GitHub Repository Cloning and Code Analysis

## Table of Contents:

1. Introduction

2. Prerequisites

3. Project Structure

4. Installation

5. Usage

i. Cloning GitHub Repositories

ii. Analyzing Code Metrics

6. Key Functions

i. Cloning Repositories

ii. Analyzing Repositories

iii. Merging Results

7. Outputs

8. Troubleshooting

9. Future Enhancements

10. visualization

11. Conclusion

1. Introduction

This project is designed to automate the process of cloning GitHub repositories and analyzing their code maintainability metrics using tools such as Radon and Pylint (or similar). It performs multiple operations, including:

Cloning repositories from GitHub using the URLs provided in a CSV file.

Analyzing each repository's codebase for various metrics (e.g., Maintainability Index, Cyclomatic Complexity, etc.).

Merging the analysis results into a comprehensive CSV report.

**Maintainability Index (MI):**

The Maintainability Index (MI) is a software metric used to measure how easy it is to maintain a particular codebase. It evaluates how complex, understandable, and modifiable the code is, providing insight into how much effort would be required to maintain or modify the code in the future.

Key Points about Maintainability Index:

Purpose:

The primary goal of the Maintainability Index is to assess how easily a piece of software can be understood, fixed, or enhanced over time. It is particularly useful for identifying areas of a codebase that may become problematic as software grows and evolves.

Calculation: The MI is calculated using several factors, including:

Cyclomatic Complexity (CC): Measures the number of linearly independent paths through the program's source code. Higher values indicate more complex and harder-to-maintain code.

Lines of Code (LOC): A higher number of lines can indicate a more complex and difficult-to-maintain system.

Halstead Volume: A measure of the size of the implementation of an algorithm. It is based on the number of operators and operands in the code.

Comment Density: Higher comment density usually improves code readability and, therefore, maintainability.

Here’s a common formula for MI (there are variations):

MI = 171 - 5.2 \* log(LOC) - 0.23 \* CC - 16.2 \* log(Halstead Volume)

If comments are included, they can increase the MI score, making the code easier to maintain.

Interpretation of MI:

High MI (> 85): Indicates the code is very maintainable, easy to understand, and likely easier to modify.

Moderate MI (between 65 and 85): Indicates the code is moderately maintainable, with some areas that may need improvement.

Low MI (< 65): Indicates the code is difficult to maintain, likely due to high complexity, lack of clarity, or poor organization.

Factors Affecting MI:

Code complexity: High cyclomatic complexity or deep nesting of code will reduce the MI.

Size of the code: Large blocks of code without modularization or refactoring will lead to lower MI.

Code readability: Lack of comments, unclear variable names, or poorly structured logic can decrease the MI.

Usage:

Code Reviews: MI helps in code reviews by highlighting sections of the code that might need refactoring or simplification.

Refactoring Efforts: Low MI values can signal that a portion of the code needs to be restructured for better maintainability.

Project Health Monitoring: Over time, as code evolves, MI can be tracked to ensure that maintainability does not degrade.

Limitations:

MI provides a quantitative measure but does not capture all aspects of code quality, like code style, best practices, or logical correctness.

The metric can sometimes be gamed (e.g., by artificially increasing comments) without improving real maintainability.

2. Prerequisites

To run this project, the following software and libraries are required:

Git: For cloning the repositories.

Python 3.6+: The project is written in Python, and Python 3.6 or above is required.

Pandas: For working with CSV files and data analysis.

Subprocess: For running shell commands such as Git clone and analysis tools.

Radon: For code maintainability analysis.

Pylint (optional): For analyzing code quality.

Make sure to install these packages by running:

pip install pandas radon pylint

3. Project Structure

The project directory is organized as follows:

Python\_program\_complexities/

│

├── cloned\_projects/ # Directory where cloned GitHub repositories will be stored

├── complexity\_calc/ # Directory where complexity calculations are handled

├── github\_projects\_with\_domains.csv # CSV file containing GitHub repository URLs and project details

├── merged\_github\_radon\_analysis.csv # Final output after merging repository details with analysis metrics

├── project\_analysis\_with\_metrics.csv # CSV file containing in-depth code analysis

├── projects\_clone.py # Python script to clone repositories

├── radon\_analysis.py # Python script to analyze repositories using Radon

├── merging\_results.py # Python script to merge analysis results

└── README.md # Project documentation file

4. Installation

1. Clone the Project: Clone the project to your local machine:

<https://github.com/vishnu366/python_program_compexities>

2. Install Dependencies: Run the following command to install the required Python packages:

pip install -r requirements.txt

3. Git Configuration: Ensure that Git is installed and properly configured on your system. You can check Git installation with:

git –version

5. Usage

Cloning GitHub Repositories

1. The project fetches repositories listed in the CSV file (github\_projects\_with\_domains.csv).

2. Run the projects\_clone.py script to clone the repositories:

python projects\_clone.py

3. The script deletes the existing cloned\_projects/ directory if it exists, then recreates it and clones the repositories listed in the CSV file.

