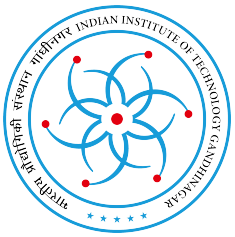

Indian Institute of Technology

Department of Computer Science

CS202: Software Tools and Technologies
Assessment 1

Jaskirat Singh Maskeen

(23110146 | jaskirat.maskeen@iitgn.ac.in)

Instructor: Prof. Shouvick Mondal

September 7, 2025

Contents

Lab 1

Introduction to Version Controlling, Git Workflows, and Actions

Github Repository for Lab 1

1.1 Introduction

Git is a version control system that helps us manage the versions of our projects either locally or on some Git provider such as GitLab or GitHub. The primary advantage of using GitHub or any such provider is that we can collaborate with other users and share our code online with other potential users. This sort of framework supports active development and constant feedback, as anyone can create an Issue or fork our repository and provide pull requests to improve or fix issues with our code.

The main goal of this lab is to understand version control systems (VCS), set up and configure git, verify the installation, and run simple Git commands to push files.

1.2 Setup and Tools

I have Git, Python, and VSCode installed. Since I have already used GitHub before, I have it preconfigured and installed, so I will skip the screenshots for installation details. I will use Git CLI on the command prompt and GitHub Web Interface to create repositories and run workflows. I will also use pylint to perform static type checking. `git --version` gives `git version 2.45.2.windows.1` as the output. While logging into git using the CLI, we first have to sign in using username and password, and verify on GitHub Mobile, as I have 2FA enabled. I also confirm if the current credentials are set (Fig. 1.1). The `--global` flag is

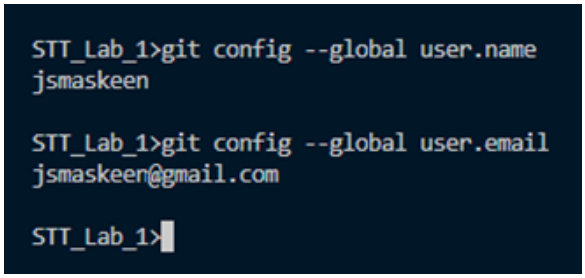

Figure 1.1: Configuring the git CLI

used because I have two GitHub accounts on my laptop, and I want to verify the global configuration. These commands were looked through using https://git-scm.com/docs/git-config. My working directory is named `STT_Lab_1`.

1.3 Methodology and Execution

I first initialized a repository in my working directory using `git init`. I have put in a sample file named **trib.py**, which calculates the n^{th} Tribonacci number. I also added a **readme.md** file, which will get rendered on the link of this repository. I also plan to have a single repository on GitHub, with a new branch for each lab to avoid clutter. Hence, I will also rename the branch to **lab1**,

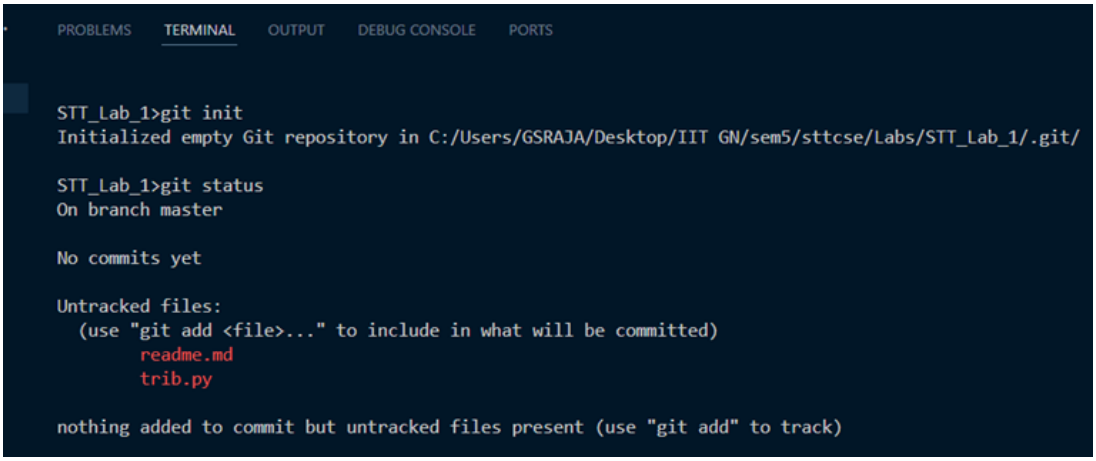

Figure 1.2: Initializing a git repository

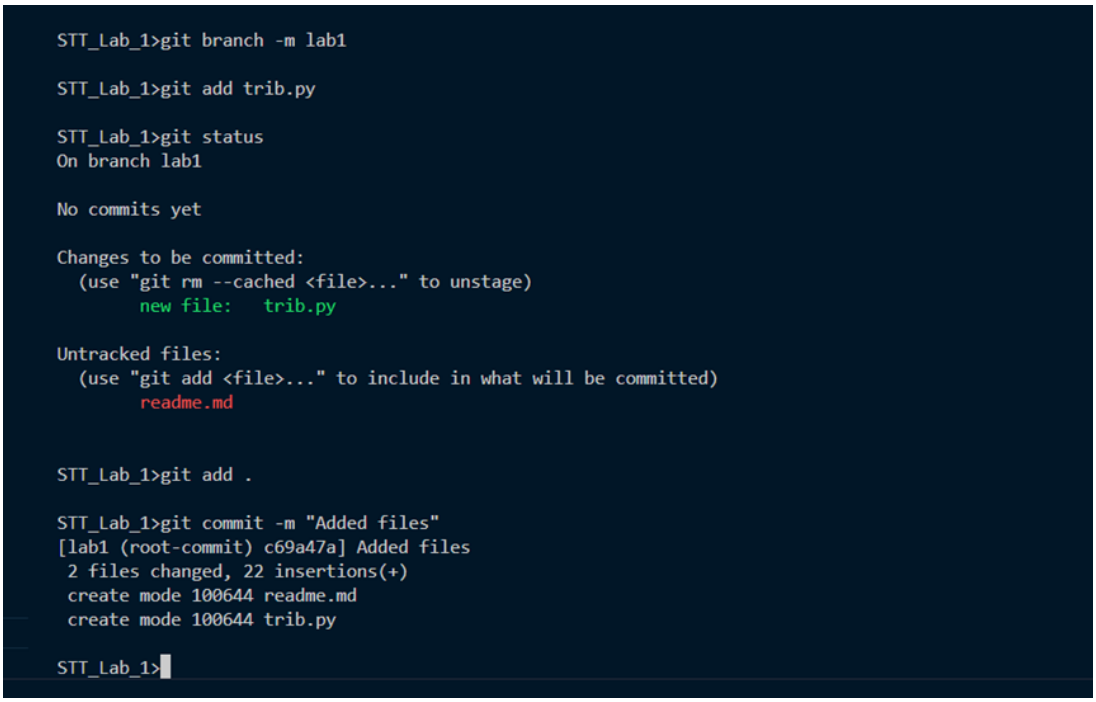

Figure 1.3: Committing to **lab1** branch.

I can check the status of the repository using `git status` (Fig. 1.2), and add files one by one using git add filename, or all upstaged files (i.e., which have a delta), at once using `git add .`, I can commit these changes using the command `git commit -m "Added files"`, where "Added files" is a commit message (Fig. 1.3). `git log` is also a useful command to check the history of the commits being made. Now I create a GitHub repository, **CS202-STTCSE**, which will be the repository where I will upload all the labs, just on their respective branches. GitHub Link: https://github.com/jsmaskeen/CS202-STTCSE/tree/lab1.

I then added the local repository to GitHub using the commands,
`git remote add origin git@github.com:jsmaskeen/CS202-STTCSE.git`,
Followed by `git push -u origin lab1`.

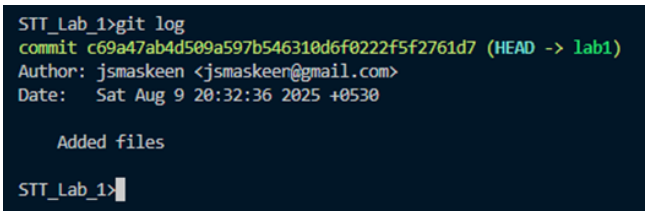

Figure 1.4: `git log` command

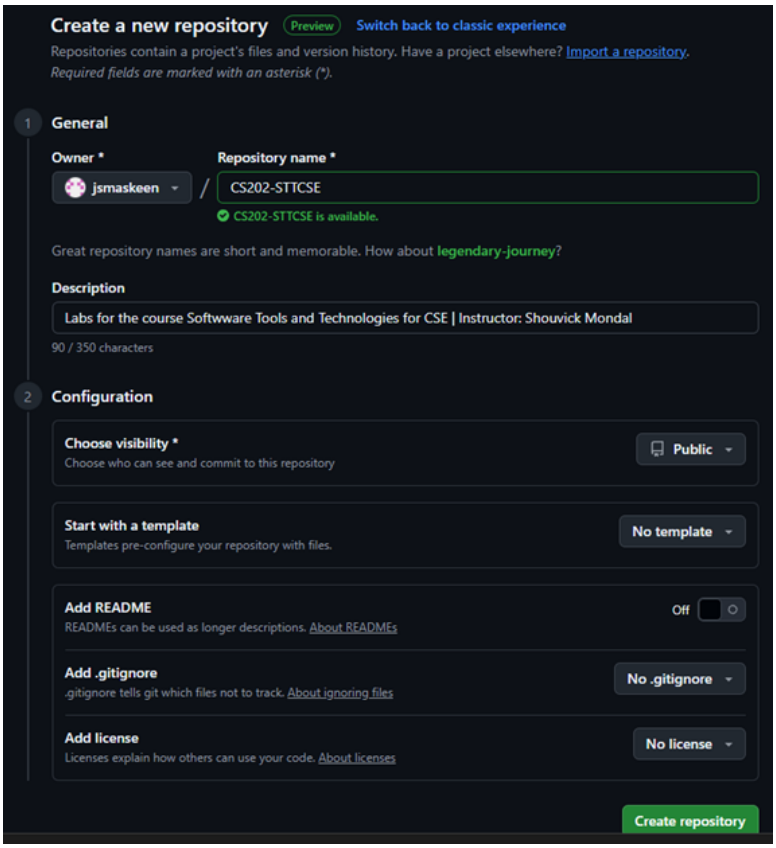

Figure 1.5: GitHub eepository creation

Figure 1.6: Pushing the code to remote repository on GitHub

So, in this way, my local repository is now on GitHub. Now, if I do any commits and push them locally, they will reflect on GitHub as well. For example, I add another file, say **mergesort.py**. I will stage and commit the file, but not push it. Then I will clone this repository elsewhere, and to demonstrate the pull command, I will push the **mergesort.py** file after I have cloned the repository. Please see the sequence of commands carefully ((Figures 1.7, 1.8, 1.9, 1.10, and 1.11)).

