



SOEN 6611 (SOFTWARE MEASUREMENT)

CONCORDIA UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE AND SOFTWARE ENGINEERING

Deliverable 2

Students:

Team N

Bharat Saini (40202642)

Tania Sanjid (40255010)

Hani Saravanan (40233005)

Siva Nagi Reddy (40221868)

Deepanshu Sehgal (40225333)

Supervisor:

Prof. PANKAJ KAMTHAN

November 22, 2023

Contents

List of Symbols and Abbreviations	3
1 About the Project	4
2 Problem 3)	5
2.1 Effort Estimate using Use Case Point approach	5
2.2 Effort Estimate using Basic COCOMO 81 approach	11
2.3 Difference in estimates using the UCP approach and COCOMO 81, and the actual effort towards the project	12
3 Problem 4)	14
4 Problem 5)	17
4.1 Cyclomatic Complexity of each of the functions in main.py	17
4.1.1 Explanation of each function	18
4.2 Cyclomatic Complexity of each of the Methods and class in metricstics.py	18
4.2.1 Explanation of each function	18
4.3 Analysis of main.py	20
4.4 Analysis of metricstics.py	20
4.5 Qualitative Conclusions	21
5 Problem 6)	22
5.1 Weighted Method Per Class (WMC)	22
5.1.1 In Class: main.py	22
5.1.2 In Class: Metricstics.py	23
5.2 Coupling Factor (CF)	24
5.3 Lack of Cohesion in Methods (LCOM*)	24
5.3.1 Main Class:	25
5.3.2 Metricstics Class:	27
5.4 Qualitative Conclusions	29
6 Problem 7 : Physical And Logical SLOC	31
6.1 Physical SLOC	31
6.1.1 Calculation of Physical SLOC METRICSTICS	31
6.2 Logical SLOC	32
6.2.1 Calculation of Logical SLOC METRICSTICS	32
6.3 Analysis of Code Metrics: Interpreting Physical and Logical SLOC	35
7 Problem 8 : Analysis of Correlation between Logical SLOC and WMC in MET- RICSTICS Project	37
7.1 Scatter plot Analysis between Logical SLOC and WMC	37
7.2 Correlation Coefficient Analysis between Logical SLOC and WMC	38
8 Collaboration Environments	40
9 References	41

List of Symbols and Abbreviations

GQM	Goal Question Metric
UC	Use Case
SLOC	Source Line of Codes
SLOC(L)	Logical SLOC
UCP	Use Case Point
PF	Productivity Factor
UUCP	Unadjusted Use Case Points
TCF	Technical Complexity Factor
Ecf	Environmental Complexity Factor
UAW	Unadjusted Actor Weight
UUCW	Unadjusted Use Case Weight
WTi	Technical Complexity Factor Weight
Fi	Perceived Impact Weight
WEi	Environmental Complexity Factor Weight

About The Project

In today's data-driven world, effective data interpretation is crucial for informed decision-making. This led to the development of METRICSTICS, our user-centric data analysis tool. METRICSTICS enhances the interpretability of statistical data by providing contextual insights and tailored recommendations. It caters to a diverse user base, from data analysts to those with limited statistical expertise. METRICSTICS simplifies the understanding of statistical findings, empowering users to make informed decisions. We prioritize user engagement, satisfaction, and time efficiency, aiming to build trust in our system's recommendations. Our goal is to foster a data-driven culture where METRICSTICS becomes a trusted ally, bridging data analysis with real-life scenarios, and offering a competitive advantage across industries.

Problem 3 : Use Case Points(UCP)

2.1 Effort Estimate using Use Case Point approach

Effort Estimation using use case points is given by,

$$EffortEstimate = UCP * PF \quad (2.1)$$

where UCP is Use Case Points, PF is Productivity Factor.
UCP is calculated as:

$$UCP = UUCP * TCF * ECF \quad (2.2)$$

where UUCP is Unadjusted Use Case Points, TCF is Technical Complexity Factor, ECF is Environmental Complexity Factor.

$$UUCP = UAW + UUCW \quad (2.3)$$

UUCP is the sum of Unadjusted Actor Weight (UAW) and Unadjusted Use Case Weight (UUCW).

For our use case model we have considered the below actors:

- **Data Analyst:** A professional who analyzes data and utilizes the METRICSTICS system to perform a variety of statistical functions and gain insights.
- **Application Manager:** An individual responsible for the overall management and maintenance of the METRICSTICS application, ensuring it runs smoothly and efficiently.
- **Business Intelligence Analyst:** A specialist who focuses on generating comprehensive reports from the METRICSTICS system to support business intelligence activities.

We will use the below table to assign a weight to each actor:

Actor Type	Description	Weight
A1	Simple Actor	1
A2	Average Actor	2
A3	Complex Actor	3

Table 2.1: The classification of actors and their associated weights in the UCP approach.

- Data Analyst - 3(Complex Actor)
- Application Manager - 3(Complex Actor)
- Business Intelligence Analyst - 3(Complex Actor)

$$\text{Total UAW} = 9$$

Now to calculate UUCW, we will classify each of the use cases as simple, average and complex.

- **Simple:** All use cases involve significant interaction or processing so no simple use case for our model.
- **Average:** Upload Data, Provide Feedback, Monitor Usage
- **Complex:** Perform Descriptive Statistics, Access Recommendations, Access Contextual Insights, Configure Preferences, Generate Reports, Manage Applications, Update Application.

We will refer the below table to assign weights to each use case based on the below table:

Use Case Type	Description	Weight
UC1	Simple Use Case	5
UC2	Average Use Case	10
UC3	Complex Use Case	15

Table 2.2: The classification of use cases and their associated weights in the UCP approach.

Based on the table we calculate total weights for each category, based on the formula,

No. of use cases(In a particular category) * Weight of that category

- **Simple:** $0 * 5 = 0$
- **Average:** $3 * 10 = 30$
- **Complex:** $7 * 15 = 105$

So, The Total Unadjusted Use Case Weight (UUCW) for the METRIC-STICS project, given the listed use cases, is $0 + 30 + 105 = 135$

$$UUCW = 135$$

Now we will calculate UUCP,

$$\begin{aligned} UUCP &= UAW + UUCW \\ &= 9 + 135 \\ &= 144 \end{aligned}$$

Calculation of TCF for our model:

The aim of TCF is to address technical issues that may affect the software project from its initiation through to its completion, encompassing the delivery phase.

Below is the formula to calculate TCF:

$$TCF = C1 + (C2 * \sum_{i=1}^{13} (WTi * Fi)) \quad (2.4)$$

c1=0.6 and c2=0.01

WTi is Technical Complexity Factor Weight

Fi is Perceived Impact Factor.

We will use the below table to calculate the technical complexity factors for our project:

TCF Type	Description	Weight
T1	Distributed System	2
T2	Performance	1
T3	End User Efficiency	1
T4	Complex Internal Processing	1
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

Table 2.3: The Technical Complexity Factors in the UCP approach.

Now we will classify what influence each TCF type has on our model, The influence can be classified as No influence, Average influence and Strong Influence:

- Distributed System (T1): No influence
- Performance (T2): Average influence

- End User Efficiency (T3): Average influence
- Complex Internal Processing (T4): Average influence
- Reusability (T5): Strong influence
- Easy to Install (T6): Strong influence
- Easy to Use (T7): Strong influence
- Portability (T8): Strong influence
- Easy to Change (T9): Strong influence
- Remaining factors (T10-T13): No influence

Now we will calculate the perceived impact factor based on the below table:

Influence	Percieved impact factor
No influence	0
Average influence	3
Strong influence	5

Table 2.4: The classification of percieved impact factor based on the type of influence.

