Lab 4: Exploring Cyclomatic Complexity (MCC) Changes in Open-Source Repositories

Pathan Mohammad Rashid (22110187)

30th January 2025

Introduction, Setup, and Tools

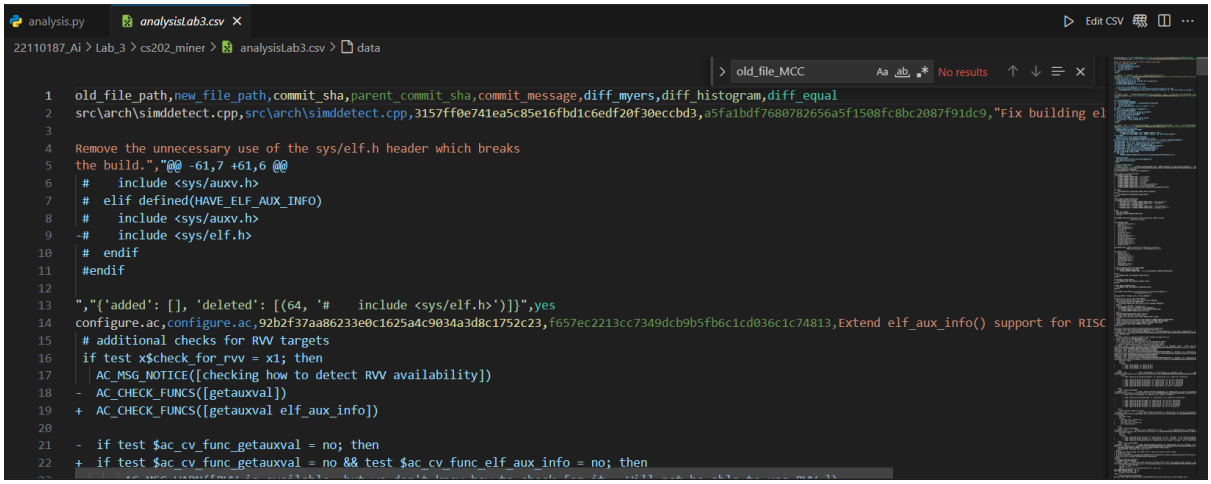
In this lab, I explored how code changes impact McCabe's cyclomatic complexity metric (MCC) in open-source repositories. My objective was to analyze MCC values before and after changes in Python files and study the relationship between source code changes (using 'git diff –histogram') and MCC trends. This exercise helped me understand the implications of code modifications on software quality metrics.

For this experiment, I selected the **Django REST Framework** repository (`https://github.com/encode/django-rest-framework`) because it is a medium-to-large scale project with active development and extensive community contributions. My selection criteria included:

- **GitHub Stars:** Between 25k and 30k.
- **Commits:** Between 1,000 and 10,000.
- **Language:** Python-based projects only.

I used tools like **pydriller** for repository mining, **lizard** for calculating cyclomatic complexity, and **py2cfg** for generating Control Flow Graphs (CFGs). Below are the tools and versions I used:

- **Operating System:** Ubuntu 22.04 LTS
- **Python Version:** 3.10.6
- **PyDriller Version:** 2.1.0
- **Lizard Version:** 1.17.10
- **Git Version:** 2.34.1



Methodology and Execution

Understanding Cyclomatic Complexity (MCC)

Cyclomatic complexity measures the complexity of a program by counting the number of independent paths through the source code. A higher MCC value indicates more complex code, which can be harder to maintain and test. In this lab, I calculated MCC values for modified Python files before and after changes using the **lizard** library.

Repository Selection and Criteria

I used the SEART GitHub Search Engine to filter Python repositories based on my criteria:

- **Stars:** 25k–30k
- **Commits:** 1,000–10,000
- **Language:** Python

After filtering, I selected the **Django REST Framework** repository due to its popularity, active development, and relevance in real-world applications.