This demonstrated the `git pull` as well as `git clone` commands. Now I have to create a pylint

```
PROBLEMS  TERMINAL  OUTPUT  DEBUG CONSOLE  PORTS

STT_Lab_1>git status
On branch lab1
Your branch is up to date with 'origin/lab1'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        mergesort.py

nothing added to commit but untracked files present (use "git add" to track)

STT_Lab_1>git add .

STT_Lab_1>git commit -m "Added mergesort"
[lab1 114ceed] Added mergesort
 1 file changed, 56 insertions(+)
 create mode 100644 mergesort.py
```

Figure 1.7: **Step 1:** Stage the **mergesort.py** and commit

```
C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\Labs>cd ..

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse>git clone https://github.com/jsmaskeen/CS202-STTCSE.git
Cloning into 'CS202-STTCSE'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 0), reused 4 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (4/4), done.

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse>cd CS202-STTCSE

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>git status
On branch lab1
Your branch is up to date with 'origin/lab1'.

nothing to commit, working tree clean

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>git log
commit c69a47ab4d509a597b546310d6f0222f5f2761d7 (HEAD -> lab1, origin/lab1, origin/HEAD)
Author: jsmaskeen <jsmaskeen@gmail.com>
Date:   Sat Aug 9 20:32:36 2025 +0530

    Added files

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>tree /F
Folder PATH listing for volume Windows
Volume serial number is E6E3-610C
C:.
    readme.md
    trib.py

No subfolders exist


C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>
```

Figure 1.8: **Step 2:** Clone the repository elsewhere

```
STT_Lab_1>git push -u origin lab1
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 16 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 765 bytes | 765.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To github.com:jsmaskeen/CS202-STTCSE.git
   c69a47a..114ceed  lab1 -> lab1
branch 'lab1' set up to track 'origin/lab1'.

STT_Lab_1>
```

Figure 1.9: **Step 3:** Now I push to GitHub.

```
C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>git fetch
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 745 bytes | 57.00 KiB/s, done.
From https://github.com/jsmaskeen/CS202-STTCSE
   c69a47a..114ceed  lab1       -> origin/lab1

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>git status
On branch lab1
Your branch is behind 'origin/lab1' by 1 commit, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

Figure 1.10: **Step 4:** Check with the cloned repository (`git fetch`, `git status`)

```
C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>tree /f
Folder PATH listing for volume Windows
Volume serial number is E6E3-610C
C:.
    readme.md
    trib.py

No subfolders exist


C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>git pull
Updating c69a47a..114ceed
Fast-forward
 mergesort.py | 56 ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++
 1 file changed, 56 insertions(+)
 create mode 100644 mergesort.py

C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\CS202-STTCSE>tree /f
Folder PATH listing for volume Windows
Volume serial number is E6E3-610C
C:.
    mergesort.py
    readme.md
    trib.py

No subfolders exist
```

Figure 1.11: **Step 5:** Check the file structure before and after pulling. (git pull)

workflow, so I have deliberately kept some issues in the **mergesory.py** file, such as unused imports, unused

functions, and an unused while loop. I hope to see pylint detect all these issues.

To create a pylint workflow, I configured the GitHub action corresponding to pylint.

Once I clicked on configure, a **pylint.yml** file appeared in editing format under a new folder named workflows (Fig. 1.12). Once I committed and pushed this file on the GitHub UI (This means I will have to do `git pull` on local (Fig. 1.13), to stay up to date), the action ran and gave a few errors.

Whenever the action fails, we are greeted with a ✗ (a red cross) (Fig. 1.14), and whenever it passes, we are greeted with a ✓ (a blue tick).

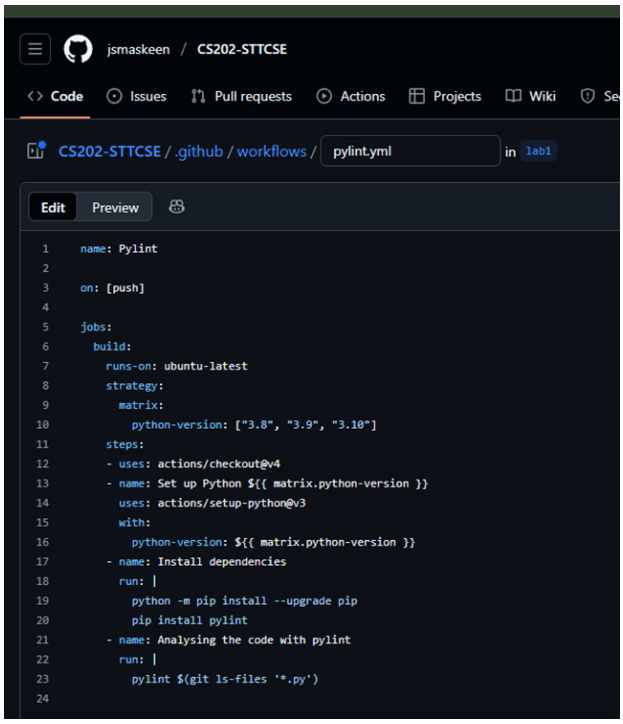

Figure 1.12: Pylint workflow

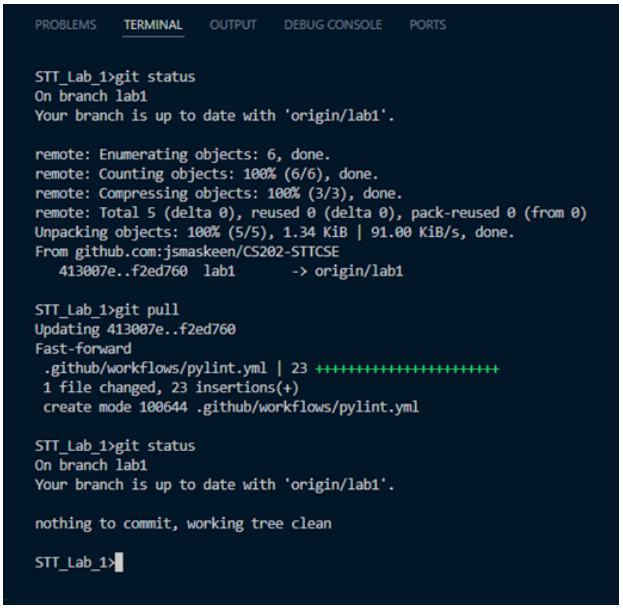

Figure 1.13: Pulling on local to stay up to date

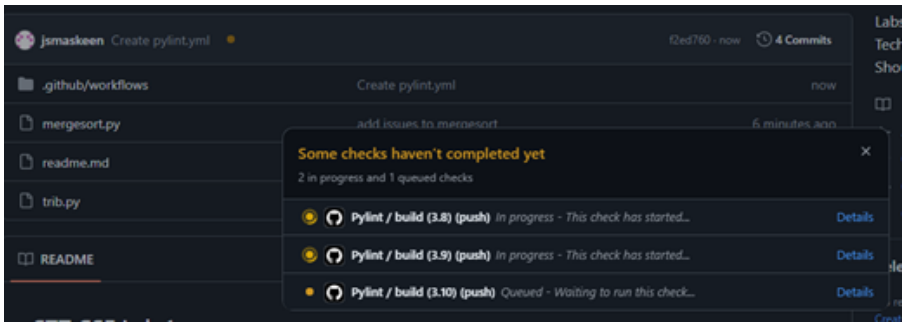

Figure 1.14: Workflow is running

1.4 Results and Analysis

Since I uploaded an erroneous file (which executes correctly, but has unused code, and should fail static checking) (Fig. 1.15).

Looking carefully at the details, we see multiple issues (Fig. 1.16, Fig. 1.17). However, the workflow was not able to detect a few issues, for example:

mergesort.py > merge

```python
import random, pandas, numpy

def mergeSort(ls):
    if len(ls) <= 1:
        return ls
    mid = len(ls) // 2
    left_half = mergeSort(ls[:mid])
    right_half = mergeSort(ls[mid:])
    return merge(left_half, right_half)

def merge(left, Right):
    mergeResult = []
    i = 0;j=0
    if len(left) > len(Right):
        left, Right = Right, left
    if len(left) > len(Right):
        left, Right = Right, left
    while i < len(left) and j < len(Right):
        if left[i] < Right[j]:
            mergeResult.append(left[i])
            i += 1
        else:
            mergeResult.append(Right[j])
            j += 1
    while i < len(
        left
    ):
        mergeResult.append(left[i])
        i += 1

    while j < len(Right):
        mergeResult.append(Right[j])
        j += 1

    return mergeResult
    print(mergeResult)

def unused_function():
    for k in range(5):
        print("Waterboy ?", k)
    return "Firegirl"
def main():
    nums = [random.randint(0, 100) for _ in range(15)]
    print("Unsorted:", nums)
    sorted_nums = mergeSort(nums)
    print("Sorted:", sorted_nums)

main()
```

Figure 1.15: **mergesort.py** file (with errors)

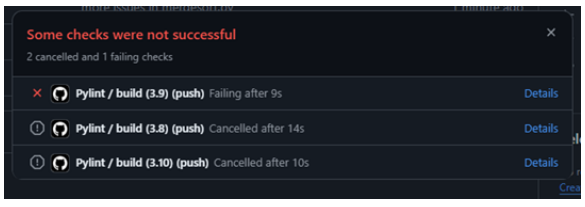

Figure 1.16: Failed Checks

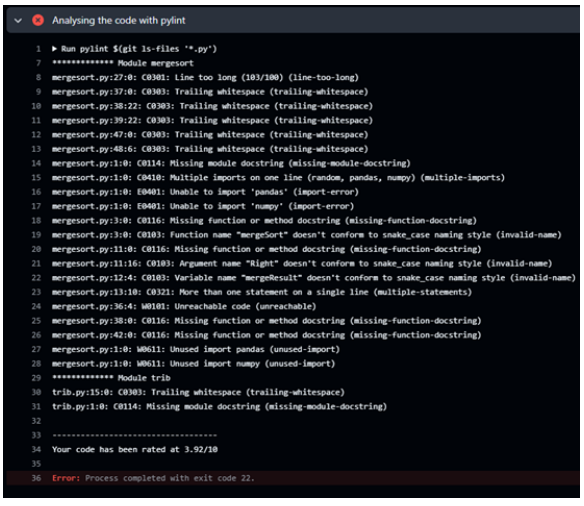

Figure 1.17: Issues found by workflow.

In Fig. 1.18 we have the same `if loop` repeated (which swaps left and right), which serves no function. So, it should be kept only once. [But static analysis cannot capture that]. It was also not able to detect the unused function. To fix the issues, I have edited the docstrings, multiple statements in a single line, and line too long issues.

