard_deviation → Complexity : 4
10. get_mean_absolute_deviation → Complexity : 3

WMC = 1 + 5 + 2 + 2 + 5 + 6 + 4 + 4 + 4 + 3 = 36

Coupling Factor (CF):
CF measures the interdependence between classes. It can be assessed by examining the dependencies of a class on other classes. In this case, DataProcessor has a dependency on the Helper class.

For DataProcessor, CF = 1 (due to one dependency on Helper).

The coupling factor of 1 indicates that there is a direct dependency on one external class (helper). Lower coupling factors are generally desirable, as they suggest lower interdependence between classes, leading to better modularity and maintainability.

Lack of Cohesion in Methods (LCOM*):
LCOM* measures the lack of cohesion within a class. It is calculated based on the number of pairs of methods that do not share instance variables.

In this class, there are no attributes (instance variables) explicitly defined. Therefore, the number of attributes accessed (a) is 0. Consequently, the LCOM* formula simplifies to:

LCOM*= m/1

Here, m is the number of methods in the class. Since there are 9 methods in the DataProcessor class, the LCOM* value would be: LCOM* = 9/1=9

Therefore, the Lack of Cohesion in Methods (LCOM*) for the DataProcessor class is 9.

5.2 Helper class

Weighted Methods per Class (WMC):

The Helper class has the following methods:

1. __init__ → Complexity : 1
2. read_txt_file → Complexity: 4
3. read_csv_file → Complexity: 5
4. calculate_mean → Complexity: 4
5. calculate_variance → Complexity: 5
6. merge_sort→ Complexity: 8
7. calculate_mean_absolute_deviation → Complexity: 6
8. generate_random_data → Complexity: 3

WMC = 1 + 4 + 5 + 4 + 5 + 8 + 6 + 3 = 36

Coupling Factor (CF):
CF for Helper class = 2

In this class, there are two external dependencies:

The csv module (imported at the beginning of the file).
The random module (imported at the beginning of the file).

Therefore, the coupling factor (CF) for helper.py is 2.

CF=Number of Dependencies/Number of Classes=2/1=2

The coupling factor of 2 indicates that there are two external dependencies in the Helper class. Lower coupling factors are generally desirable, as they suggest lower interdependence between classes, leading to better modularity and maintainability.

Lack of Cohesion in Methods (LCOM*):
In this class, there are no attributes (instance variables) explicitly defined. Therefore, the number of attributes accessed (a) is 0. Consequently, the LCOM* formula simplifies to:

LCOM*= m/1

Here, m is the number of methods in the class. Since there are 7 methods in the Helper class, the LCOM* value would be: LCOM* = 7/1=7

Therefore, the Lack of Cohesion in Methods (LCOM*) for the Helper class is 7.

5.3 Qualitative Conclusions

DataProcessor class
Weighted Method Per Class (WMC): 36
Conclusion: The class exhibits a high level of complexity with a WMC of 36. Developers should carefully review the methods to identify opportunities for refactoring and simplification.

Coupling Factor (CF): 1
Conclusion: The class has low coupling with other classes (CF = 1), indicating good encapsulation. This is positive for maintainability and modularity.

Lack of Cohesion in Methods (LCOM)*: 9
Conclusion: The LCOM* value is moderate, suggesting some room for improvement in terms of method cohesion. Consider restructuring methods for better organization.

Helper class
Weighted Method Per Class (WMC): 36
Conclusion: Similar to data_processor.py, the class has a high WMC, indicating complexity. Reviewing and potentially refactoring methods may enhance maintainability.

Coupling Factor (CF): 2
Conclusion:The class exhibits moderate coupling (CF = 2). While not extremely high, developers should be mindful of dependencies to maintain flexibility.

Lack of Cohesion in Methods (LCOM)*: 7
Conclusion: The LCOM* value is moderate, suggesting some room for improvement in terms of method cohesion. Consider restructuring methods for better organization.

6 PROBLEM 7: Physical And Logical SLOC

6.1 Physical SLOC

Physical SLOC refers to the number of lines in the source code of a software project that contain executable statements, meaning lines that have actual code

instructions and are not blank lines or comments.

Using Radon
We'll use the Radon library to calculate the physical SLOC. Radon is a Python library that provides various code quality metrics, including physical lines of code. Here's how to use it:

Installing Radon
We installed Radon using pip, the Python package manager. Open your terminal and run the following command to install Radon:

```
pip install radon
```

Calculating Physical SLOC with Radon
To calculate the physical SLOC using Radon, navigate to the main directory of your project in your terminal and use the following command:

```
radon raw main
```

This command will provide you with a report that includes physical lines of code, as well as other code quality metrics.