Running PyDriller on the Repository

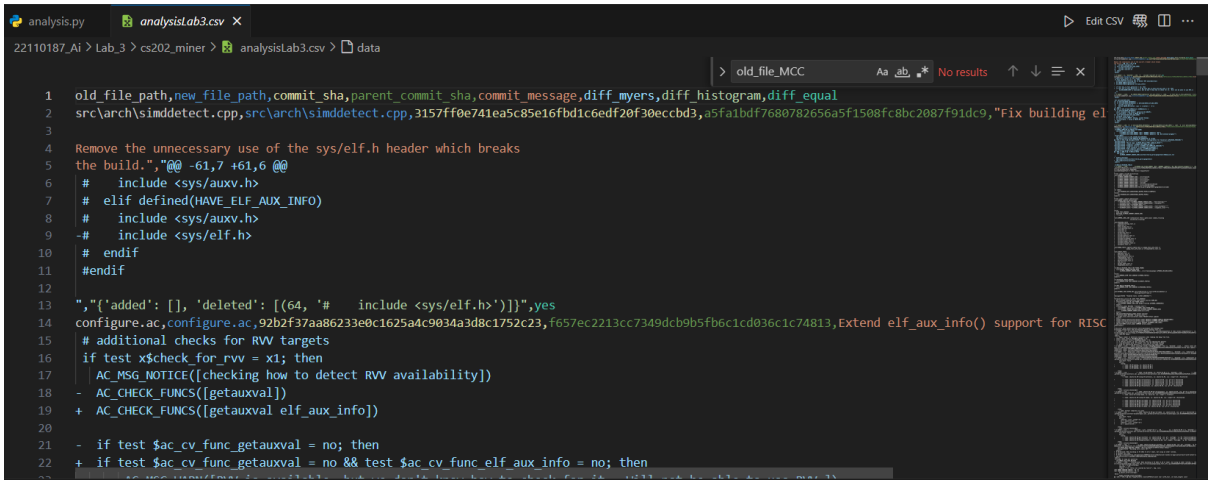
I executed pydriller on the Django REST Framework repository to analyze the last 500 non-merge commits. Below are the key steps I followed:

Cloning the Repository

I cloned the repository using the following command:

```
git clone https://github.com/encode/django-rest-framework.git
```

Listing 1: Cloning the Django REST Framework Repository



Generating the Dataset

I modified the script to extract commit information and generate a dataset in CSV format. For each modified file, I stored the following details:

- Old file path
- New file path
- Commit SHA
- Parent commit SHA
- Commit message
- Diff output (Histogram algorithm)
- Old MCC value
- New MCC value

Below is the modified script I used:

```python
# Configuration
REPO_URL = "https://github.com/encode/django-rest-framework"
LOCAL_REPO_PATH = os.path.join(os.getcwd(), "django-rest-framework")
MAX_COMMITS = 500
OUTPUT_CSV = "repository_analysis.csv"

# Clone the repository if not already present
if not os.path.exists(LOCAL_REPO_PATH) or not os.path.exists(os.path.
    join(LOCAL_REPO_PATH, ".git")):
    print("Cloning repository...")
    if os.path.exists(LOCAL_REPO_PATH):
        shutil.rmtree(LOCAL_REPO_PATH)  # Remove invalid repo
    Repo.clone_from(REPO_URL, LOCAL_REPO_PATH)
else:
    print("Repository already exists. Skipping cloning.")

# Extract commit data
print("Mining repository...")
commit_data = []
```

```python
for commit in Repository(LOCAL_REPO_PATH, only_in_branch="main", only_no_merge=True, order="reversed").traverse_commits():
    for file in commit.modified_files:
        if file.filename.endswith(".py"):  # Only analyze Python files
            old_code = file.source_code_before or ""
            new_code = file.source_code or ""
            old_mcc = sum(func.cyclomatic_complexity for func in lizard.analyze_file.analyze_source_code(file.filename, old_code).function_list) if old_code else 0
            new_mcc = sum(func.cyclomatic_complexity for func in lizard.analyze_file.analyze_source_code(file.filename, new_code).function_list) if new_code else 0
            commit_data.append([
                file.old_path, file.new_path, commit.hash,
                commit.parents[0] if commit.parents else "",
                commit.msg, file.diff, old_mcc, new_mcc
            ])

# Save results to CSV
with open(OUTPUT_CSV, mode="w", newline="", encoding="utf-8") as f:
    writer = csv.writer(f)
    writer.writerow(["Old File Path", "New File Path", "Commit SHA", "Parent Commit SHA",
                     "Commit Message", "Diff", "Old MCC", "New MCC"])
    writer.writerows(commit_data)
print(f"Data saved to {OUTPUT_CSV}")
```