Analyzing Code Metrics

1. After cloning, use the radon\_analysis.py script to analyze code complexity and maintainability for the cloned repositories:

python radon\_analysis.py

2. This script will process all the Python files in each cloned repository and calculate various metrics:

Maintainability Index (MI)

Cyclomatic Complexity (CC)

Halstead Volume

3. The results will be saved to a CSV file radon\_analysis\_results.csv.

Merging Results

1. To combine the GitHub project metadata with the analysis results, run the merging\_results.py script:

python merging\_results.py

2. This script merges the CSV files and generates merged\_github\_radon\_analysis.csv, which contains a comprehensive overview of the projects and their analysis results.

6. Key Functions

Cloning Repositories (projects\_clone.py):

Iterates through the list of repositories in the CSV file.

Clones each repository into the cloned\_projects/ directory.

Automatically deletes and recreates the directory with each execution.

Analyzing Repositories (radon\_analysis.py):

Uses Radon to compute maintainability, cyclomatic complexity, and other metrics.

Outputs the analysis results to a CSV file.

Merging Results (merging\_results.py):

Merges project metadata (e.g., stargazers, forks, watchers) with analysis metrics (e.g., MI, CC).

Saves the merged data into a new CSV file for further analysis.

7. Outputs

The main outputs of the project are:

1. cloned\_projects/: Directory containing cloned GitHub repositories.

2. radon\_analysis\_results.csv: Contains analysis metrics for each repository.

3. merged\_github\_radon\_analysis.csv: Final CSV file with metadata and analysis metrics merged together.

4. project\_analysis\_with\_metrics.csv: CSV file with detailed metrics such as lines of code, number of functions, and operators.

8. Troubleshooting

Permission Issues on Windows: If you encounter permission errors while deleting the cloned\_projects/ directory, ensure that no files are open or locked by another process. Running the script with administrator privileges can help.

Cloning Failure: If any repository fails to clone, check the URL in the CSV file and ensure the repository is accessible.

9. Future Enhancements

Automatic Dependency Installation: Implement automatic dependency installation for each cloned repository.

Multi-threading: Enhance the cloning and analysis process by running multiple threads for faster execution.

GUI: Add a graphical user interface to visualize and analyze the results more interactively.

10. Visualizations:

i) Cyclomatic Complexity vs Maintainability Index (MI)

A screenshot of a computer

Description automatically generated

Key Insights:

High Cyclomatic Complexity → Low Maintainability:

As Cyclomatic Complexity increases (rightward movement along the x-axis), the average Maintainability Index decreases (lower y-axis values).

This is expected because more complex code tends to be harder to maintain.

Threshold Effects:

There seems to be a clear threshold at Cyclomatic Complexity = 3, where the MI peaks. After this point, the maintainability steadily decreases as complexity rises.

Simpler code with lower Cyclomatic Complexity (between 0.0 to 3.0) tends to have higher MI, indicating it's more maintainable.

Rapid Decrease in MI:

The graph shows that as Cyclomatic Complexity exceeds 6.0, the Maintainability Index drops more sharply, indicating that very complex code is significantly harder to maintain.

Interpretation:

Cyclomatic Complexity and MI are inversely related. As the complexity of code grows, it becomes harder to maintain, as indicated by a lower MI.

The ideal goal in software development is to keep Cyclomatic Complexity low (preferably between 0.0 and 3.0) to ensure that the code remains maintainable.

For very high Cyclomatic Complexity values (e.g., 13.0 and beyond), you may need to refactor the code, break down complex functions, or simplify logic to improve maintainability.

ii) No. of functions vs MI

A screenshot of a computer

Description automatically generated

Key Insights:

Maintainability Generally Decreases with Function Count:

In general, as the number of functions increases, there is a trend for the MI to decrease slightly. This makes sense because more functions can indicate larger and potentially more complex codebases, which can be harder to maintain.

However, there are some fluctuations. For instance, the MI for 161 functions is higher than for 187 functions, suggesting that the specific code structure and complexity have a significant effect.

Cyclomatic Complexity's Influence on MI:

The Cyclomatic Complexity is tied to the number of functions, and complex functions generally reduce maintainability. For instance, functions with higher complexity (shown by higher function counts) tend to have lower MI.

There are peaks and valleys in the MI, suggesting that the maintainability is not simply dependent on the number of functions but also the complexity of the code inside them.

Outliers:

The bars for 0 functions and few functions (like 1 function) have higher MI values, implying that simpler code or smaller functions tend to be much more maintainable.

On the contrary, function counts like 187 and 249 are associated with lower MI values, signaling that these areas of the code are more complex and less maintainable.

Interpretation:

Smaller function counts (like 1 to 30 functions) generally have higher MI, indicating that simpler, smaller blocks of code are easier to maintain. As the number of functions grows, particularly beyond 100 functions, the MI drops more frequently, indicating more complexity and reduced maintainability.

The chart reveals that maintainability is not just affected by the number of functions but also by how complex these functions are. You might see spikes in MI where the number of functions is relatively high, but the complexity is well-managed.

iii) Lines of code vs MI

A screenshot of a computer

Description automatically generated

General Observations

1. Higher Maintainability with Lower LOC:

Projects with fewer lines of code tend to have a higher maintainability score. This trend is visible for projects with low LOC values, as they maintain a high MI (around 80–90). This makes sense because smaller projects are usually easier to manage, understand, and maintain.