The below table represents TCF type, Description, Weight, Fi(Impact factor)

TCF Type	Description	Weight	Fi
T1	Distributed System	2	0
T2	Performance	1	3
T3	End User Efficiency	1	3
T4	Complex Internal Processing	1	3
T5	Reusability	1	5
T6	Easy to Install	0.5	5
T7	Easy to Use	0.5	5
T8	Portability	2	5
T9	Easy to Change	1	5
T10	Concurrency	1	0
T11	Special Security Features	1	0
T12	Provides Direct Access for Third Parties	1	0
T13	Special User Training Facilities are Required	1	0

Table 2.5: Table depicting TCF type, Description, Weight and Fi based on our model.

Now putting these values into the TCF formula and calculating TCF below:

$$TCF = 0.6 + (0.01 * ((2 * 0) + (1 * 3) + (1 * 3) + (1 * 3) + (1 * 5) + (0.5 * 5) + (0.5 * 5) + (2 * 5) + (1 * 5) + (1 * 0) + (1 * 0) + (1 * 0) + (1 * 0)))$$

$$TCF = 0.94$$

Calculation of Environmental Complexity Factors(ECF):

The objective of ECF is to consider the personal characteristics of the development team, such as experience. Generally, the greater the experience of the development team, the larger the influence of ECF on UCP.

ECF is calculated using below formula:

$$ECF = C1 + (C2 * \sum_{i=1}^8 (WEi * Fi)) \quad (2.5)$$

c1 = 1.4 and c2 = -0.03

WEi is the Environmental Complexity Factor Weight

Fi is Perceived Impact Factor.

We will use the below table to calculate the environment complexity factors for our project:

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

Table 2.6: The Environmental Complexity Factors in the UCP approach.

Now we will classify what influence each ECF type has on our model, The influence can be classified as No influence, Average influence and Strong positive influence and Strong negative influence:

- Familiarity with Use Case Domain (E1): Average influence
- Part-Time Workers (E2): No influence
- Analyst Capability (E3): Strong, positive influence
- Application Experience (E4): Strong, positive influence
- Object-Oriented Experience (E5): Strong, positive influence
- Motivation (E6): Strong, positive influence
- Difficult Programming Language (E7): Strong, negative influence

- Stable Requirements (E8): Strong, positive influence

Now we will calculate the perceived impact factor based on the below table:

Influence	Percieved impact factor
No influence	0
Strong positive influence	1
Average influence	3
Strong positive influence	5

Table 2.7: The classification of percieved impact factor based on the type of influence. approach.

The below table represents ECF type, Description, Weight, Fi(Impact factor)

ECF Type	Description	Weight	Fi
E1	Familiarity with Use Case Domain	1.5	3
E2	Part-Time Workers	-1	0
E3	Analyst Capability	0.5	5
E4	Application Experience	0.5	5
E5	Object-Oriented Experience	1	5
E6	Motivation	1	5
E7	Difficult Programming Language	-1	1
E8	Stable Requirements	2	5

Table 2.8: Table depicting ECF type, Description, Weight and Fi based on our model.

Now putting these values into the ECF formula and calculating ECF below:

$$ECF = 1.4 + (-0.03 * (1.5 * 3 + -1 * 0 + 0.5 * 5 + 0.5 * 5 + 1 * 5 + 1 * 5 + -1 * 1 + 2 * 5))$$

$$ECF = 0.545$$

Now we will calculate UCP based on 2.2, 2.3, 2.4 and 2.5

$$UCP = UUCP * TCF * ECFUCP = 144 * 0.94 * 0.545 \quad (2.6)$$

$$UCP = 73.77$$

Now based on 2.1, we will estimate the effort. To estimate the effort, we assume the PF to be 20.

$$\begin{aligned} \text{Effort Estimate} &= UCP * PF \\ &= 73.77 * 20 \\ &= 1475.424 \text{ Person-Hours} \end{aligned}$$

2.2 Effort Estimate using Basic COCOMO 81 approach

The Basic COCOMO (Constructive Cost Model) 81 is a procedural software cost estimation model developed by Barry W. Boehm. The model is used to estimate the software development effort based on the size of the program, which is measured in thousands of lines of code (KLOC).

The model's effort estimation formula for an "organic" type project is:

$$E = a * (KLOC^b) \quad (2.7)$$

where:

E is the effort applied in person-months

KLOC is the estimated number of delivered lines of code expressed in thousands

a and **b** are constants that depend on the project type

So, the lines of source code for our project(LOC) is 400 Now for basic COCOMO 81 approach, there are 3 types of projects based on the modes of development and complexity namely Organic, Semi-Detached and Embedded.

Our project comes under Organic project so, $a = 2.4$ $b = 1.05$ (predefined constants for organic projects)

KLOC is calculated as: $LOC/1000$

$$\begin{aligned} &= 400/1000 \\ \mathbf{KL0C} &= 0.4 \end{aligned}$$

Now Substituting these values in the above formula for Estimation effort using Basic COCOMO 81:

$$\begin{aligned} E &= 2.4 \times (0.4^{1.05}) \\ E &= 0.917 \text{ person months} \end{aligned}$$

So, the estimated effort required for our project would be approximately 0.917 Person-Months

2.3 Difference in estimates using the UCP approach and COCOMO 81, and the actual effort towards the project

The difference in estimates obtained using the UCP (Use Case Points) approach and the COCOMO 81 (Constructive Cost Model 1981) approach, as well as the actual effort towards the project, can be analyzed as follows:

Estimate using UCP Approach: 1475.424 Person-Hours

- The UCP estimate of 1475.424 Person-Hours suggests a detailed analysis of the system's functionality and user interactions.
- The UCP approach bases its estimation on the analysis of use cases which represent the interaction between users and the system, hence its consideration for user interface design complexity.
- Technical factors such as distributed systems, performance requirements, and end-user efficiency are taken into account, which significantly impact the estimated effort.
- Environmental factors including team experience, motivation, and working conditions are also integrated into the calculation, providing a comprehensive view of the project landscape.

Estimate using Basic COCOMO 81 Approach: 0.917 Person-Months

- COCOMO 81 uses a formulaic approach based on project size and type, estimating effort in Person-Months.
- Assuming a standard 152-hour work month, the COCOMO 81 estimate translates to approximately 139.38 Person-Hours ($0.917 \text{ Person-Months} \times 152 \text{ hours/month}$)

Difference in Estimates:

- The UCP estimate is higher at 1475.424 Person-Hours, reflecting a detailed assessment of user interactions and system complexities.
- COCOMO 81 suggests a lower effort of roughly 139.38 Person-Hours, focusing mainly on code size and project complexity.
- The significant difference in estimates may indicate that the UCP approach accounted for more factors or complexities that the COCOMO 81 approach did not consider, such as the influence of technical and environmental factors specific to the project.

- The UCP estimate suggests that there may be a greater number of use cases or a higher degree of complexity within those use cases, which could require more effort than the size-based estimate provided by COCOMO 81.
- The difference could also reflect the UCP's incorporation of organizational and environmental factors such as team experience and motivation, which can have a substantial impact on effort but might be less emphasized or captured in the COCOMO model.
- COCOMO 81 may yield lower effort estimates, as it could both undervalue the efficiencies of modern development tools and maintain an optimistic view of productivity, potentially overlooking complexities accounted for in the UCP analysis.