```python
def merge(left, right):
    """Merges two sorted lists."""
    merge_result = []
    i = 0
    j = 0
    if len(left) > len(right):
        left, right = right, left

    if len(left) > len(right):
        left, right = right, left

    while i < len(left) and j < len(right):
        if left[i] < right[j]:
            merge_result.append(left[i])
            i += 1
        else:
            merge_result.append(right[j])
            j += 1
    while i < len(left):
        merge_result.append(left[i])
        i += 1
    while j < len(right):
        merge_result.append(right[j])
        j += 1
    return merge_result
```

Figure 1.18: Corrected Code

After pushing with these changes, we see that our tests have passed (Fig. 1.19).

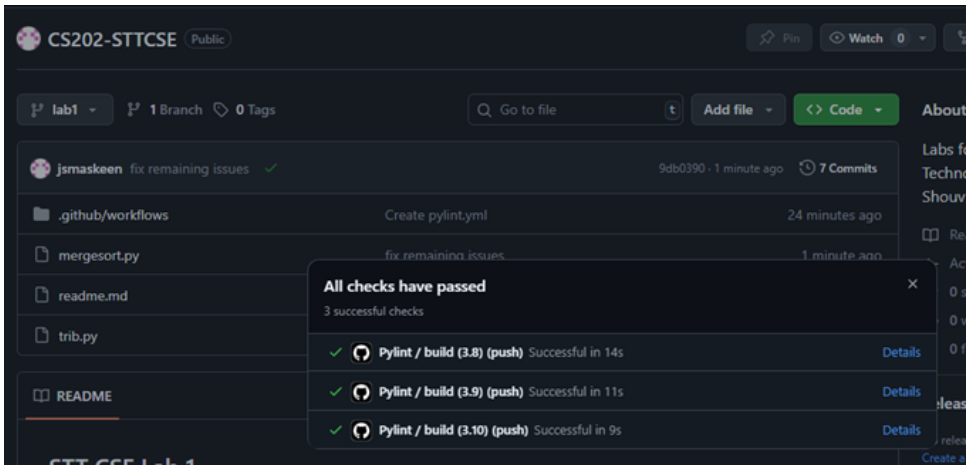

Figure 1.19: All checks passed after fixing the **mergesort.py** file.

1.5 Discussion and Conclusion

This was a really interesting and useful lab session. Using git/GitHub will help us manage our code and collaborate with others; specifically, GitHub actions can configure our workflows, be it static checking or deploying a website after some post-processing. Maybe we write a library that requires a compilation, and various languages comprise the code base, so we can have a GitHub action to compile in the order we like and merge the files, to finally create an executable, for different OS. I did not face any difficulty in this lab; however, while configuring git, I had two accounts. I configured it with SSH tokens, so to set origin for a repository in my main account, I do `git remote add origin git@github.com:jsmaskeen/repo.git`, and for my other account, I do `git remote add origin git@github.com-unofficial:jsxxx/repo.git`.

Another thing I learnt, if we have ssh enabled, we can check if we are authenticated to GitHub via the command `ssh -T git@github.com`. Lastly, pylint was good, but it can only check issues statically. One mode of improvement can be to have two or more workflows, which also perform static checking, and they do not yield conflicting results, to find any issues missed by pylint (for example, the unused functions).

References

[1] "Git documentation," https://git-scm.com/docs, accessed: 2025-08-08.

[2] Stack Overflow, "How to verify github ssh key," https://stackoverflow.com/questions/70371192/how-to-verify-github-ssh-key, 2022, accessed: 2025-08-08.

[3] "Lab assignment document," Google Classroom, accessed: 2025-08-08.

Lab 2

Commit Message Rectification for Bug-Fixing Commits in the Wild

Github Repository for Lab 2

Discussed with Siddhesh (23110347) and Aarsh (23110003)

2.1 Introduction

This lab is targeted at mining bugs in open source software repositories. The basic idea is that developers often fix certain bugs and issues throughout the development of the project. Our goal is to identify these bug-fixing commits using keyword analysis or regex (inspired by `minecpp`). Then, with these commits, I check the `CommitPredictorT5` (CPT5) LLM model for its inference by giving it the diff between each of the modified files in the bug-fixing commit and the previous commit. Once this is done, I check if any rectification is needed. Basically, sometimes developers might fix multiple bugs in one go, and they might not write a descriptive enough commit message. So, I use `gpt-4o-mini` (rectifier), pass it the diff for a modified file, the original message, and get a modified commit message, had that file been committed independently. I then utilize the embedding model from Hugging Face, and try to classify the developer's commit message quality, the quality of the CPT5, and the quality of the rectifier. Algorithms are explained in the subsequent sections.

2.2 Setup and Tools

I am using `pydriller`, the `mamiksik/CommitPredictorT5` (CPT5) LLM model, `sentence-transformers/all-MiniLM-L12-v2` model for embedding generation, and `gpt-4o-mini` for the rectifier. I will also be using a virtual environment, so that my global packages do not interfere with torch, transformers, and pydriller libraries. and `python --version` gives Python 3.12.4, the pydriller version is shown in Fig. 2.1. SEART Search Engine https://seart-ghs.si.usi.ch/ is used to search and filter the repository based on my criteria.

```
(venv) C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\Labs\STT_Lab_2>pip show pydriller
Name: PyDriller
Version: 2.8
```

Figure 2.1: Pydriller version

2.3 Methodology and Execution

I first chose the repository to mine the bugs. For this, I used the tool SEART Search Engine. I set a very simple criterion, I want to analyze the repositories that feature the creation of Discord bots, and they should have

- a minimum of 4000 starts
- a minimum of 1000 commits
- language should be Python
- discord in the title

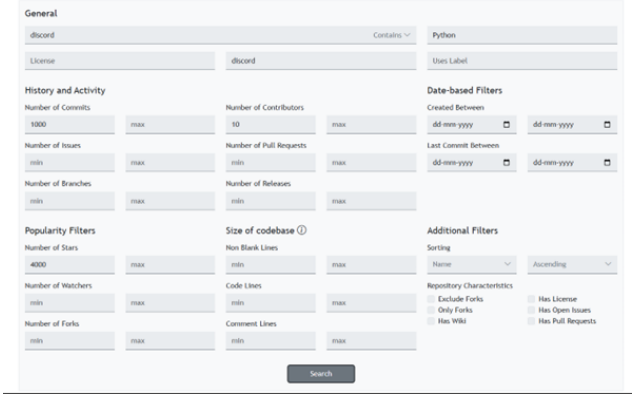

Figure 2.2: Search parameters for finding the repository

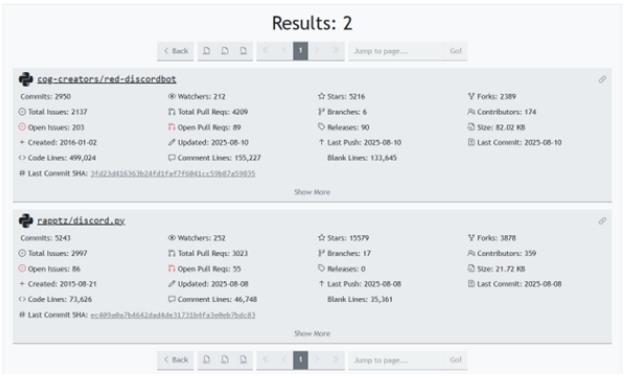

Figure 2.3: Search results

With this filtering, I was down to just two results. I will choose rapptz/discord.py as I have previously used this library to create Discord bots as well. I cloned the repository using `git clone https://github.com/Rapptz/discord.py.git`.

Now I read the pydriller documentation, and wrote a custom script to go through the commits of the repository, and go through each modified file, and store the required details, as asked in the lab assignment.

The next step was to identify the bug-fixing commits. For this, I took inspiration from minecpp and from the lecture slides, and used the 52 keywords that were provided and the regex to search for in commit messages. I also have a word cloud function that gives me the word cloud of the words used in commit messages.

```
self.keywords = list(
    map(
        re.compile,
        [
            "fixed ", " bug", "fixes ", "fix ", " fix", " fixed", " fixes", "crash", "solves", " resolves",
            "resolves ", " issue", "issue ", "regression", "fall back", "assertion", "coverity", "reproducible",
            "stack-wanted", "steps-wanted", "testcase", "failur", "fail", "npe ", " npe", "except", "broken",
            "differential testing", "error", "hang ", " hang", "test fix", "steps to reproduce", "crash",
            "assertion", "failure", "leak", "stack trace", "heap overflow", "freez", "problem ", " problem",
            " overflow", "overflow ", "avoid ", " avoid", "workaround ", " workaround", "break ", " break",
            " stop", "stop "
        ],
    )
)
self.pattern = re.compile(
    r".*((solv(ed|es|e|ing))|(fix(s|es|ing|ed)?)|((error|bug|issue)(s)?)).*"
)
```

Figure 2.4: 52 keywords for bug message identification

```
def gen_wordcloud(self):
    text = " ".join(self.all_commits)
    self.word_cloud = WordCloud(
        width=1600, height=800, background_color="white"
    ).generate(text)
    return self.word_cloud

def bugfix_commits(self, count=-1):
    prev_cur_map: list[list[Commit, Commit]] = []
    i = 1
    while i < len(self.all_commits):
        current = self.all_commits[i]
        prev = self.all_commits[i - 1]
        commit_message = current.msg.lower()
        for kw in self.keywords:
            if (
                kw.search(commit_message)
                or self.pattern.search(commit_message)
                or kw.pattern in commit_message
            ):
                prev_cur_map.append([prev, current])
                break
        i += 1
    if count > -1:
        prev_cur_map = np.array(prev_cur_map)
        rows = np.random.choice(len(prev_cur_map), size=count, replace=False)
        prev_cur_map = prev_cur_map[rows]
    self.prev_cur_map = prev_cur_map
    return prev_cur_map
```

Figure 2.5: Wordcloud function

So, with this, I am able to find around 1875 commits, out of the total 5243 commits, which fix some bug(s). Due to resource and time constraints, I will only analyze 1397 bug-fix commits (out of 1875), which include 2405 modified files. The exact details for the bug-fixing commits are in the next section.

Rectifier is simply gpt-4o-mini running on a specific prompt, which is,

```
You are an assistant that produces a rectified commit message for a single modified
    file.
Input:
FILE: {}
ORIGINAL_COMMIT_MSG: "{}"
PARSED_DIFF:
{}

Produce a short rectified commit message [12-16 words] for this file maybe
    consisting of:
- Title
- One-line root cause
- One-line change summary

DO NOT OUTPUT ANYTHING ELSE and 8 TO 10 WORDS ONLY, starting filler messages etc.
```

Here, for each modified file, I fill in the corresponding details and save the output as the rectified message. The LLM inference is obtained in the following code snippet. The only preprocessing done on diff is to replace + with <add>, use <ide> as a prefix to unchanged lines, and to replace - with <del>, so that it is easy for the CPT5 to understand.

Now, to evaluate how good the LLM, the rectifier, and the developer are, I follow the following procedure:

For any commit C_i, let the commit message be o_i. This commit will have some modified files. So, for any modified file m_j, I first get the LLM Inference, l_j. Now, using the prompt described above, we obtain a rectified message, r_j.

Now, for each of these, o_i, l_j and, r_j We first get the embeddings, $\tilde{o_i}$, $\tilde{l_j}$ and, $\tilde{r_j}$ respectively. I consider the rectified message to be the common comparison factor.

To assess the developer's quality, I find the cosine similarity between the original commit message and the rectified message. Similarly, to assess the LLM, I find the cosine similarity between the LLM Inference and the rectified commit message.

So,

$$D_{(i,j)} = \frac{\tilde{o_i} \cdot \tilde{r_j}}{\|\tilde{o_i}\|\|\tilde{r_j}\|}, \quad L_j = \frac{\tilde{l_j} \cdot \tilde{r_j}}{\|\tilde{l_j}\|\|\tilde{r_j}\|}$$

where both these values are between -1 and 1 (1 denotes maximum alignment, and -1 denotes opposite alignment).

Finally, to evaluate how good the rectifier is, I consider the ideal scenario. Ideally, the developer should give a clear idea of all the changes made across all modified files in the commit message. So, assuming the developer is good, we can say, if $\tilde{o_i}$, and $\sum_j \tilde{r_j}$ are close, then rectifier correctly captures the meaning of the original commit message, so no rectification is needed. If they are far, then the developer is not able to correctly capture all the changes done in the modified files for that commit, so the rectifier is needed.

So, again choosing the cosine similarity as a quantitative metric,

$$R_i = \frac{\left(\sum_j \tilde{r_j}\right) \cdot \tilde{o_i}}{\left\|\sum_j \tilde{r_j}\right\| \|\tilde{o_i}\|}$$

After all the bug-fixing commits have been processed, I can plot them and set a threshold for each of these three values. From there, we can quantify the hit rate as,

$$\text{Developer hit rate} = \frac{|\{D_{(i,j)} : D_{(i,j)} \geq D_{\text{thresh}}\}|}{\sum_i \sum_j 1}$$

$$\text{LLM hit rate} = \frac{\sum_i |\{L_j : L_j \geq L_{\text{thresh}}\}|}{\sum_i \sum_j 1}$$

$$\text{Rectifier hit rate} = \frac{|\{R_i : R_i \leq R_{\text{thresh}}\}|}{\sum_i 1}$$

can then be multiplied by 100 to achieve hit rates as a percentage.

2.4 Results and Analysis

After letting the code analyze the discord.py repository (I had enabled checkpoint saves, after every 100 modified files are analyzed), and running the wordcloud function on commit messages, we see the results described in subsequent paragraphs.