6.2 Logical SLOC

It is a measure of the size or complexity of a software project based on the number of logical statements or instructions in the source code that affect the program's control flow or behavior.

Calculating Logical SLOC with Radon
To calculate the physical SLOC using Radon, navigate to the main directory of your project in your terminal and use the following command:

```
radon raw main
```

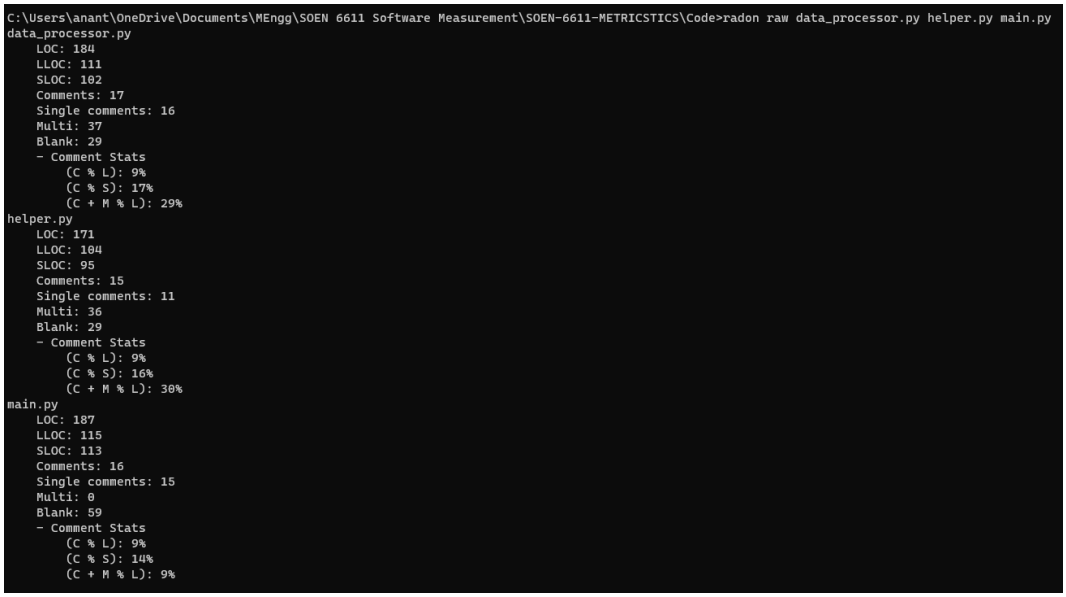

Fig. 2. Values of Physical and Logical SLOC using Radon

6.3 Qualitative Conclusions

A Logical SLOC of 330, and Physical SLOC of 542 indicate a small size system. It suggests that the software project is small in comparison to generic systems and may have a simpler and more manageable source code.

7 PROBLEM 8:WMC vs Logical SLOC

7.1 Scatter Plot:

Table 3. WMC vs Logical SLOC

Class	WMC	Logical SLOC
DataProcessor	36	111
Helper	36	104

The Weighted Methods per Class (WMC) and Logical Source Lines of Code (Logical SLOC) are both metrics used to measure the complexity or size of a class in software metrics. In general, a higher WMC or Logical SLOC value indicates greater complexity or size. Let's analyze the correlations between these metrics for the provided classes:

DataProcessor:
WMC: 36
Logical SLOC: 111
There is a positive correlation between WMC and Logical SLOC, as both metrics indicate the complexity of the class. In this case, the correlation is moderate, as the WMC is 36 and the Logical SLOC is 111. This suggests that the class has a moderate level of complexity relative to its size.

Helper:
WMC: 36
Logical SLOC: 104
Similar to the DataProcessor class, there is a positive correlation between WMC and Logical SLOC. The WMC is 36, and the Logical SLOC is 104, indicating a moderate level of complexity relative to size.