Actual Effort Towards the Project:

- The actual effort can be influenced by real-time decision-making, crisis management, and the agility of the team to respond to unexpected changes or requirements.
- Differences between estimated and actual efforts may arise from scope changes, requirement evolution, and stakeholder engagement which are not always predictable.
- Projects can also experience variance in effort due to technical debt, code quality issues, and integration challenges which are typically not accounted for in initial estimates

Problem 4 : METRICSTICS Implementation

Our METRICSTICS project is a testament to the robustness and flexibility of Python, especially when combined with object-oriented programming principles. The project is structured into two main components: `main.py`, which orchestrates user interaction and data handling, and `metricstics.py`, which encapsulates the core statistical functionality.

Graphical User Interface (GUI):

- Developed using `tkinter`, the GUI in `main.py` offers an intuitive platform for users to interact with METRICSTICS. It includes features for data generation, data upload, and displaying calculated metrics.
- The `ScrolledText` widget elegantly presents the randomly generated or uploaded data, enhancing user experience and data visibility.
- Dedicated buttons for each statistical metric allow users to perform individual calculations, adding to the application's interactivity and ease of use.

Custom Implementation of Statistical Functions:

- The `Metricstics` class in `metricstics.py` forms the core of our application, offering essential statistical functions such as mean, median, mode, and standard deviation.
- These functions are uniquely implemented from scratch, bypassing Python's built-in utilities and external libraries. Notable implementations include sorting algorithms, frequency calculations, and square root computation via Newton's method.
- Designed with OOP principles, the class features public methods for statistical analysis and private methods for internal processes, ensuring data encapsulation and method abstraction.

Data Handling and Validation:

- In main.py, robust data handling mechanisms ensure that only valid numeric data is processed during file uploads. This includes parsing data from GUI input or file uploads, with comprehensive error checking and user-friendly error messages
- The Metricstics class validates and sorts the input data upon initialization, ensuring accurate statistical computations.

Testing with Large Data Sets:

- To thoroughly test METRICSTICS, we generated a dataset with over 1000 values randomly distributed between 0 and 1000. This extensive dataset ensures that our statistical functions are reliable and efficient even with large data sets.
- We conducted various tests to verify each statistical measure, comparing our application's output with manual calculations to ensure accuracy.

Accessing and Running METRICSTICS:

Our METRICSTICS application is readily available on GitHub, allowing users to easily download, review, and contribute to the code. To access our project, visit the GitHub repository at [SOEN-6611-TEAM-N-Code](#).

Below are the steps to run metricstics:

- **Clone the Repository:** First, clone the repository from GitHub to your local machine using git clone.
- **Install Dependencies:** Ensure Python is installed on your system. There are no external dependencies required for this project, as it solely uses Python's standard library.
- **Launch the Application:** Run main.py using Python to start the METRICSTICS application. This can be done through a command line or an IDE by navigating to the project directory and executing python main.py.
- **Use the Application:** Once launched, the application's GUI will guide you through generating data, performing statistical calculations, and managing session data

Below are some screenshots of our METRICSTICS project:

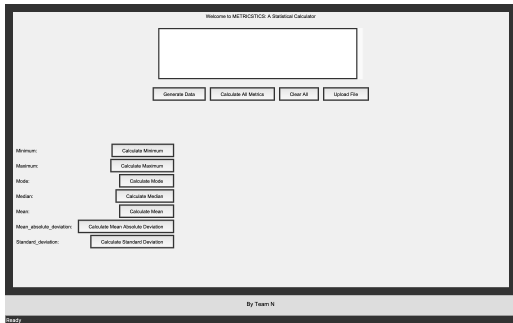


Figure 3.1: Metrics GUI

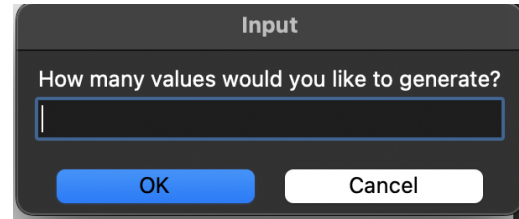


Figure 3.2: Random Data Generator

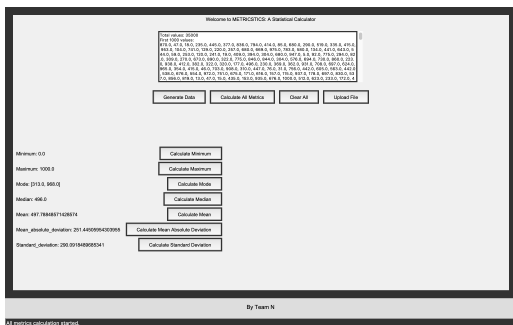


Figure 3.3: Evidence of Test-1

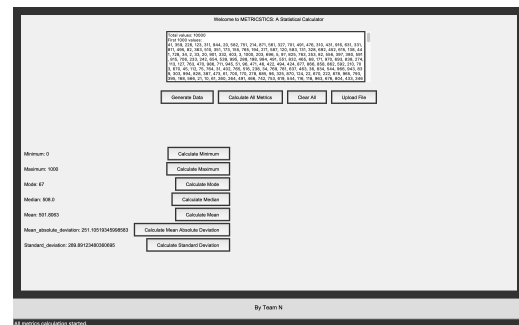


Figure 3.4: Evidence of Test-2

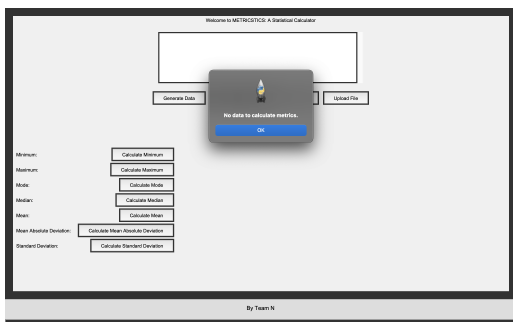


Figure 3.5: Exception Handling: No Data Generated

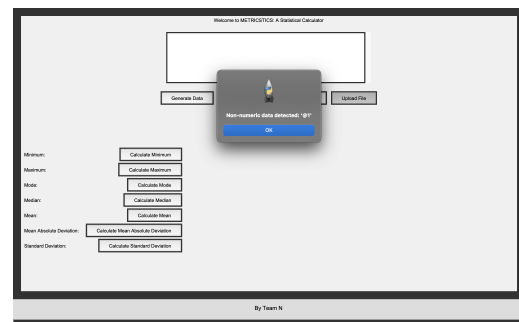


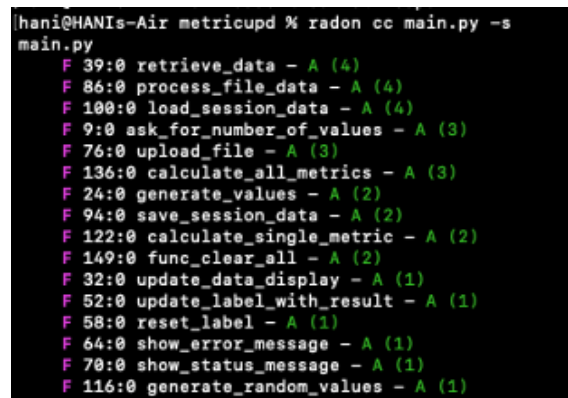
Figure 3.6: Exception Handling: Numeric Value Only

Problem 5 : Cyclomatic Number

We have two main Python files: `main.py` and `metricstics.py`, each serving different functionalities. In our analysis, we employ the Radon tool to calculate the cyclomatic number, which provides insights into the complexity of our codebase. Here are the key aspects of how we utilize Radon for this purpose:

- We utilize the Radon tool's `cc` command to compute the Cyclomatic Complexity.
- Here, 'F' typically denotes a function, 'M' signifies a method, and 'C' stands for a class.
- We use the `-a` option with Radon which instructs Radon to calculate the average complexity.
- When assessing code complexity, we pay particular attention to the complexity rank of 'C' or worse. In this context, 'A' represents average complexity.