```
ddf,df2 = await miner.analyze_all_pairs(verbose=True,max_workers=16,save_every=100)

[6]  96m 30.4s

...  Resuming from saved state: 1004 files processed, 1004 entries.
     Total modified-files (ordered): 3774. To process: 2770
     Processed 1010/3774 files
     Processed 1020/3774 files
     Processed 1030/3774 files
     Processed 1040/3774 files
     Processed 1050/3774 files
     Processed 1060/3774 files
     Processed 1070/3774 files
     Processed 1080/3774 files
     Processed 1090/3774 files
     Processed 1100/3774 files
     Checkpoint saved at 1100 files -> checkpoints\pairs_1100.csv, checkpoints\commits_1100.csv
     Processed 1110/3774 files
     Processed 1120/3774 files
     Processed 1130/3774 files
     Processed 1140/3774 files
     Processed 1150/3774 files
     Processed 1160/3774 files
     Processed 1170/3774 files
     Processed 1180/3774 files
     Processed 1190/3774 files
     Processed 1200/3774 files
     Checkpoint saved at 1200 files -> checkpoints\pairs_1200.csv, checkpoints\commits_1200.csv
     Processed 1210/3774 files
     ...
     Processed 1370/3774 files
     Processed 1380/3774 files
     Processed 1390/3774 files
     Processed 1400/3774 files
     Output is truncated. View as a scrollable element or open in a text editor. Adjust cell output settings...
```

Figure 2.6: Running of the commit analysis function

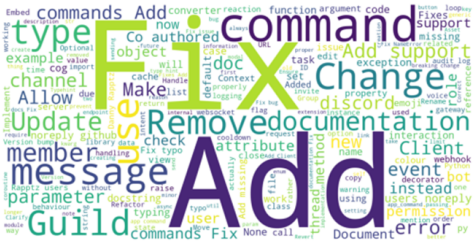

Figure 2.7: Wordcloud of commit messages.

Here I am sharing a subset of details of bug-fixing commits (refer to Table 2.1), the CSV file containing all of these commits is present on the GitHub link

Table 2.1: Sample of Bug-Fix Commits

Hash	Message	Hashes of parents	Is a merge commit ?	List of modified files
4c74523794b655bcff72ace92c7ea6f535a0e26a	Fix *versionadded* docstrings in voice connect methods	cf031f71b91854b3eeedb16004412ff5e05c9360	No	**abc.py**, **voice_client.py**
af311bff09f57ba3a769cae4298559ea14b50d50	*Change .pot files to only contain at most 5 context lines* Crowdin did not like having many comments which caused the upload to fail.	78a026aae2da0ae20669880e6c638457f7166419	No	**message.pot_t**, **builder.py**
b28a4a115e72977a9091091983a1213e91f8a5e9	Fix potentially stuck ratelimit buckets	ef06d7d9db2b868c57f45fe1252d055490ad31a9	No	**http.py**
446c5029952d2b6f466348b973888c01b0b2ef6e	Change lowercase detection to work with CJK languages. str.islower() does not properly work with characters in the Lo category, so CJK languages fail the check. Fix #7698	fd5dea4e340e69cd2527114283d8c06b2396689b	No	**commands.py**
400936df6989748c2ec391778088a07af355dbcf	Fix type for content param in HTTPClient.send_message	cdf46127ae2e1a183e5c94d9eb970bbccbed64af	No	**http.py**
cdfccfbe74ba368058702b8035f1e6378b245664	Fix description of user parameter in reaction remove event	53694724c13d0791d77fcfd425648173771555a8	No	**api.rst**

One thing to note here is that none of the bug-fix commits are merge commits, even though in the repository, there are thousands of merged pull requests. The reason is that discord.py tries to maintain a linear commit tree, so it squashes the merges.

If I had used some other repository, then of course some of the bug fix commits would be merge commits, and they would have two parents.

We know every commit has at least one modified file, and for each modified file, I store the details (refer to Fig. 2.8) (again, this is a subset of the modified files; the entire CSV is on GitHub, moreover, I have pasted in links for the diff, source code (before) and (after), to display here, because they were very large. However in the CSV they are stored as plain text).

Hash	Message	Filename	Source Code (before)	Source Code (current)	Diff	LLM Inference (fix type)	Rectified Message
72e84a1b47049d44498738458f8bf89c302a033a	Change all email/password pair examples to use token.	docs\migrating.rst	https://katb.in/ozowedariro	https://katb.in/botudarebeq	https://katb.in/erixufiviju	use token instead of password	Update examples to use token instead of email/password.
f56dbb0379eabfe99c4fdeca1fbabc0bf5b0f6df	Fix Reaction not importing. Also bump version to v0.14.1	discord__init__.py	https://katb.in/gekoziwunos	https://katb.in/ukefijiwigi	https://katb.in/gaxelejucuh	update version numbers for discord 2.0	Fix Reaction import and update version to 0.14.1.
46babe822a11ba482be864f687d2bcef66c19005	Fix bug with editing messages over private messages.	discord\client.py	https://katb.in/ticaqodinoy	https://katb.in/afodubicula	https://katb.in/ubofowuzebo	fix logs_from for private channels	Fix message editing in private channels. Handle guild_id for private messages. Update edit_message call accordingly.
73919fe1523d6b5e2e61d01b493a0800a68766eb0	Documentation fixes and version bump.	discord__init__.py	https://katb.in/ecivevoxuvu	https://katb.in/ihepocobiku	https://katb.in/wujupakinar	set version to 0.5.0	Bump version to 0.5.0. Update version and build numbers in __init__.py.
ab2512785b16d76a78f33e61a60ce90a2b225233	Handle VOICE_STATE_UPDATE websocket events. This adds a lot of new attributes into the Member class such as giving a voice_channel that the user is currently connected to. Initially there was a plan to have a voice_members attribute in the Channel class but this proved to be difficult when it came to actually removing users from the voice channel as the response would return channel_id as null. Fixes #16.	docs\api.rst	https://katb.in/apalezeholo	https://katb.in/amaxagapaza	https://katb.in/lopacuridig	add documentation for on_voice_state_update	Add on_voice_state_update function to handle voice state changes.

Figure 2.8: Table 2.1 extended with LLM inference and rectified message columns. Links are clickable in Table 2.2

Table 2.2: Source Code Comparison with Hashes

Hash	Source Code (before)	Source Code (current)	Diff
72e84a1b47049d44498738458f8bf89c302a033a	https://katb.in/ozowedariro	https://katb.in/botudarebeq	https://katb.in/erixufiviju
f56dbb0379eabfe99c4fdeca1fbabc0bf5b0f6df	https://katb.in/gekoziwunos	https://katb.in/ukefijiwigi	https://katb.in/gaxelejucuh
46babe822a11ba482be864f687d2bcef66c19005	https://katb.in/ticaqodinoy	https://katb.in/afodubicula	https://katb.in/ubofowuzebo
73919fe1523d6b5e2e61d01b493a080a68766eb0	https://katb.in/ecivevoxuvu	https://katb.in/ihepocobiku	https://katb.in/wujupakinar
ab2512785b16d76a78f33e61a60ce90a2b225233	https://katb.in/apalezeholo	https://katb.in/amaxagapaza	https://katb.in/lopacuridig

I wrote functions to plot the three metrics, $D_(i,j)$, L_j, and R_i.Based on these I decide the values of the threshold for each of these metric (that is plotted as well).

$$L_{\text{thresh}} = \text{mean}\left(\bigcup_i \{\, D_{(i,j)} \;:\; L_j > D_{(i,j)} \,\} \right) = 0.4388$$

[The threshold for LLM is the mean of similarity values for all the times when it outperforms the developer's original message.]

$$D_{\text{thresh}} = \text{mean}\left(\bigcup_i \{\, L_j \;:\; D_{(i,j)} > L_j \,\} \right) = 0.3574$$

[The threshold for Developer is the mean of similarity values for all the times when Developer outperforms the LLM (CPT5)'s inference.]

$$R_{\text{thresh}} = \text{median}(R_i) \;-\; 1.3 \times \text{median}\left(\left| R_i - \text{median}(R_i) \right| \right) = 0.6156$$

[So, if the similarity between the rectified message for all modified files in a commit is more than 1.3 times the median absolute deviation away from the developer's commit message, then the rectifier is needed.]

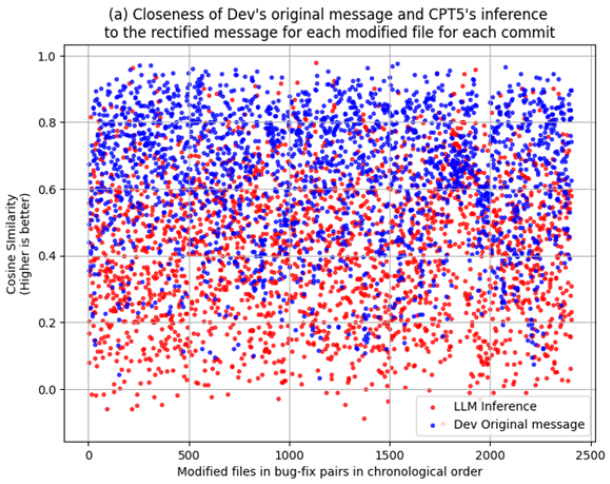

Figure 2.9: Dev's message, and CPT5's inference, as compared on rectified message.

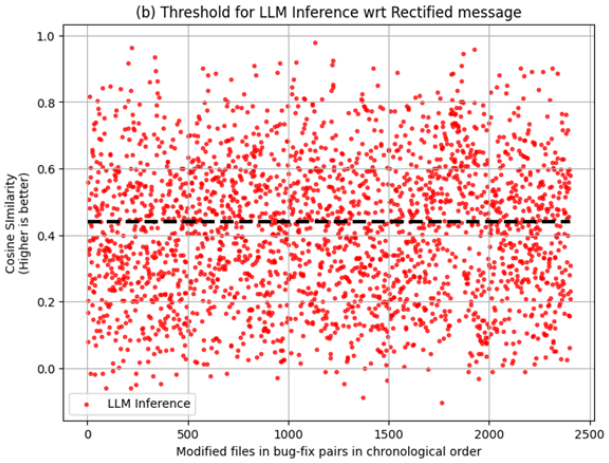

Figure 2.10: CPT5's inference, and threshold.

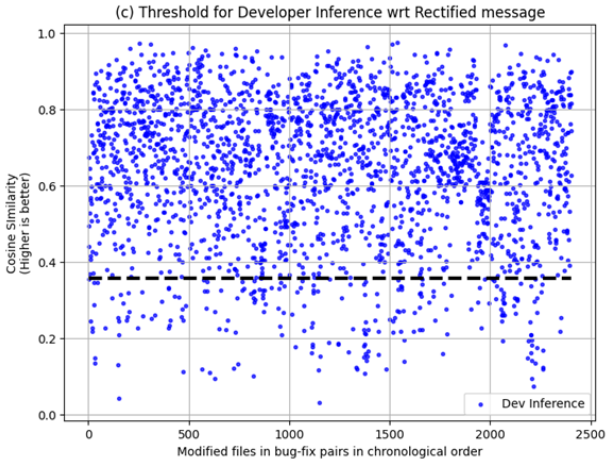

Figure 2.11: Dev's inference, and threshold.

Figure 2.12: Rectified message, and threshold.

So, finally, the hit rates come out to be:
Developer hit rate = 90.686 %
LLM hit rate = 46.611%
Rectifier hit rate = 23.407%

2.5 Discussion and Conclusion

In this lab session, I built a tool to mine a GitHub repository, analyze the bug-fixing commits, and answer really interesting questions, such as, Do developers use a precise commit message? Does using the CPT5 LLM work well, and finally, does the rectifier I created work well?

The results indicate that 90% of the bug fix commits were clearly explained by the developer. However, there were cases where the developer committed multiple files in one commit but failed to list all the changes. That's where the rectifier comes into play. Out of the 1397 commits analyzed, 23.4% of the commit messages required rectification. As far as CPT5 LLM is considered, it only performed well about 46.6% of the time, which shows that a small local model like CPT5 does not generalize well. It is helpful to generate a commit message, but it requires the user's input as well. The major difficulty I faced during this lab was of the resources to compute the embeddings, and gpt-4o-mini. I also had to use async functions

to speed up the analysis of the repository.

For future work in this, I can weigh each modified file in a particular commit by the number of lines that have changed for that file. In this way, I will ensure that the commit message stays short in length and still effectively captures the major changes that have been made (major bug fixes). I also took inspiration to find the bug-fix commits from the minecpp library available on GitHub. I would like to explore that even more, and how it works with ASTs, and how that may outperform my code.

References

[1] Puter, "Puter developer documentation," https://developer.puter.com/, accessed: 15 August 2025.

[2] "Minecpp github repository," https://github.com/SET-IITGN/MineCPP/, SET-IITGN, accessed: 15 August 2025.

[3] "Lab assignment document (on google classroom)," Google Classroom, SET-IITGN, accessed: 15 August 2025.

[4] Wikipedia contributors, "Median absolute deviation — Wikipedia, the free encyclopedia," https://en.wikipedia.org/wiki/Median_absolute_deviation, 2025, accessed: 15 August 2025.

[5] Contributors, "wordcloud: A little word cloud generator in python," https://pypi.org/project/wordcloud/, accessed: 15 August 2025.

[6] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[7] U. d. S. i. Software Institute, "Seart github search," https://seart-ghs.si.usi.ch/, accessed: 15 August 2025.

Lab 3

Multi-Metric Bug Context Analysis and Agreement Detection in Bug-Fix Commits

Github Repository for Lab 3

3.1 Introduction

This lab is a continuation of the previous lab. We add additional code quality metrics to the rectified messages database. We are instructed to use `radon`, to measure **Maintainability Index** (it is a function of Lines of Code, Cyclomatic Complexity, and other factors. It tells how easy to support and change the source code is), **Cyclomatic Complexity** (number of linearly independent paths through the code), and **Lines of Code** (I have excluded blank spaces, so basically, I have counted the number of source lines of code). Then we were to use `SacreBLEU` (a python library), and `CodeBERT` (from Hugging Face) to analyze the difference in BLEU and BERT scores of the source code after the commit and source code before the commit. This was then used to find a threshold, to find if there was a Major Fix or a Minor Fix in a modified file of any commit.

3.2 Setup and Tools

I am using radon, the `microsoft/codebert-base` model, `sacrebleu` model for embedding generation, and rectified_commits.csv (abbreviated as **rect database**) from the previous lab. I will also be using a virtual environment, so that my global packages do not interfere with torch and, transformers libraries. and `python --version` gives `Python 3.12.4`. Other versions are shown in the images.

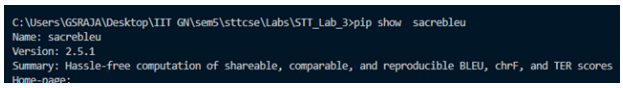

Figure 3.1: sacrebleu version

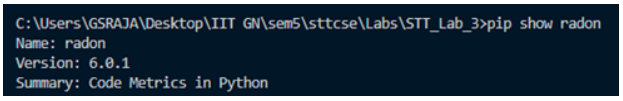

Figure 3.2: radon version

3.3 Methodology and Execution

I start with the trivial data analysis, such as finding the total number of commits and files. For total commits I just check the unique number of hashes in the rect database. And for total number of files, I check the unique filenames in the rect database. (One thing to note is, I do not store just the filename, I store the file path also). Since each row in the database is a file, to get average files per commit, we can simply get a frequency distribution of the hashes, and take the mean.