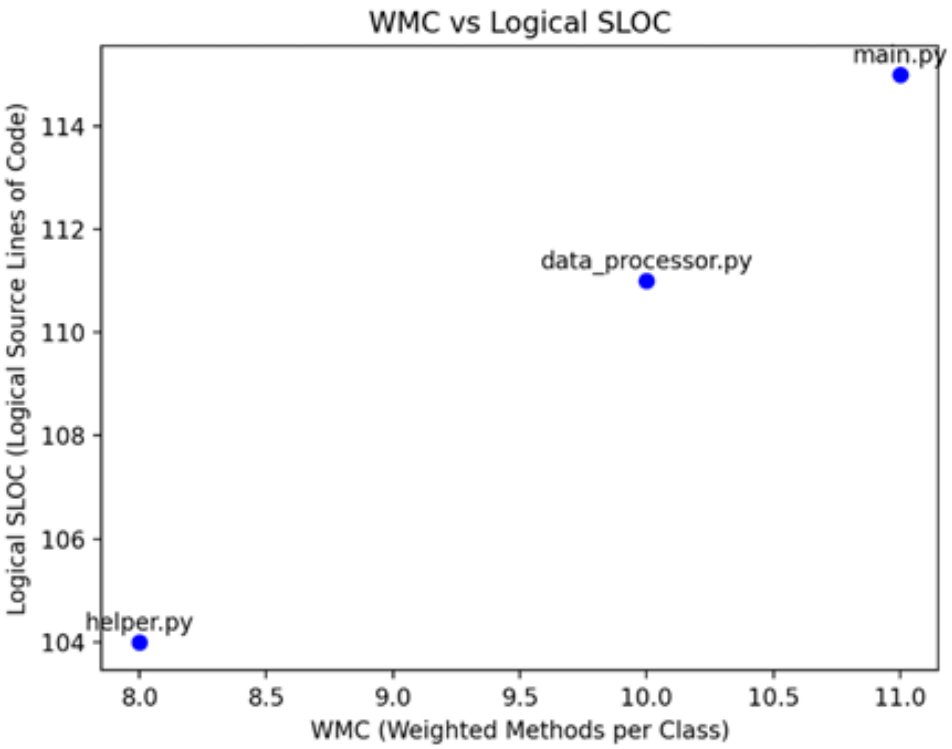

Fig. 3. WMC vs Logical SLOC

7.2 Correlation Coefficient

The scatter plot above indicates that the values of x's and y's are not normally distributed. Consequently, the

WMC_{xi}	$Rank_{xi}$	$SLOC$	$Rank_{yi}$	d	d^2
36	2	111	2	0	0
36	2	104	1	1	1
16	1	115	3	-2	4

Statistics for calculating r_s

$r_s = 1 - \frac{6\sum_{i=1}^{3} D^2}{n(n^2-1)}$

$r_s = 1 - \frac{30}{3\times 8}$

$r_s = 1 - 1.25$

$r_s = -0.25$

The value -0.25 indicates a weak negative correlation.

8 Conclusion

In conclusion, this project report provides a comprehensive overview of the development, implementation, and evaluation of a software system aimed at calculating and analyzing various descriptive statistics related to software projects. The report outlines methodologies, and presents outcomes achieved in the course of implementing METRICSTICS.Through thorough documentation and analysis, the report enhances our understanding of software measurement practices, offering a valuable resource for both practitioners and researchers in the field.

9 Glossary

1. **Scatter Plot:** A type of plot that displays individual data points on a two-dimensional graph. It is used to visualize the relationship between two continuous variables.
2. **Spearman's Rank Correlation Coefficient (r_s):** A measure of statistical dependence between the rankings of two variables. It assesses how well the relationship between two variables can be described using a monotonic function.
3. **Correlation Coefficient:** A numerical measure that quantifies the strength and direction of a linear relationship between two variables. The correlation coefficient ranges from -1 to 1, where -1 indicates a perfect negative correlation, 1 indicates a perfect positive correlation, and 0 indicates no correlation.
4. **Monotonic Function:** A function that preserves or reverses the order of elements in its domain. In the context of Spearman's rank correlation, it measures the strength and direction of a monotonic relationship between two variables.
5. **Weak Negative Correlation:** A correlation between two variables where an increase in one variable tends to be associated with a decrease in the other, but the relationship is not strong.

References

1. Pankaj Kamthan, CONTROL FLOW STRUCTURE OF SOURCE CODE.
2. Pankaj Kamthan, INTRODUCTION TO SOFTWARE PROJECT COST ESTIMATION.
3. *Chatgpt.* ChatGPT. (n.d.).) , https://openai.com/chatgpt.html
4. *Command-line usage¶.* Command-line Usage - Radon 4.1.0 documentation. (n.d.), https://radon.readthedocs.io/en/latest/commandline.html