4.1 Cyclomatic Complexity of each of the functions in `main.py`



```
hani@HANIs-Air metricupd % radon cc main.py -s
main.py
F 39:0 retrieve_data - A (4)
F 86:0 process_file_data - A (4)
F 100:0 load_session_data - A (4)
F 9:0 ask_for_number_of_values - A (3)
F 76:0 upload_file - A (3)
F 136:0 calculate_all_metrics - A (3)
F 24:0 generate_values - A (2)
F 94:0 save_session_data - A (2)
F 122:0 calculate_single_metric - A (2)
F 149:0 func_clear_all - A (2)
F 32:0 update_data_display - A (1)
F 52:0 update_label_with_result - A (1)
F 58:0 reset_label - A (1)
F 64:0 show_error_message - A (1)
F 70:0 show_status_message - A (1)
F 116:0 generate_random_values - A (1)
```

Figure 4.1: Cyclomatic Complexity of `main.py`

4.1.1 Explanation of each function

- `retrieve_data` (CC: 4): Function reads data from the widget, checks if the data is empty, attempts to split and convert the data to integers, and raises a `ValueError` if the data is not numeric.
- `process_file_data` (CC: 4): Function Reads data from the file, splits the data by commas, converts data to integers, handles errors, and raises exceptions.
- `load_session_data` (CC: 4): Function loads session data from a file and updates the UI.
- `ask_for_number_of_values` (CC: 3): Function prompts the user to enter the number of random values to generate.
- `upload_file` (CC: 3): Function allows the user to upload a file and process its data.
- `calculate_all_metrics` (CC: 3): Function calculates all predefined metrics for the data.
- `generate_values` (CC: 2): Function generates random values based on user input.
- `save_session_data` (CC: 2): Function saves session data to a file.
- `calculate_single_metric` (CC: 2): Function calculates a single metric and updates its label.
- `func_clear_all` (CC: 2): Function clears all data and resets labels to their default text.
- `Other functions` (CC: 1): Functions with a cyclomatic complexity of 1 have a single, straightforward path through the code and do not involve branching or conditional statements.

4.2 Cyclomatic Complexity of each of the Methods and class in `metricstics.py`

4.2.1 Explanation of each function

- `Metricstics.__init__` (CC: 1): Constructor initializes a `Metricstics` object, calling `_initialize_data` for data validation and sorting.
- `Metricstics._initialize_data` (CC: 1): Method is responsible for validating and sorting the data upon initialization.

```

hani@HANIs-Air metricupd % radon cc metricstics.py -s
metricstics.py
M 20:4 Metricstics._validate_data - A (4)
M 31:4 Metricstics._sort_data - A (4)
M 74:4 Metricstics.mode - A (4)
C 1:0 Metricstics - A (3)
M 118:4 Metricstics._sqrt - A (3)
M 63:4 Metricstics.median - A (2)
M 84:4 Metricstics.mean_absolute_deviation - A (2)
M 92:4 Metricstics.standard_deviation - A (2)
M 100:4 Metricstics._manual_sum - A (2)
M 109:4 Metricstics._calculate_frequency - A (2)
M 7:4 Metricstics.__init__ - A (1)
M 13:4 Metricstics._initialize_data - A (1)
M 44:4 Metricstics.minimum - A (1)
M 50:4 Metricstics.maximum - A (1)
M 56:4 Metricstics.mean - A (1)
M 130:4 Metricstics._dummy_complex_operation - A (1)
M 137:4 Metricstics.future_feature_placeholder - A (1)

```

Figure 4.2: Cyclomatic Complexity of metricstics.py

- `Metricstics._validate_data` (CC: 4): Private method validates the data to ensure it is not empty and contains only numeric values.
- `Metricstics._sort_data` (CC: 4): Private method sorts the data using the insertion sort algorithm.
- `Metricstics.minimum` (CC: 1): Method returns the minimum value in the sorted data.
- `Metricstics.maximum` (CC: 1): Method returns the maximum value in the sorted data.
- `Metricstics.mean` (CC: 1): Method calculates and returns the mean (average) of the data.
- `Metricstics.median` (CC: 2): Method calculates and returns the median of the data.
- `Metricstics.mode` (CC: 4): Method calculates and returns the mode(s) of the data.
- `Metricstics.mean_absolute_deviation` (CC: 2): Method calculates and returns the mean absolute deviation of the data.
- `Metricstics.standard_deviation` (CC: 2): Method calculates and returns the standard deviation of the data.
- `Metricstics._manual_sum` (CC: 2): Private method manually sums up an iterable of numbers.
- `Metricstics._calculate_frequency` (CC: 2): Private method calculates the frequency of each value in the data and returns a dictionary.

- `Metricstics._sqrt` (CC: 3): Private method calculates and returns the square root of a value using Newton's method.
- `_dummy_complex_operation` and `future_feature_placeholder` (CC: 1): These are placeholder methods with a cyclomatic complexity of 1 each, as they do not contain branching logic.

4.3 Analysis of main.py

```
hani@HANIs-Air metricupd % radon cc main.py -a
main.py
F 39:0 retrieve_data - A
F 86:0 process_file_data - A
F 100:0 load_session_data - A
F 9:0 ask_for_number_of_values - A
F 76:0 upload_file - A
F 136:0 calculate_all_metrics - A
F 24:0 generate_values - A
F 94:0 save_session_data - A
F 122:0 calculate_single_metric - A
F 149:0 func_clear_all - A
F 32:0 update_data_display - A
F 52:0 update_label_with_result - A
F 58:0 reset_label - A
F 64:0 show_error_message - A
F 70:0 show_status_message - A
F 116:0 generate_random_values - A

16 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.1875)
```

Figure 4.3: Average Complexity of main.py

- Average Complexity: A (2.1875)
- The functions in main.py mostly have a complexity rating of A, with values ranging from 1 to 4.
- The majority of functions have low complexity (1 to 3), which suggests that the code in main.py is relatively straightforward and easy to maintain.
- Functions with a complexity of 4, such as `retrieve_data`, `process_file_data`, and `load_session_data`, are more complex but still within a manageable range.

4.4 Analysis of metricstics.py

- Average Complexity: A (2.0588235294117645)
- Similar to main.py, metricstics.py also has functions primarily rated as A, with complexity values ranging from 1 to 4.
- The distribution of complexity scores is slightly more even here, with a good mix of functions at different levels of complexity.

```

hani@HANIs-Air metricupd % radon cc metricstics.py -a
metricstics.py
M 20:4 Metricstics._validate_data - A
M 31:4 Metricstics._sort_data - A
M 74:4 Metricstics.mode - A
C 1:0 Metricstics - A
M 118:4 Metricstics._sqrt - A
M 63:4 Metricstics.median - A
M 84:4 Metricstics.mean_absolute_deviation - A
M 92:4 Metricstics.standard_deviation - A
M 100:4 Metricstics._manual_sum - A
M 109:4 Metricstics._calculate_frequency - A
M 7:4 Metricstics.__init__ - A
M 13:4 Metricstics._initialize_data - A
M 44:4 Metricstics.minimum - A
M 50:4 Metricstics.maximum - A
M 56:4 Metricstics.mean - A
M 130:4 Metricstics._dummy_complex_operation - A
M 137:4 Metricstics.future_feature_placeholder - A

17 blocks (classes, functions, methods) analyzed.
Average complexity: A (2.8588235294117645)

```

Figure 4.4: Average Complexity of metricstics.py

- Functions like `_validate_data`, `_sort_data`, and `mode` are at the higher end of the complexity scale, suggesting more intricate logic or control structures.