```
total_commits = len(modified_files_dataset['hash'].unique())
total_files = len(modified_files_dataset['filename'].unique())
average_files_per_commit = modified_files_dataset['hash'].value_counts().mean()
```

Next task is to get the distribution for the fix type, so, upon inspecting the LLM Inference, almost always the first word directly hints at what the fix type should be, So I make five categories of fixes, namely add, remove, refactor, fix, update, misc. And following is the first word assignment to each category:

```
term_to_category = {
    "update": "update", "upgrade": "update", "improve": "update", "apply": "update",
     "set": "update",
    "use": "update", "simplify": "update", "replace": "update", "convert": "update",
    "reorganize": "update", "fix": "fix", "escape": "fix", "ignore": "fix", "handle
    ": "fix",
    "reset": "fix", "add": "add", "support": "add", "call": "add", "poll": "add",
    "remove": "remove", "delete": "remove", "disconnect": "remove", "stop": "remove
    ",
    "terminate": "remove", "rename": "refactor", "return": "refactor", "pass": "
    refactor",
    "deprecate": "refactor", "send": "misc", "check": "misc", "raise": "misc"
    }
```

With this I can easily check what the fix type of my commit is. In a similar manner I also find the frequency distribution for the file extensions, which tells me what are the extensions of most modified files. Now I proceed to updating the rect database, by first computing the structural metrics using radon. I run the radon's analyze and visit methods to find out the Lines of Code, Cyclomatic Complexity and Maintainability Index. One thing to note is, that incase a new file is created, or a file is deleted then one of the Source Code Before, or Source Code After values will be NaN. So to handle them, I simply return 0 as each of the metrics. These metrics are computed for both the Source Code Before commit and the Source Code After commit. Then the differences are appended to the rect database.

Moreover, radon only works for python files, so, for files which have codes other than what radon can analyze, the CC and the MI values are set to 0, however, the lines of code can still be computed.

Next task is to compute the Sematic Similarity (using CodeBERT) and the Token Similarity (using BLEU). So I load the CodeBert Model, generate embeddings for the Source Code Before commit and Source Code After Commit, and then find the cosine similarity between them. Since I was running this on my GPU, so I had to split the rect database rows in batches.

Similarly, I use the BLEU score to find the token similarity between Source Code Before and Source Code Current.

```
def radon_metrics(code):
    if isinstance(code,float):
        return 0,0,0
    try:
        raw = analyze(code)
        sloc = raw.sloc
        mi = mi_visit(code, True)
        cc_values = [r.complexity for r in cc_visit(code)]
        cc_total = int(sum(cc_values)) if cc_values else 0
    except SyntaxError:
        mi = 0
        cc_total = 0
        sloc = len(code.splitlines())
    return mi,cc_total,sloc
```

Figure 3.3: Radon metrics function

```
for i in range(0, len(before_code_list), batch_size):
    batch_before = before_code_list[i : i + batch_size]
    batch_current = current_code_list[i : i + batch_size]

    embeddings_before = generate_embeddings(batch_before)
    embeddings_current = generate_embeddings(batch_current)

    bert_cos_similarity = F.cosine_similarity(embeddings_before, embeddings_current)
    all_similarities.extend(bert_cos_similarity.cpu().numpy())
✓ 6m 59.7s
```

Figure 3.4: Computing similarity for BERT.