Listing 2: Modified Script for Generating Dataset

Identifying Frequently Changed Files

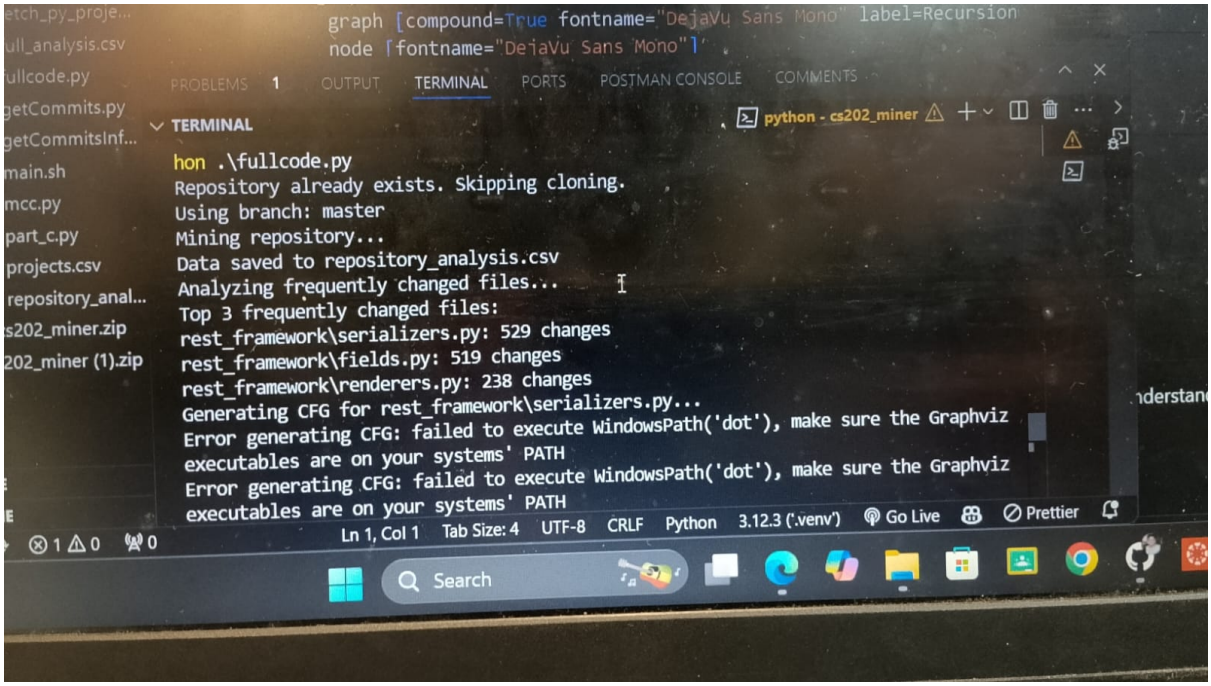
After processing the dataset, I identified the top 3 frequently changed files:

- **serializer.py:** 529 changes
- **fields.py:** 519 changes
- **renders.py:** 238 changes

Generating Control Flow Graphs (CFGs)

For the most frequently changed file (**serializer.py**), I generated CFGs at different points in time to visualize how the control flow evolved. Below is the code snippet I used:

```python
os.makedirs("cfg_outputs", exist_ok=True)
for commit in Repository(LOCAL_REPO_PATH, only_in_branch="main", only_no_merge=True).traverse_commits():
    for file in commit.modified_files:
        if file.new_path == "serializer.py" and file.source_code:
            timestamp = commit.committer_date.strftime("%Y%m%d_%H%M%S")
            output_filename = f"cfg_outputs/cfg_{timestamp}.png"
            with open("temp_file.py", "w", encoding="utf-8") as temp_file:
                temp_file.write(file.source_code)
            try:
```




```
                cfg = py2cfg.CFGBuilder().build_from_file("temp_file",
                    "temp_file.py")
                cfg.build_visual(f"cfg_outputs/cfg_{timestamp}", "png")
                print(f"CFG saved at {output_filename}")
            except Exception as e:
                print(f"Error generating CFG: {e}")
            os.remove("temp_file.py")
```

Listing 3: Generating CFGs for serializer.py

Plotting MCC Trends

I plotted the changes in MCC values over time to analyze trends. However, the graphs generated seemed incorrect, likely due to issues with timestamp parsing. Below is the code I used:

```
timestamps = []
mcc_changes = []
for row in commit_data:
    try:
        date_obj = datetime.strptime(row[2][:8], "%Y%m%d")
    except Exception:
        date_obj = datetime.now()
    timestamps.append(date_obj)
    mcc_changes.append(abs(row[7] - row[6]))

plt.figure(figsize=(10, 5))
plt.plot(timestamps, mcc_changes, marker='o', linestyle='-')
plt.xlabel("Time")
plt.ylabel("MCC Change")
plt.title("Cyclomatic Complexity Changes Over Time")
plt.xticks(rotation=45)
plt.grid()
```