4.5 Qualitative Conclusions

Maintainability: Both `main.py` and `metricstics.py` show a predominance of functions with low to moderate complexity. This indicates good maintainability and suggests that the code is relatively easy to understand, test, and modify.

Code Quality: The average complexity scores being close to 2 for both files imply that the code quality is likely good, with well-structured functions that avoid excessive branching or deep nesting.

Testing: Lower complexity generally means easier testing. The functions in these files are straightforward to cover with unit tests, ensuring more reliable code.

Problem 6 : WMC, CF, LCOM*

5.1 Weighted Method Per Class (WMC)

WMC is a metric used to quantify the complexity and size of a software module, often a class in object-oriented programming. WMC is calculated using the complexity weights of the methods within a class. For this calculation, non-normalized weights are assumed. The formula for WMC is given by:

$$WMC = \sum_{i=1}^n ci(Mi)$$

Where:

- WMC is the Weighted Method Per Class.
- $ci(Mi)$ is the complexity (weight) of the i th method in the class.
- n is the total number of methods in the class.

5.1.1 In Class: main.py

- Method: `retrieve_data`, CC: 4
- Method: `process_file_data`, CC: 4
- Method: `load_session_data`, CC: 4
- Method: `ask_for_number_of_values`, CC: 3
- Method: `upload_file`, CC: 3
- Method: `calculate_all_metrics`, CC: 3
- Method: `generate_values`, CC: 2
- Method: `save_session_data`, CC: 2
- Method: `calculate_single_metric`, CC: 2

- Method: `func_clear_all`, CC: 2
- Method: `update_data_display`, CC: 1
- Method: `update_label_with_result`, CC: 1
- Method: `reset_label`, CC: 1
- Method: `show_error_message`, CC: 1
- Method: `show_status_message`, CC: 1
- Method: `generate_random_values`, CC: 1

$$WMC = 4 + 4 + 4 + 3 + 3 + 3 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 = 35$$

5.1.2 In Class: `Metricstics.py`

- Method: `_validate_data`, CC: 4
- Method: `_sort_data`, CC: 4
- Method: `mode`, CC: 4
- Method: `_sqrt`, CC: 3
- Method: `median`, CC: 2
- Method: `mean_absolute_deviation`, CC: 2
- Method: `standard_deviation`, CC: 2
- Method: `_manual_sum`, CC: 2
- Method: `_calculate_frequency`, CC: 2
- Method: `__init__`, CC: 1
- Method: `_initialize_data`, CC: 1
- Method: `minimum`, CC: 1
- Method: `maximum`, CC: 1
- Method: `mean`, CC: 1
- Method: `_dummy_complex_operation`, CC: 1
- Method: `future_feature_placeholder`, CC: 1

$$WMC = 4 + 4 + 4 + 3 + 2 + 2 + 2 + 2 + 2 + 1 + 1 + 1 + 1 + 1 + 1 + 1 = 32$$

5.2 Coupling Factor (CF)

The Coupling Factor (CF) is a metric used to quantify the level of coupling between a class and other classes in an object-oriented system. The formula for CF is given by:

$$CF = \sum_{i=1}^n \sum_{j=1}^n (\text{IsClient}(C_i, C_j)) \cdot (n^2 - n)$$

CF for Metricstics and Main Classes:

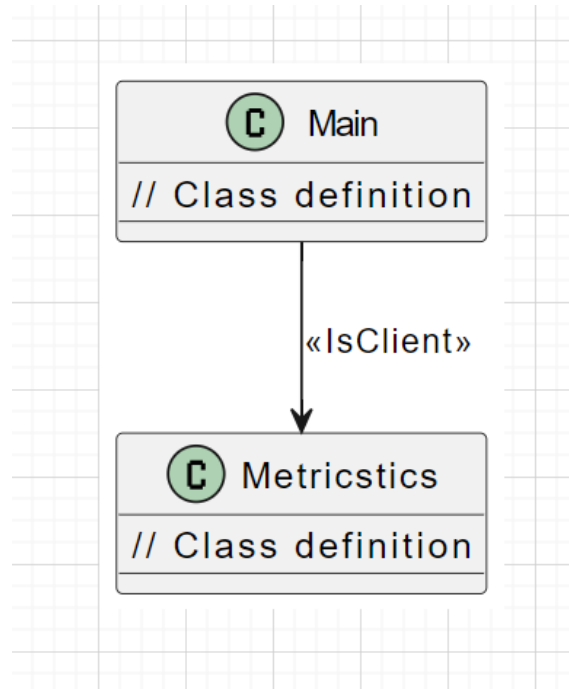


Figure 5.1: Coupling Factor

By Fig 6.1, $IsClient(\text{Main}, \text{Metricstics}) = 1$

$$n = 2 \quad (\text{Number of Classes})$$

$$CF = \frac{\sum (\sum IsClient(\text{Main}, \text{Metricstics}))}{2^2 - 2} = \frac{1}{2}$$

5.3 Lack of Cohesion in Methods (LCOM*)

The Lack of Cohesion in Methods (LCOM*) is a metric used to measure the cohesiveness of a class by determining the number of attributes common to the methods of that class. The formula for LCOM* is given as:

$$LCOM_* = \frac{\frac{1}{a} \sum_{i=1}^a \mu(A_i) - m}{1 - m}$$

Where:

- m is the number of methods in the class.
- a is the number of attributes accessed by the methods.
- $\mu(A_i)$ is the number of methods that access attribute A_i .

5.3.1 Main Class:

To calculate the Lack of Cohesion in Methods (LCOM*) for the given class `main`, you'll need to analyze the methods and attributes within the class. From the provided code, it appears that `main` is more of a script or the main part of the application rather than a class. However, I'll assume you are treating it as a class for the sake of the LCOM* calculation.

Here's a breakdown of the LCOM* formula for the class `main`:

m : the number of methods in the class.

a : the number of attributes accessed by the methods.

$\mu(A_i)$: the number of methods that access attribute A_i .

First, let's identify the methods and attributes in the class:

Methods (m):

1. `ask_for_number_of_values`
2. `generate_values`
3. `update_data_display`
4. `retrieve_data`
5. `update_label_with_result`
6. `reset_label`
7. `show_error_message`
8. `show_status_message`
9. `upload_file`
10. `process_file_data`

11. `save_session_data`
12. `load_session_data`
13. `generate_random_values`
14. `calculate_single_metric`
15. `calculate_all_metrics`
16. `func_clear_all`

Attributes (a):

1. `numeric_val`
2. `val`
3. `file_path`
4. `file_data`
5. `data_str`
6. `data`
7. `metrics`
8. `result`
9. `lines`

To calculate $\mu(A_i)$ for each attribute:

$\mu(\text{numeric_val})$: Appears to be accessed only in the method `ask_for_number_of_v`

$\mu(\text{val})$: Appears to be accessed only in the method `generate_values`.

$\mu(\text{file_path})$: Appears to be accessed only in the method `upload_file`.