```
def compute_bleu(row):
    if pd.isna(row['source_code_before']) or pd.isna(row['source_code_current']):
        return np.nan
    before_ls = [row['source_code_before']]
    after = row['source_code_current']
    return sacrebleu.sentence_bleu(after, before_ls).score
```

Figure 3.5: Compute BLEU function

Then I plotted these values as a histogram with y axis in log scale, to figure out a good threshold, for which, I can classify the commit as a Minor Fix or a Major Fix for a particular modified file. Lastly, I

added one more column, Classes Agree, where if both Semantic Similarity is less than semantic threshold (Major Fix) and Token Similarity is less than token threshold (Major Fix), or vice-versa then both classes agree with each other.

3.4 Results and Analysis

The total number of commits are: 1402
The total (unique) number of modified files are: 146
The number of average files per commit is: 1.715
I categorize each row of the rect database, on the basis of first word of LLM Inference into six bug-fix type categories, and the histogram for the same is shown in Fig. 3.6.

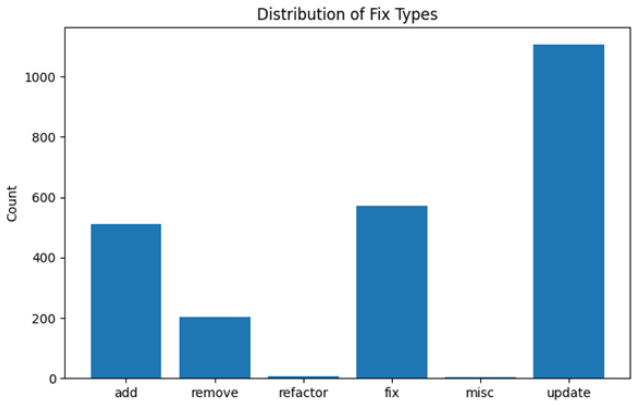

Figure 3.6: LLM inference fix types

This shows that most of the bug-fix commits are update in nature, that is, apart from fixing a bug in that commit, the developers update other code too. This category is followed by the fix category, where the developers explicitly fix a described issue. Next category is add, here developers, apart from fixing a bug, ad additional functionality. Lastly we have the remove category, here it may be the case that bug causing code is removed, or deprecated. The last two categories, refactor and misc, might be false positives, as the rect database might have some rows which are not bug-fix commits (because the developer might have used one of the 52 keywords in some context other than bug fixing (Refer to Lab 2)).

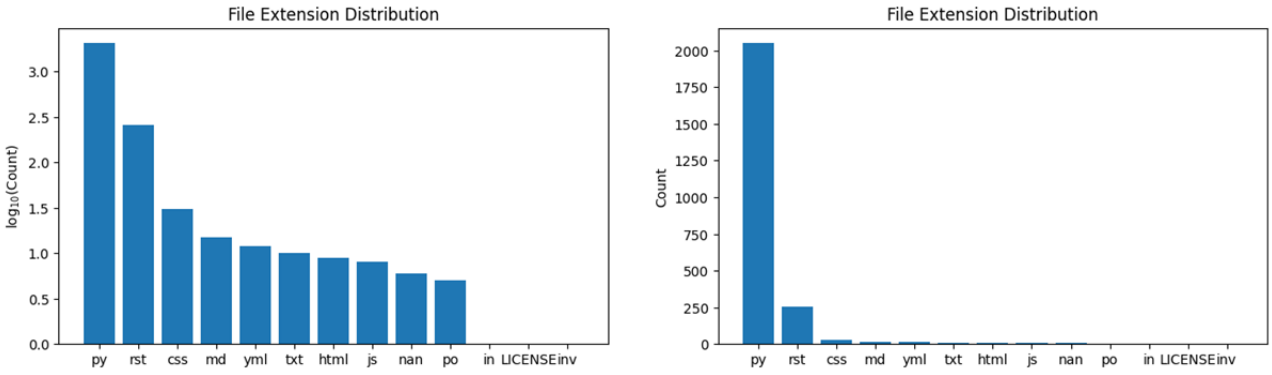

Figure 3.7: File extension distribution, with log scale and linear scale

The two graphs in Fig. 3.7, depict the same file extension distribution, the first graph is in log scale for y axis to promote visibility of the less frequent file extensions. The most common file extension for modified files, with over 2000 count is `.py`. which is preferred as the radon analysis works only with python files. The second most frequent extension is `.rst`, which is for restructured text, and it is used in the files which have documentation of the analyzed library.

After running the radon analysis for all the modified files, (a few of them will error out, as they have extension which is not `.py`), we obtain the subsequent plots.

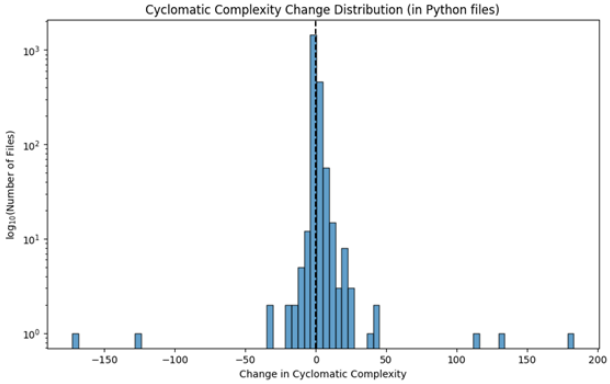

Figure 3.8: CC change distribution

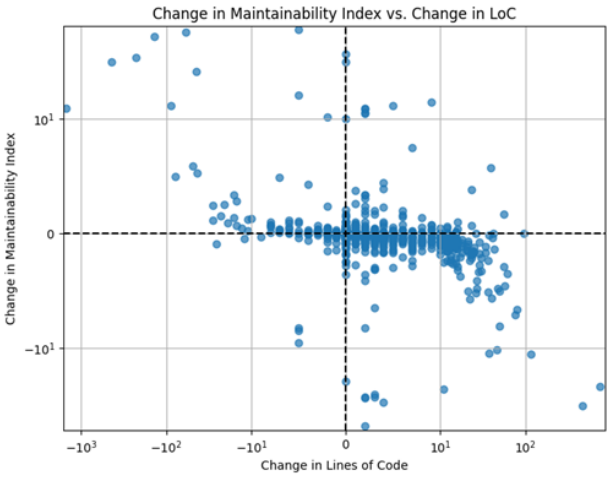

Figure 3.9: MI vs LOC distribution

Fig. 3.8 shows a almost gaussian like distribution (slightly skewed to right), almost equal number of files have an increased and a decreased complexity. Most of the files (1000) have no significant complexity change. But overall we can say that the complexity has slightly increased (distribution is skewed to right, mean of cc change is 0.73 which is more than 0).

Fig. 3.9 indicates how maintainability is affected by change in number of lines of code. Note that both the axes are in log scale.

We can observe a clear negative correlation with change in maintainability index and change in lines of code. As more code is introduced, the maintainability index goes down, and as code is removed, the maintainability index goes up.

Similar trend is observed if we plot change in maintainability index vs change in code complexity (Fig. 3.10).

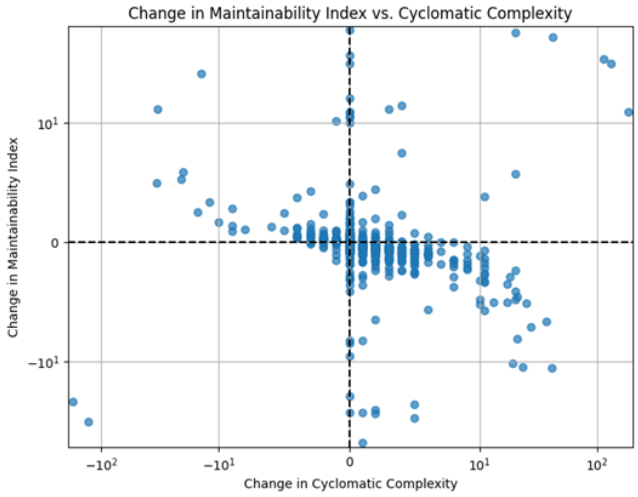

Figure 3.10: MI change vs CC distribution

Then I used CodeBERT and sacrebleu to compute and plot the BERT and BLEU scores, and on that basis decide the thresholds to classify a fix as Major or Minor. One thing to note is, the repository I chose is in continuous development, and is backwards compatible, So I do not think there would be many Major fixes, hence I will choose conservative thresholds, which favor Minor fixes.

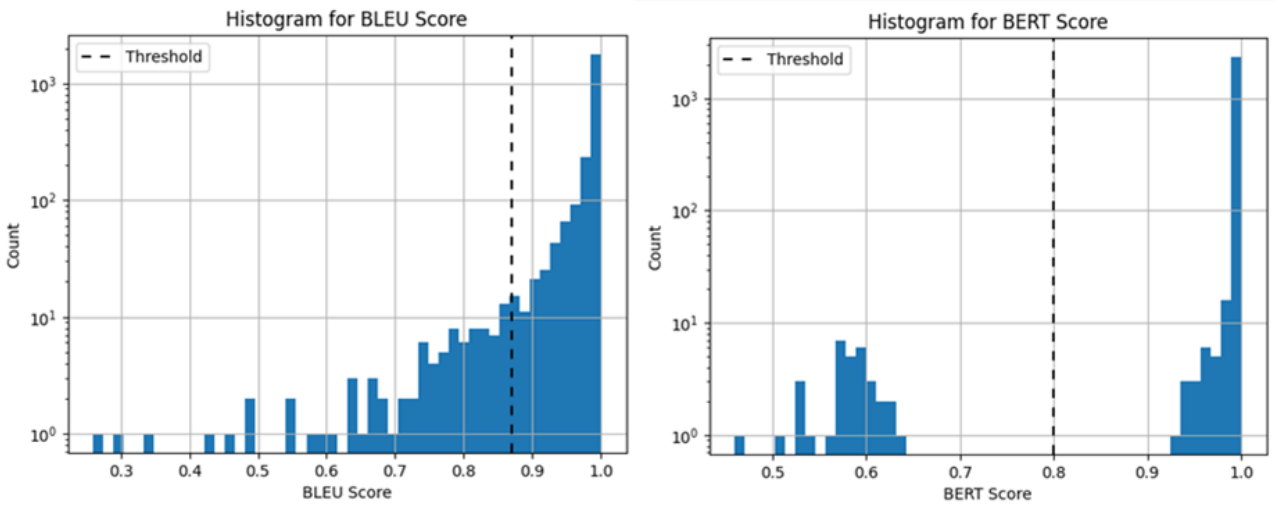

Figure 3.11: Threshold for BLEU and BERT

For BERT, we can clearly see a bimodal distribution (Fig. 3.11). This indicates a clear boundary for classifying a fix as a Major Fix or a Minor Fix. So for BERT I chose a threshold of 0.8. However, for BLEU there is no such boundary but the shape of histogram changes near 0.9, but a better way would be to set the threshold to the 4th quantile, which comes out to be 0.87. After these commits are classified into Major or Minor on the basis of BERT and BLEU scores, I make a confusion matrix, to see how they both agree on whether a commit is a major fix or a minor fix.

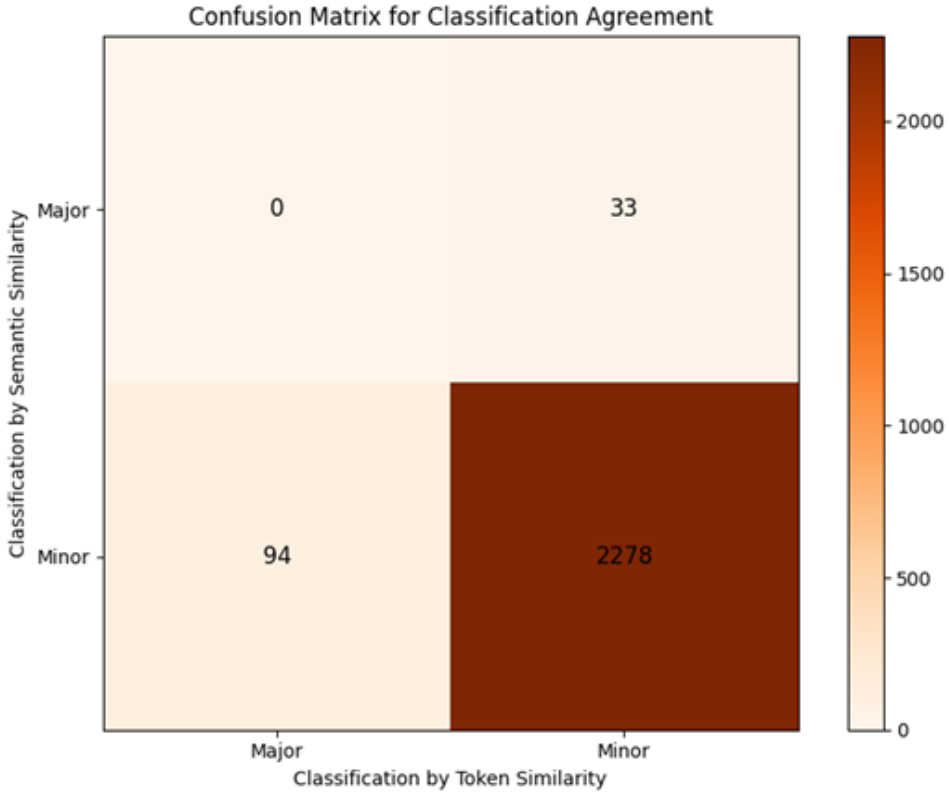

Figure 3.12: Confusion Matrix using BLEU and BERT

Fig. 3.12 shows that minor fixes are classified really well. The overall agreement rate is

$$\frac{2278}{2405} \times 100 = 94.72\%$$

The most common type of disagreement is CodeBERT says Minor Fix and BLEU says Major Fix. This

indicates that most of the text shares similar words (high token similarity) but their underlying meaning is different (low semantic similarity).

For cases where BLEU says Minor Fix and CodeBERT says Major Fix, here the words are different but the meaning remains the same.

Both CodeBERT and BLEU do not agree on even a single Major Fix. This shows the fundamental difference between checking similarity using both the models.

One captures the meanings and the other captures the semantics. However, it could also be the case that the repository (python library) I chose, discord.py, has only incremental changes, and there are no significant breaking changes (It can be confirmed that this library is backwards compatible.

3.5 Discussion and Conclusion

In this lab session, I extended the previous lab's dataset, and added structural metrics to better analyze the bug fix commits. I also capture the code complexity change, maintainability index change, and source lines change using the radon library. Next I use CodeBERT and sacrebleu to compute BERT and BLEU scores, and apply thresholds derived from the patterns in data, and classify each bug-fix pair as a Major Fix or a Minor Fix. Lastly, I plot the confusion matrix and analyze where both metrics agree or disagree.

The results indicate that, as the number of lines of code increase, the maintainability index decreases which is in turn an effect of increase in code complexity. Note that these structural metrics were only available for python files. Between BERT and BLEU, the former more accurately captures the semantics and gives a clear distinct threshold, below which I call the commits as Major fix. This is not the case with BLEU. It is hard to capture the notion of Major fix in BLEU, unless we know the developer's philosophy that there will always be incremental changes, and no major code change between commits. Lastly both these agree well on Minor Fixes, but do not generalize to Major Fixes, as same code can be written with less or different tokens (leading to low BLEU) but having similar semantic meaning (leading to high BERT).

I did not face much difficulty in this lab, as most of the tedious work of dataset creation was done in the previous lab. I would, however, like to run this analysis on some other repository to see how well these techniques generalize.

References

[1] Radon-Developers, "Radon documentation," https://radon.readthedocs.io/en/latest/intro.html, accessed: 15 August 2025.

[2] Shouvick-Mondal, "Lab assignment document (on google classroom)," Google Classroom, accessed: 15 August 2025.

[3] Matt-Post, "Sacrebleu github repository," https://github.com/mjpost/sacrebleu/blob/master/README.md, accessed: 15 August 2025.

[4] Microsoft, "Codebert github repository," https://github.com/microsoft/CodeBERT/blob/master/README.md, accessed: 15 August 2025.

Lab 4

Exploration of different diff algorithms on Open-Source Repositories

Github Repository for Lab 4

4.1 Introduction

In this lab, we explored how different diff algorithms in Git, specifically **Myers** and **Histogram**, produce different outputs when applied to real-world open-source repositories. Diff algorithms play a important role in version control systems such as git, as they determine how changes in **code** and **other** files (markdown files, test files and, license file) are represented. Even small differences in the algorithm can impact how developers interpret modifications. For this lab, I selected three open-source repositories and used `pydriller` to analyze their commit histories. Each repository was partitioned into code artifacts and non-code artifacts. The purpose of separation is to study if diff algorithms behave differently depending on the type of file being analyzed. I also heavily refer to the paper by Yusuf Nugroho, on different diff algorithms in Git. I generated both diff outputs (Myers and Histogram) for all modified files in all commits, and compared them to identify mismatches. Lastly the mismatches were analyzed across file type categories to understand the effectiveness of both the diff algorithms.

4.2 Setup and Tools

I am using `pydriller` to obtain data of commit history, histogram diff and myers diff for the chosen repositories, `pandas` library for dataframe processing, `matplotlib` for plotting. In my environment, `python --version` gives Python 3.12.4, the pydriller version is shown in Fig. 4.1. SEART Search Engine https://seart-ghs.si.usi.ch/ is used to search and filter the repositories based on my criteria.