2. Varied MI for Higher LOC:

For projects with a greater number of lines of code (e.g., beyond 2000), the MI is more variable. While some large projects maintain a relatively high MI (above 70), others exhibit a lower MI (below 60).

This indicates that larger codebases can still be maintainable if they are well-structured, though larger size often leads to a drop in maintainability due to the increased complexity.

3. Lower MI with Higher LOC:

Projects with the lowest MI scores (around 40–50) tend to have more lines of code. This reflects that the complexity and difficulty of maintaining a project generally increase with its size.

Key Insight:

The graph suggests that as the lines of code increase, the maintainability of the project tends to decrease, although there are exceptions where some larger projects still maintain a high MI. However, for small to medium projects, the maintainability remains consistently high.

iv) Domain vs MI:

A screenshot of a computer

Description automatically generated

Observations:

1. Data Science and Web Development:

The Data Science and Web Development domains show the highest average MI values, indicating that projects in these fields tend to have higher code maintainability.

This could imply that these domains often employ better coding practices or have smaller, more modular codebases that contribute to easier maintainability.

2. General Domain:

The General domain shows a moderate maintainability index, slightly lower than Data Science and Web Development.

This indicates that the projects in this category may have some variability in coding quality, resulting in a lower MI on average.

3. Machine Learning/AI:

The Machine Learning/AI domain shows the lowest average MI, suggesting that projects in this field may be more complex, have larger codebases, or are less modular, which can make them harder to maintain.

Machine learning projects often involve a lot of experimentation, data processing, and complex algorithms, which might explain the reduced maintainability.

General Insights:

The graph shows a clear difference between domains in terms of maintainability. Fields like Data Science and Web Development are generally more maintainable, likely due to the use of well-established libraries, frameworks, and best practices.

On the other hand, domains like Machine Learning/AI may suffer from lower maintainability due to the inherent complexity of the tasks they are solving and the rapid pace of experimentation in the field.

This chart is helpful in comparing how maintainability differs across domains, giving insights into which fields may need more focus on improving coding practices for easier long-term maintenance.

11. Conclusion:  
Conclusion Based on the Graphs:

1. Maintainability Index (MI) Trends:

From the "Lines of Code vs Maintainability Index (MI)" graph, it is evident that there is no direct linear relationship between the number of lines of code and the Maintainability Index (MI). Some projects with fewer lines of code (like "big-list-of-naughty-strings") have a high MI, indicating that smaller codebases are easier to maintain. However, other large projects like "algorithms" also have a high MI, suggesting that large codebases can still be highly maintainable if well-structured.

2. Number of Functions and Operators:

Projects with a higher number of functions and operators generally tend to have a lower MI. This suggests that as the complexity of the code (in terms of the number of functions and operations) increases, the maintainability decreases. For example, projects like "grok-1" and "python-spider" show lower MI values with a larger number of functions, implying higher complexity and reduced maintainability.

3. Category-Based Maintainability:

From the "Domain vs Maintainability Index (MI)" graph, we observe that different domains exhibit distinct patterns in terms of MI:

Web Development and Data Science projects generally have the highest MI, indicating that projects in these domains are often easier to maintain. This could be due to established frameworks, best practices, and more standardized development processes in these fields.

Machine Learning/AI projects show the lowest MI, suggesting that these projects are typically harder to maintain. This is expected, given that machine learning code often deals with complex algorithms, large datasets, and experimental models, making it less maintainable in the long run.

The General domain lies somewhere in between, with moderate maintainability, likely because it encompasses a variety of project types with varying levels of complexity.

4. Complexity vs Maintainability:

Projects with high cyclomatic complexity, such as "nginx-proxy" and "python-spider", generally have lower MI values. This aligns with the expectation that more complex logic reduces the maintainability of code. Projects with simpler logic, as indicated by low cyclomatic complexity (e.g., "minGPT"), tend to have a higher MI, indicating better maintainability.

5. Influence of Size and Operators:

The number of lines of code and the number of operators in the code also play a role in maintainability. For instance, projects with excessive operators or very long code (e.g., "algorithms") may have a lower MI due to the increased difficulty of managing such codebases over time.

General Conclusion:

Small, modular projects generally have higher maintainability, as seen from the MI scores of smaller projects like "big-list-of-naughty-strings."

Well-established domains like Web Development and Data Science tend to exhibit higher maintainability, likely due to standardized frameworks, best practices, and more modular code structures.

Machine Learning/AI projects have lower maintainability, likely due to the inherent complexity of the domain. More focus on coding standards, modularity, and maintainability practices is needed for projects in this field.

There is a correlation between code complexity (functions, operators, and lines of code) and maintainability. Simplifying code, reducing complexity, and keeping functions smaller can positively influence the MI and improve long-term maintainability.

In summary, the analysis suggests that focusing on modular design, reducing code complexity, and leveraging best practices in specific domains can significantly improve the maintainability of projects, making them easier to work with over time.