$\mu(\text{file_data})$: Appears to be accessed only in the method `process_file_data`.

$\mu(\text{data_str})$: Appears to be accessed in the methods `retrieve_data` and `proce`

$\mu(\text{data})$: Appears to be accessed in multiple methods.

$\mu(\text{metrics})$: Appears to be accessed in the methods `save_session_data` and `l`

$\mu(\text{result})$: Appears to be accessed in multiple methods.

$\mu(\text{lines})$: Appears to be accessed in the method `load_session_data`.

Now, let's substitute these values into the LCOM* formula:

$$LCOM^* = \frac{\frac{1}{a} \sum_{i=1}^a \mu(A_i) - m}{1 - m}$$

$$LCOM^* = \frac{\frac{1}{9} (1 + 1 + 1 + 2 + 2 + 4 + 2 + 2 + 1) - 16}{1 - 16}$$

$$LCOM^* = \frac{\frac{16}{9} - 16}{-15}$$

$$LCOM^* = \frac{-14.2222}{-15}$$

$$LCOM^* \approx 0.9481$$

The result of .95 is quite high, approaching 1.0. This suggests low cohesion within the main class, as a substantial portion of methods access common attributes. In the context of LCOM*, a high value like .95 indicates that there is a significant lack of cohesion in the Main class, which may make the class more difficult to understand, maintain, and modify.

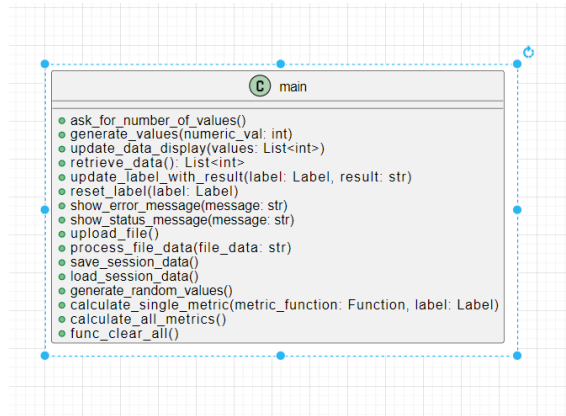


Figure 5.2: LCOM* for main

5.3.2 Metricstics Class:

In the Metricstics class, it has the following methods:

1. `__init`
2. `__initialize_data`
3. `__validate_data`
4. `__sort_data`
5. `minimum`
6. `maximum`
7. `mean`

8. median
9. mode
10. mean_absolute_deviation
11. standard_deviation
12. _manual_sum
13. _calculate_frequency
14. _sqrt
15. _dummy_complex_operation
16. future_feature_placeholder

And the following attribute:

1. data

To calculate $\mu(A_i)$ for each attribute:

1. $\mu(data)$: This attribute is accessed in all methods, so $\mu(data) = 16$.

After substituting these values into the LCOM* formula:

$$LCOM^* = \frac{\frac{1}{1} \cdot 16 - 16}{1 - 16} = \frac{0}{-15} = 0$$

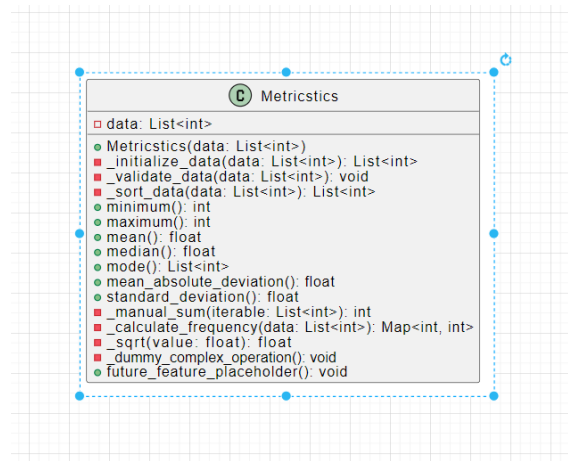


Figure 5.3: LCOM* Metricstics

The result is 0. The interpretation of LCOM* is that a lower value is better, indicating higher cohesion. In this case, since LCOM* is 0, it suggests that all methods in the class are accessing the same attribute, which is a positive sign for cohesion. However, it's essential to consider this metric along with other factors and domain knowledge for a comprehensive evaluation of the class design.

5.4 Qualitative Conclusions

Here are the qualitative conclusions for the object-oriented metrics (WMC, CF, and LCOM*) based on their respective quantitative thresholds:

1. Weighted Method Per Class (WMC):

Main Class (main.py):

- $WMC = 35$
- Thresholds: No universal thresholds, but higher values indicate higher complexity.
- Qualitative Conclusion: Moderate complexity in the main class.

Metricstics Class (Metricstics.py):

- $WMC = 32$
- Thresholds: No universal thresholds, but higher values indicate higher complexity.
- Qualitative Conclusion: Moderate complexity in the main class.

2. Coupling Factor (CF):

CF for Main and Metricstics Classes:

- $CF = 1$
- Thresholds: CF ranges from 0 to 1; lower values indicate better encapsulation and lower coupling.
- Qualitative Conclusion: High coupling between the Main and Metricstics classes.

3. Lack of Cohesion in Methods (LCOM*):

Main Class

$$LCOM^* = 0.95$$

LCOM* value of .95 for the main class suggests low cohesion, and it would be advisable to review and refactor the class to bring the LCOM* value within the desired threshold for better maintainability and understandability of the code.

Metricstics Class

$$\text{LCOM}^* = 0$$

A perfect score of 0 indicates excellent cohesion in the Metricstics class. All methods in the class access the same attribute, contributing to high cohesion.

Overall Conclusion:

- **WMC:** The WMC values for both 'main.py' class is 35 and 'Metricstics.py' classes is 32. While this falls within an acceptable range, it's important to review the class structure, ensure that methods are appropriately organized, and consider refactoring if there are opportunities to improve code maintainability.
- **CF:** A low coupling factor is good for maintainability and flexibility. The value of $\frac{1}{2}$ suggests that changes in one class are less likely to impact the other significantly.
- **LCOM*:**
 - For the Main class, the LCOM* value is reasonable (0.95). It is considered high and should be investigated for possible refactoring to improve the design and maintainability of the code.
 - For the Metricstics class, achieving a perfect LCOM* of 0 is excellent. It signifies strong cohesion and suggests that the class is well-designed in terms of method-attribute relationships.

Problem 7 : Physical And Logical SLOC

6.1 Physical SLOC

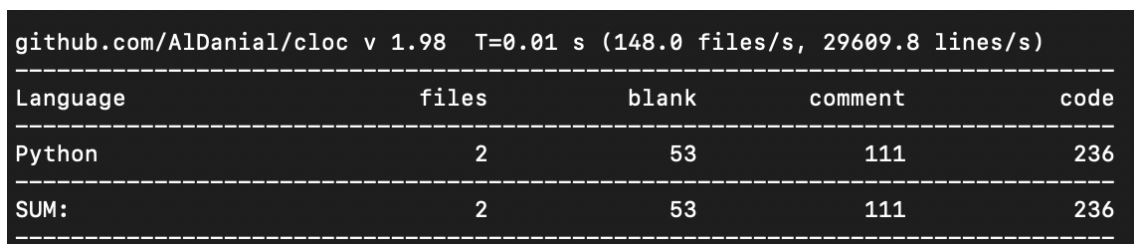
Physical Source Lines of Code (SLOC) offers a direct method for measuring the size of a software program by calculating all the lines present in the source code. It encompasses every element within the source files:

- **Code Lines:** These are lines that include actual programming code, irrespective of how long or complex they are.
- **Comment Lines:** Lines with comments, crucial for clarifying the code's purpose and aiding in its upkeep and understanding.
- **Blank Lines:** Empty lines used to segment code into readable sections or logical groups.