```
C:\Users\GSRAJA\Desktop\IIT GN\sem5\sttcse\Labs\STT_Lab_4>pip show pydriller
Name: PyDriller
Version: 2.8
Summary: Framework for MSR
Home-page: https://github.com/ishepard/pydriller
Author: Davide Spadini
Author-email: spadini.davide@gmail.com
License: Apache License
```

Figure 4.1: Pydriller version

4.3 Methodology and Execution

I first searched through the SEART search engine, three repositories, which are mainly based on python, to use for analysis during this lab. I chose the following repositories: `unclecode/crawl4ai`, `datalab-to/marker`,

and `zylon-ai/private-gpt`. Fig. 4.2 shows the search criteria I used, I ensured the following things:

- Major language should be python.
- It should have a minimum of 20000 stars.
- It has a license file.
- It has a wiki (having a wiki increases the chances that the repository has a **tests** folder).

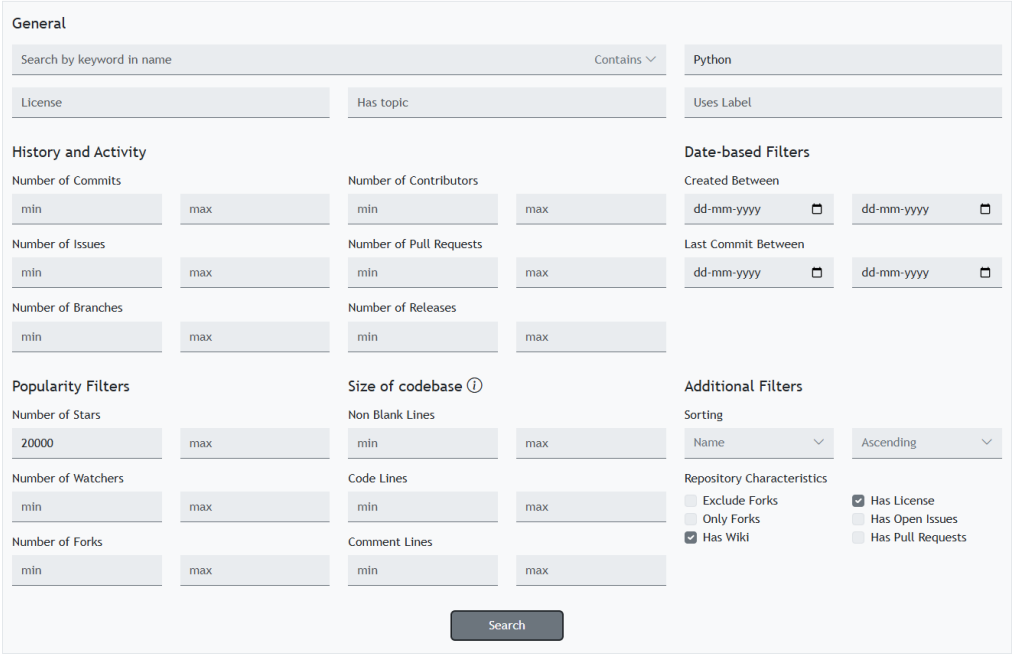

Figure 4.2: SEART search engine (criteria for searching)

I use the following procedure to extract information from each of the three repositories:

1. Iterate through the commits in the repository, with the parameters of `skip_whitespaces=True`, and `zip(Repository(...,histogram_diff=True),Repository(...,histogram_diff=False)` to access both the myers diff and histogram diff in same loop.
2. If the length of modified files in this commit is 0, then continue, as this would be a merge commit.
3. For each file in the modified files, store the following details:

```
    {
        "old_file_path": mf.new_path,
        "new_file_path": mf.new_path,
        "hash": commit.hash,
        "parents_hash": commit.parents,
        "message": commit.msg,
        "diff_myers": mf.diff,
        "diff_hist": mf_hist.diff,
        "myers_added_lines": mf.diff_parsed["added"],
        "myers_deleted_lines": mf.diff_parsed["deleted"],
        "hist_added_lines": mf_hist.diff_parsed["added"],
        "hist_deleted_lines": mf_hist.diff_parsed["deleted"],
    }

```

To check if Myers and Histogram diff resulted in the same or the different outputs, I use the same methodology as described in the paper [1]. Fig. 4.3 shows the screenshot from that paper.

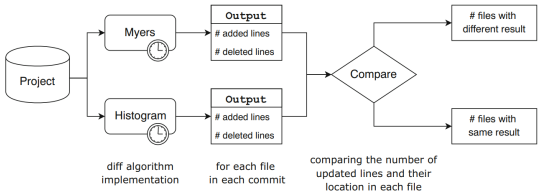

Fig. 11 Overview of the metrics collection procedure

Figure 4.3: Screenshot from [1].

Using this, we say there is no discrepancy if #lines added by Myers = #lines added by Histogram as well as #lines removed by Myers = #lines removed by Histogram.

All the three repositories which I chose have LICENSE file, few markdown files, and test files under a folder name `tests/`. So I make bitmasks for these conditions on path of the modified file, and with that I can filter out the **code** and the **non code** files (Fig. 4.4).

```
def statistics(df):
    md_files_mask = df['new_file_path'].apply(lambda x: False if x == None else x.endswith('.md'))
    license_files_mask = df['new_file_path'].apply(lambda x: False if x == None else x.lower() == 'license')
    tests_files_mask = df['new_file_path'].apply(lambda x: False if x == None else x.startswith('tests\\'))
    source_files_mask = ~(md_files_mask | license_files_mask | tests_files_mask)
    return {
        'md':df[md_files_mask]['Discrepency'].value_counts(),
        'license':df[license_files_mask]['Discrepency'].value_counts(),
        'tests':df[tests_files_mask]['Discrepency'].value_counts(),
        'source':df[source_files_mask]['Discrepency'].value_counts()
    }
```

Figure 4.4: Bitmasks for code and non code files.

To evaluate programatically which diff algorithm is better, I again refer to the paper given [1],