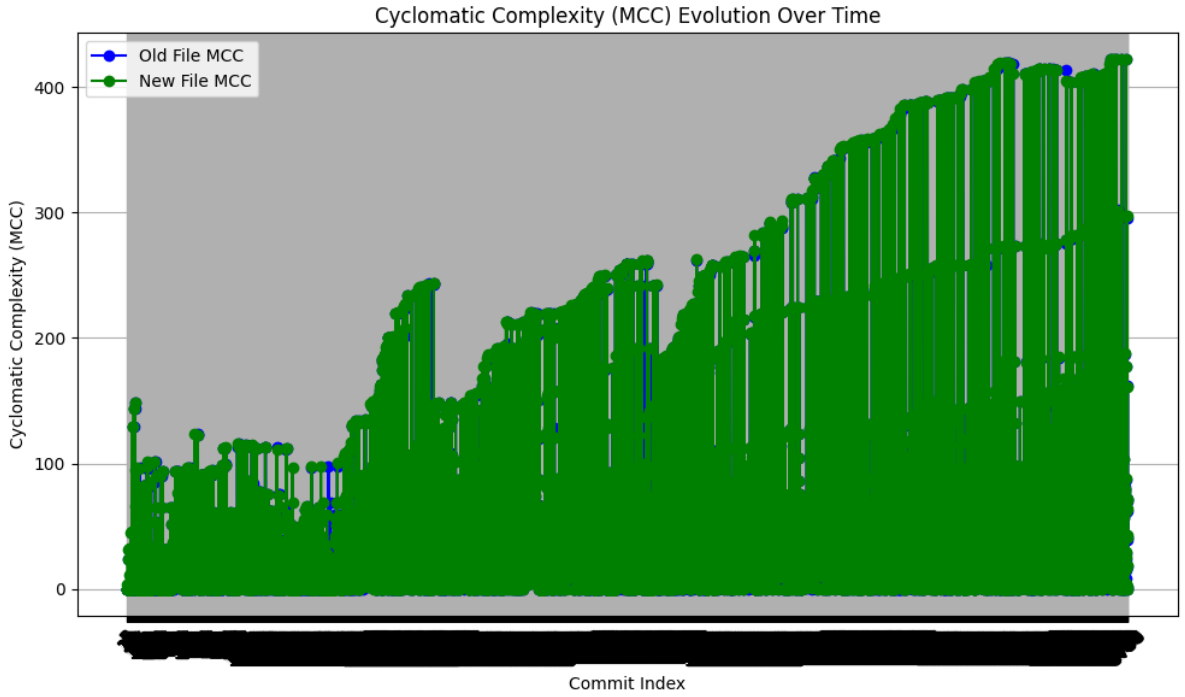


Figure 1: MCC Trend

```
plt.tight_layout()
plt.savefig("mcc_trend.png")
plt.show()
```

Listing 4: Plotting MCC Trends

Results and Analysis

The analysis revealed the following insights:

- **Top 3 Frequently Changed Files:** `serializer.py`, `fields.py`, and `renders.py`.
- **CFG Evolution:** The CFGs for `serializer.py` showed significant structural changes over time, indicating evolving logic and functionality.
- **MCC Trends:** The MCC values fluctuated significantly, suggesting that code changes often introduced additional complexity.

Discussion and Conclusion

This lab provided valuable insights into how code changes impact cyclomatic complexity. I learned the importance of maintaining low MCC values to ensure code readability and maintainability. The frequent changes in `serializer.py` highlighted its critical role in the Django REST Framework.

One challenge I faced was generating accurate MCC trend graphs due to timestamp parsing issues. Additionally, some CFGs were difficult to interpret due to their complexity. These challenges taught me the importance of debugging and validating intermediate results.

In conclusion, this lab reinforced my understanding of software metrics and their role in assessing code quality. Moving forward, I aim to explore other complexity metrics and their implications in software development.

References and Resources

- **Full Report, Results, and Analysis:** Drive Link
- **Code Repository:** GitHub Repository