6.1.1 Calculation of Physical SLOC METRICSTICS

To calculate the Physical (SLOC) for the METRICSTICS project, we employed the cloc (Count Lines of Code) tool. cloc is a popular, open-source tool that analyzes the source code files of a project and counts the number of lines of code, distinguishing between code, comments, and blank lines. This tool is widely used due to its accuracy and ability to handle multiple programming languages

Below is the result from cloc tool for our metricstics project:



github.com/AlDanial/cloc v 1.98 T=0.01 s (148.0 files/s, 29609.8 lines/s)				
Language	files	blank	comment	code
Python	2	53	111	236
SUM:	2	53	111	236

Figure 6.1: Physical SLOC

Based on the above results, the calculations are as follows:
Physical SLOC: This includes all lines (code, blank, and comments).

Therefore,

Physical SLOC = code + blank + comment = 236 (code) + 53 (blank) + 111 (comment) = 400 lines.

Advantages of Physical SLOC Physical SLOC quantifies the METRICSTICS project's scale, encompassing documentation crucial for upkeep and updates. It informs broader project management aspects like budgeting and scheduling. A larger Physical SLOC typically denotes extensive documentation, enhancing code clarity and easing teamwork and developer onboarding.

Inference: A Physical SLOC of 400 for the METRICSTICS project suggests a sizeable and well-documented codebase, implying thoroughness in coding and maintenance, and potentially greater ease in managing project scalability and team collaboration

6.2 Logical SLOC

It focuses on the functionality of the code and counts only the lines that contribute directly to the actions or operations that the software performs:

- **Executable Statements:** Lines of code that perform an action, such as calculations, data processing, or calling functions.
- **Control Structures:** Lines that control the flow of the program, like loops and conditional statements, are also counted as they dictate the execution path.

6.2.1 Calculation of Logical SLOC METRICSTICS

To calculate the Logical Source Lines of Code (SLOC) for the METRICSTICS project, which consists of two Python files `main.py` and `metricstics.py`, we used a manual analysis approach. This method involved carefully examining each line of code in both files to determine if it contributes to the functionality of the program. Here's the breakdown of the process and results:

Counting Scheme

- **Logical SLOC Definition:** It includes counting only those lines that contribute directly to the program's functionality. This mainly includes executable statements, control structures, and declarations of functions and classes. Exclude comments, blank lines, and non-functional syntax

- **File Analysis:** In the analysis of each file, We segmented the code into distinct sections, including import statements, class and method declarations, and the core logic within functions. For each of these sections, we meticulously counted every line that met the criteria for Logical SLOC, ensuring a thorough and accurate representation of the code's functional components.

Calculation of Logical SLOC For main.py

- **Import Statements:** 7 SLOCs
 - import tkinter as tk
 - from tkinter import simpledialog, messagebox, filedialog
 - from tkinter.scrolledtext import ScrolledText
 - from metricstics import Metricstics
 - import random
 - import threading
 - from concurrent.futures import ThreadPoolExecutor
- **Function Declarations:** 16 SLOCs
 - def ask_for_number_of_values(): 1 SLOC
 - def generate_values(numeric_val): 1 SLOC
 - def update_data_display(values): 1 SLOC
 - def retrieve_data(): 1 SLOC
 - def update_label_with_result(label, result): 1 SLOC
 - def reset_label(label): 1 SLOC
 - def show_error_message(message): 1 SLOC
 - def show_status_message(message): 1 SLOC
 - def upload_file(): 1 SLOC
 - def process_file_data(file_data): 1 SLOC
 - def save_session_data(): 1 SLOC
 - def load_session_data(): 1 SLOC
 - def generate_random_values(): 1 SLOC
 - def calculate_single_metric(metric_function, label):
1 SLOC
 - def calculate_all_metrics(): 1 SLOC
 - def func_clear_all(): 1 SLOC

- **Core Functionality within Functions:** 72 SLOCs
 - This includes the executable code within each function, like try-except blocks, conditional statements, loops, data manipulations, and function calls.
- **GUI Setup and Layout:** 53 SLOCs
 - Creation of the root window and setting its properties: 3 SLOCs
 - Styling definitions for buttons, labels, and data display: 3 SLOCs
 - Setup of main frame, heading label, and data display: 4 SLOCs
 - Creation and configuration of buttons frame and action buttons: 8 SLOCs
 - Setup of metrics frame and dynamic creation of labels and buttons for metrics: 16 SLOCs
 - Status bar and footer setup: 6 SLOCs
 - Loading of session data and main loop initiation: 2 SLOCs
 - Event bindings and finalization: 11 SLOCs

So, Logical SLOC for main.py = 148 SLOCs

Calculation of Logical SLOC For metricstics.py

- **Class declaration:** 1 SLOC
- **Import statements:** 0 SLOCs
- **__init__ method:** 2 SLOCs
 - Initializer method definition and data initialization call
- **_initialize_data method:** 3 SLOCs
 - Method definition, data validation call, and data sorting call
- **_validate_data method:** 6 SLOCs
 - Method definition, data presence check, type validation loop
- **_sort_data method:** 10 SLOCs
 - Method definition, sorting algorithm implementation
- **Statistical methods:** 33 SLOCs
 - minimum, maximum, mean, median, mode, mean_absolute_deviation, standard_deviation methods

- **Helper methods:** 14 SLOCs
 - `_manual_sum`, `_calculate_frequency`, `_sqrt` methods
- **Dummy methods for future features:** 2 SLOCs
 - `_dummy_complex_operation`, `future_feature_placeholder` methods

So, Logical SLOC for `metricstics.py` = 69 SLOCs

Now we will calculate the Total Logical SLOC for entire `metricstics` project which include both the files,

$$\begin{aligned}
 \text{Total Logical SLOC(Metricstics)} &= \text{Logical SLOC(main.py)} + \\
 &\quad \text{Logical SLOC(metricstics.py)} \\
 &= 148 + 69 \\
 &= 217
 \end{aligned}$$

Advantages of Logical SLOC: Logical SLOC accurately reflects the project’s core functionality, aiding in evaluating complexity and the necessary development effort. It’s vital for resource estimation and coding efficiency assessment, offering a precise view of the project’s functional depth, unaffected by coding style choices.

Inference: A Logical SLOC of 217 for the METRICSTICS project indicates a moderate level of complexity and a well-structured codebase. This metric is useful for estimating the effort and resources required for maintenance and further development. It suggests efficient coding practices and potential ease of manageability.

6.3 Analysis of Code Metrics: Interpreting Physical and Logical SLOC

With a Logical SLOC of 217 and a Physical SLOC of 400 in the METRICSTICS project, below are the several qualitative conclusions:

- **Code Documentation and Readability:** The significant difference between Physical and Logical SLOC suggests that the codebase is well-documented, with ample comments and possibly well-structured whitespace, which can enhance readability and maintainability.
- **Efficiency of Code:** The project appears to maintain a lean approach to writing functional code, as indicated by the Logical SLOC.

This suggests that the functional aspects of the code are concise and potentially more efficient.

- **Potential for Refactoring:** The comparison hints at potential areas where the code could be made more concise. Blocks of code that are less dense in terms of functionality (lower Logical SLOC) may be candidates for refactoring to improve efficiency.
- **Project Estimations:** A higher Physical SLOC points to a larger codebase affecting project timelines for testing and development. Yet, Logical SLOC offers a truer gauge of the effort needed for the code's essential features
- **Scalability:** The lower Logical SLOC hints at easier scalability and modification of the code, though careful code structuring is essential to avoid added complexity.
- **Quality and Technical Debt:** Despite a high Physical SLOC potentially indicating increased technical debt, a moderate Logical SLOC reflects well-managed functional complexity, key to sustained code quality.

In conclusion, the METRICSTICS project demonstrates a commitment to code clarity and documentation, as observed by its Physical SLOC, while maintaining a focus on efficient functional implementation, as shown by its Logical SLOC. This balance is indicative of a mature approach to software development, prioritizing both current functionality and future project maintainability

Problem 8 : Analysis of Correlation between Logical SLOC and WMC in METRICSTICS Project

This problem presents an analysis of the correlation between Logical Source Lines of Code (SLOC) and Weighted Methods per Class (WMC) for the METRICSTICS project, which is composed of two Python files `main.py` and `metricstics.py`. The relationship is examined using a scatter plot and Spearman's rank correlation coefficient. Logical SLOC and WMC are key metrics in software engineering for assessing the size and complexity of software. This problem aims to explore the correlation between these metrics for the METRICSTICS project to infer any relational insights. For the METRICSTICS project, the Logical SLOC was manually calculated, focusing on executable statements and control structures. The WMC was computed based on the cyclomatic complexity of the methods.

The Logical SLOC and WMC for each file were manually calculated as follows:

- Logical SLOC for `main.py`: 148
- Logical SLOC for `metricstics.py`: 69
- WMC for `main.py`: 35
- WMC for `metricstics.py`: 32

7.1 Scatter plot Analysis between Logical SLOC and WMC

The scatter plot is a type of data visualization that uses Cartesian coordinates to display values for two variables within a set of data. To visualize the correlation between Logical SLOC and WMC, a scatter

plot was created with WMC on the x-axis and Logical SLOC on the y-axis, resulting in two data points as mentioned in the below table:

Class	Logical LOC	WMC	Data Points(x,y)
Metricstics	69	32	(32,69)
Main	148	35	(35,148)

Table 7.1: Class Metrics

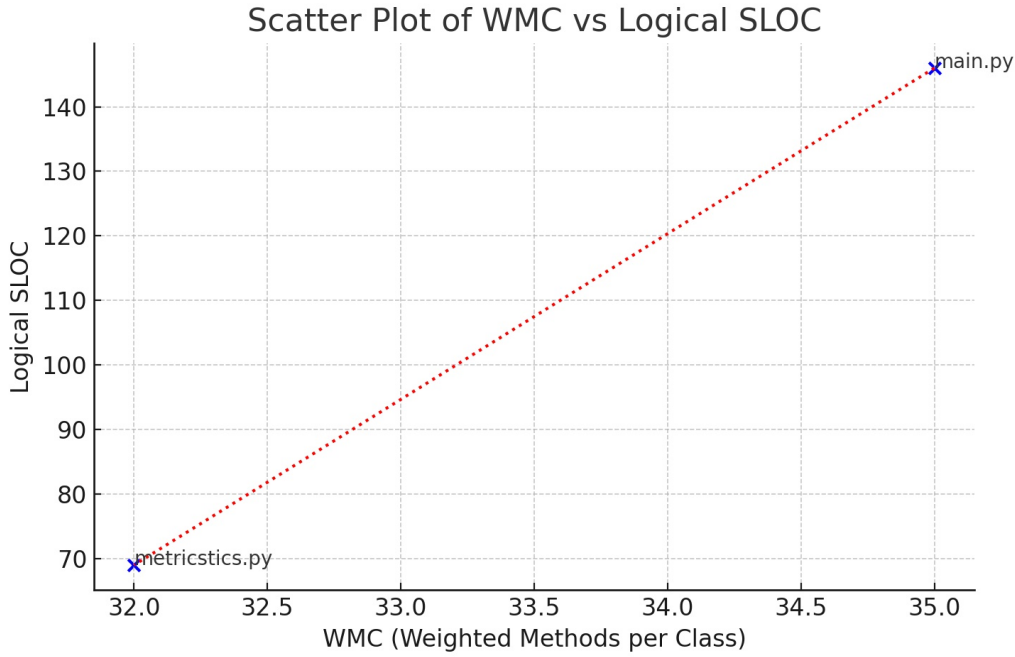


Figure 7.1: Scatter Plot (Logical SLOC and WMC)

Observations: The scatter plot shows that there is a positive relationship between the two metrics since an increase in Logical SLOC corresponds to an increase in WMC. This trend can be critical for software developers and project managers who aim to understand and manage the complexity of their software systems

7.2 Correlation Coefficient Analysis between Logical SLOC and WMC

Before calculating Spearman's rank correlation coefficient, the ranks of Logical SLOC and WMC were determined and compared. The following table presents the ranks and their differences $d = (\text{Rank}(\text{WMC}) - \text{Rank}(\text{SLOC}))$:

File	Logical SLOC	WMC	Rank(WMC)	Rank(SLOC)	d	d ²
main.py	148	35	2	2	0	0
metricstics.py	69	32	1	1	0	0

Table 7.2: Ranks and Differences Table

Spearman's rank correlation coefficient (r_s) is a statistical measure that is calculated using the formula:

$$r_s = 1 - \frac{6 \sum d^2}{n(n^2 - 1)}$$

Where

d: difference between the Rank(WMC) and Rank(SLOC)

n: no. of files

Inserting the values from the table into Spearman's formula:

$$r_s = 1 - \frac{6 \times 0^2}{2(2^2 - 1)}$$

$$r_s = 1$$

Observations: The calculation of the Spearman's rank correlation coefficient for the METRICSTICS project yields a value of 1. This result is quite significant as it indicates a perfect positive correlation between Logical Source Lines of Code (SLOC) and Weighted Methods per Class (WMC). In practical terms, it suggests that as the size of the code (measured in terms of Logical SLOC) increases, the complexity of the software (as indicated by WMC) increases in a directly proportional manner.

Collaboration Environments

1. **Github** : <https://github.com/sehgaldeepanshu985/SOEN-6611-METRICSTICS-Team-N/tree/main/D2>
2. **Google Drive** : <https://drive.google.com/drive/folders/1gxH5prfIofg2zCvq7tEKBnhNrBBMqtSy?usp=sharing>
3. **Roles and Responsibilities Tracker**: https://docs.google.com/spreadsheets/d/1srGVXL0wph_ztf5VTQf1I-01Sq-ruTeE/edit?usp=sharing&ouid=115551500586105153010&rtpof=true&sd=true

References

References

1. <https://www.c-sharpcorner.com/uploadfile/nipuntomar/cocomo-1-cocomo81-constructive-cost-estimation-model/>
2. https://en.wikipedia.org/wiki/Source_lines_of_code
3. https://en.wikipedia.org/wiki/Use_case_points
4. https://en.wikipedia.org/wiki/Cyclomatic_complexity
5. <https://chartio.com/learn/charts/what-is-a-scatter-plot/#:~:text=What%20is%20a%20scatter%20plot,to%20observe%20relationship%20between%20variables.>
6. <https://geographyfieldwork.com/SpearmanRank.htm#:~:text=Spearman's%20Rank%20correlation%20coefficient%20is%20a%20technique%20which%20can%20be,between%201%20and%20minus%201.>
7. <https://www.python.org/>
8. Lecture notes by Prof. Pankaj Kamthan