```
1 The diff results might show different change region with a contiguous list of
    deleted and added lines that is called as a change hunk (Ray et al, 2015).
2
3 We expect that a set of changing operations done by developers can be represented by
    change hunks.
```

Using this ideology, I evaluate both diff algorithms by treating contiguous added or deleted lines as change hunks. Here I assume that a developer's single logical edit often appears as one hunk.

1. For each modified file and for each algorithm (Myers and Histogram) I extract the change hunks separately for added lines and for deleted lines.
2. Ignore 1-line hunks when computing statistics, because one-line differences are common noise and do not reliably represent an operation.
3. Compute:
   (a) The median length of the change hunks (combined over added and deleted hunks)
   (b) The average number of change hunks approximated by,

$$\frac{\#\text{ lines changed}}{\text{median hunk length}}$$

   where # lines changed is computed separately for added and deleted lines and then averaged.
4. The average-number measure estimates how many logical hunks the changed lines are split into.
5. **If** $(\text{median hunk length})^{\text{algo1}} < (\text{median hunk length})^{\text{algo2}}$, then algo1 captures smaller hunks (more fine grained developer operations).
6. **Else If** there is a tie, then I use the maximum hunk length rather than the median.

7. I prefer the algorithm that overall produces smaller median hunk lengths and higher average number of hunks.

4.4 Results and Analysis

4.4.1 Private-gpt

Fig. 4.5 shows, in a tabular format, the extracted details for each commit.

	old_file_path	new_file_path	hash	parents_hash	message	diff_myers	diff_hist
24	chroma_preference.py	chroma_preference.py	8c6a81a07fc9c800d53f62a33f5ae3b5247a22a6	['60225698b6bd46f905a9525253a2974980ade0ca']	Fix: Disable Chroma Telemetry\n\nOpts-out of a...	@@ -0,0 +1,11 @@\n+from chromadb.config import...	@@ -0,0 +1,11 @@\n+from chromadb.config import...
413	fern\fern.config.json	fern\fern.config.json	f339f7608c5b1b7592b5075ac4340fd0ce40da7f	['ff7e2bc9dda7a694a6e940313261f888be902122']	Move Docs to Fern (#1257)	@@ -0,0 +1,4 @@\n+{\n+ "organization": "priva...	@@ -0,0 +1,4 @@\n+{\n+ "organization": "priva...
507	settings.yaml	settings.yaml	56af625d71afc62f4830de07079aea2e1c938e62	['b7ca7d35a0c181bfcf79aac00c01e66e50c18f97']	Fix the parallel ingestion mode, and make it a...	@@ -1,3 +1,6 @@\n+# The default configuration ...	@@ -1,3 +1,6 @@\n+# The default configuration ...
748	settings.yaml	settings.yaml	966af4771dbe5cf3fdf554b5fdf8f732407859c4	['d13029a046f6e19e8ee65bef3acd96365c738df2']	fix(settings): enable cors by default so it wi...	@@ -5,7 +5,7 @@ server:\n env_name: ${APP_EN...	@@ -5,7 +5,7 @@ server:\n env_name: ${APP_EN...
793	fern\docs\pages\installation\concepts.mdx	fern\docs\pages\installation\concepts.mdx	4523a30c8f004aac7a7ae224671e2c45ec0cb973	['01b7ccd0648be032846647c9a184925d3682f612']	feat(docs): update documentation and fix previ...	@@ -8,18 +8,25 @@ It supports a variety of LLM...	@@ -8,18 +8,25 @@ It supports a variety of LLM...

Figure 4.5: Details for each modified file in all commits.

All the table images, I have added in this document, are a sample of the actual csv files which are available on the Github Link. After processing the discrepancies, we make a new csv with the discrepancy column, as shown in Fig. 4.6.

	old_file_path	new_file_path	hash	parents_hash	message	diff_myers	diff_hist	Discrepency
30	requirements.txt	requirements.txt	85528db7431d57a2bb2ca6e3e8d0fb37a668fd82	['918b384e38f34a709c1f96ccfa81bcbb67e10d0e']	Update langchain to 0.0.166\n\nTested:\n\nRele...	@@ -1,4 +1,4 @@\n-langchain==0.0.165\n+langcha...	@@ -1,4 +1,4 @@\n-langchain==0.0.165\n+langcha...	False
70	ingest.py	ingest.py	7844553ca1fd39aa5962795f2209e550322212b1	['42046c5ec0322e82e545cd43d96e0c99d2fc4287']	Implement a way of ingesting more documents\nM...	@@ -24,6 +24,16 @@ from langchain.docstore.doc...	@@ -24,6 +24,16 @@ from langchain.docstore.doc...	False
113	ingest.py	ingest.py	28537b6a84ce51487f477e2619d1c177e08f399b	['b1057afdf8f65fdb10e4160adbd8462be0c08271']	Better error message if .env is empty/does not...	@@ -24,11 +24,12 @@ from langchain.text_splitt...	@@ -24,12 +24,13 @@ from langchain.text_splitt...	True
147	README.md	README.md	51cc638758f68032bd971ed7d80fa5e284f185de	['78d1ef44adea1b72235a4cb603bbf0e4d9033d10']	Next version of PrivateGPT (#1077)\n\n* Docker...	@@ -1,152 +1,158 @@ privateGPT\n-Ask quest...	@@ -1,152 +1,158 @@ privateGPT\n-Ask quest...	True
159	poetry.lock	poetry.lock	51cc638758f68032bd971ed7d80fa5e284f185de	['78d1ef44adea1b72235a4cb603bbf0e4d9033d10']	Next version of PrivateGPT (#1077)\n\n* Docker...	@@ -1,99 +1,110 @@\n # This file is automatica...	@@ -1,99 +1,110 @@\n # This file is automatica...	True

Figure 4.6: Sample of commits, after processing the discrepancies

Overall we see that, 1.7% of the modified files have a discrepancy (Fig. 4.7). Fig. 4.8 shows the discrepancy across the markdown, license, tests, and the source code files.

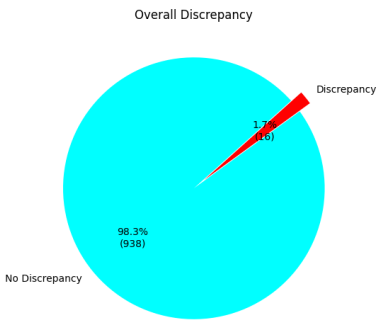

Figure 4.7: Pie chart for overall discrepancy.

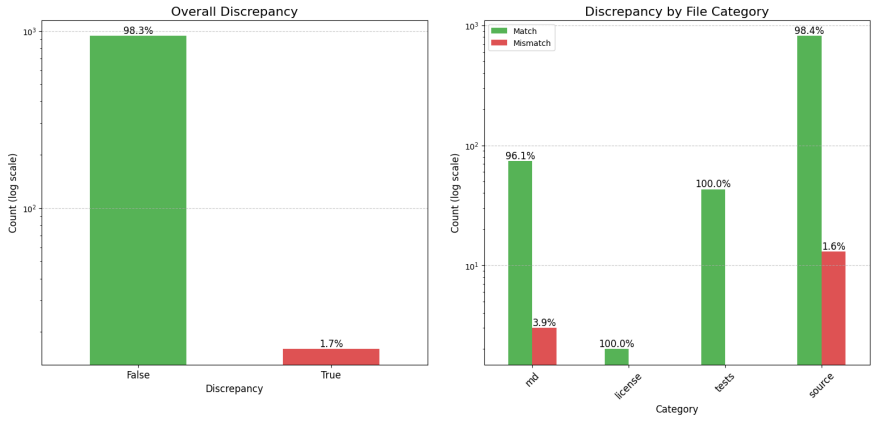

Figure 4.8: Bar graph showing discrepancy across categories of files.

```
#Mismatches for Source Code files   : 13
#Mismatches for Test Code files     : 0
#Mismatches for README files        : 3
#Mismatches for LICENSE files       : 2
```

Comparing both the diff algorithms based on the approach described before, we get the following results.

```
Hist is better in 12 instances (1.8%)
Myers is better in 11 instances (1.6%)
ties 646 (96.6%)
```

4.4.2 Crawl4ai

I will proceed with the graphs and diff algorithm comparison, as the csv file for details of commits and discrepancy analysis is available on GitHub.

```
#Mismatches for Source Code files  : 77
#Mismatches for Test Code files    : 2
#Mismatches for README files       : 56
#Mismatches for LICENSE files      : 2
```

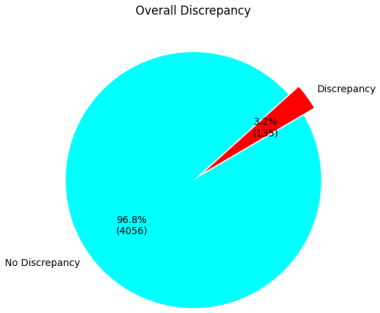

Figure 4.9: Pie chart for overall discrepancy.

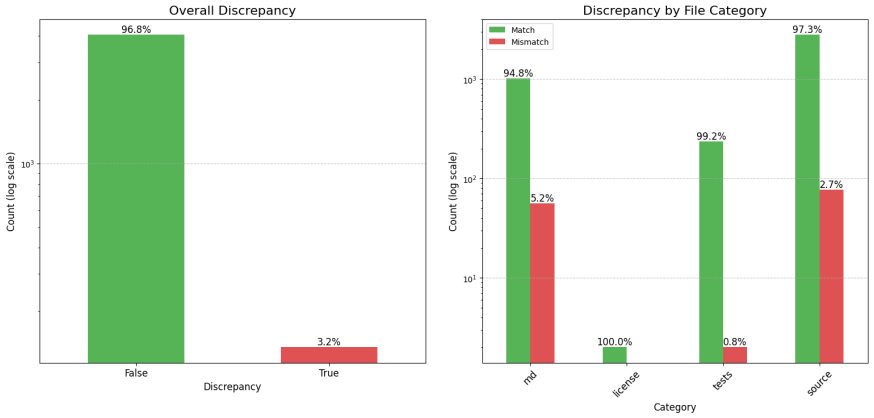

Figure 4.10: Bar graph showing discrepancy across categories of files.

We observe that, 3.2% of the modified files have a discrepancy (Fig. 4.9). Fig. 4.10 shows the discrepancy across the markdown, license, tests, and the source code files.

Comparing both the diff algorithms based on the approach described before, we get the following results.

```
Hist is better in 116 instances (3.8%)
Myers is better in 79 instances (2.6%)
ties 2878 (93.7%)
```

4.4.3 Marker

I will proceed with the graphs and diff algorithm comparison, as the csv file for details of commits and discrepancy analysis is available on GitHub.

```
#Mismatches for Source Code files  : 73
#Mismatches for Test Code files    : 2
#Mismatches for README files       : 15
#Mismatches for LICENSE files      : 2
```

We observe that, 2.1% of the modified files have a discrepancy (Fig. 4.11). Fig. 4.12 shows the discrepancy across the markdown, license, tests, and the source code files.

Comparing both the diff algorithms based on the approach described before, we get the following results.

```
Hist is better in 71 instances (2.7%)
Myers is better in 62 instances (2.3%)
ties 2516 (95.0%)
```

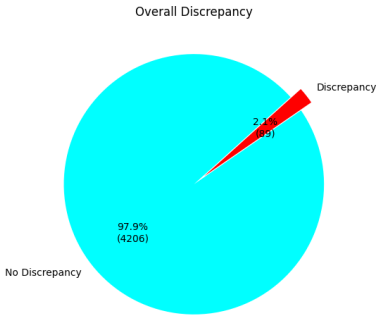

Figure 4.11: Pie chart for overall discrepancy.

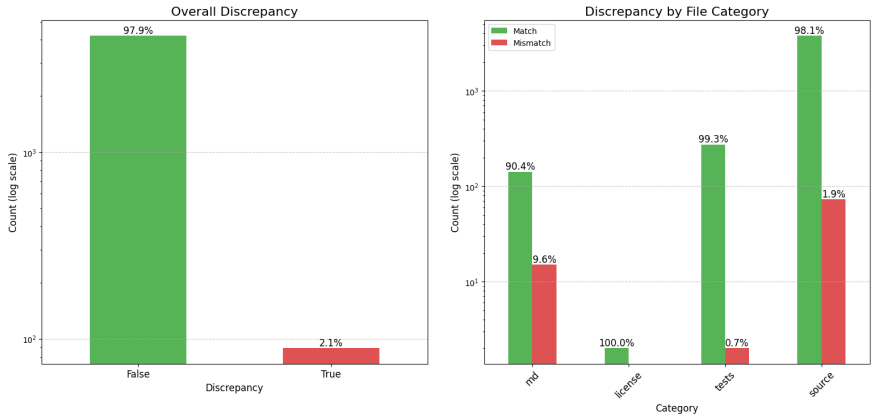

Figure 4.12: Bar graph showing discrepancy across categories of files.

4.5 Discussion and Conclusion

In this lab I compared Myers and Histogram diff algorithms on three real-world repositories to see which algorithm produces diff that better reflect developers' edits. I have assumed (motivated by [1]) that a developer's single logical change tends to appear as a contiguous run of changed lines (a change hunk). I simply find the median hunk length and the average number of code hunks, for both Myers and Histogram diff algorithms, and whichever is has smaller median hunks and larger number of hunks is treated as better aligned with developer's intent.

The three repositories show low overall discrepancy rates between the two algorithms: 1.7%, 3.2% and 2.1% of modified files had different added/deleted counts, respectively, which is consistent with the [1]'s findings (1.7% to 8.2%).

Overall, while comparing the diff algorithms, Histogram wins more often than Myers. Across the three datasets I observed 199 Histogram-wins versus 152 Myers-wins (the remainder were ties). That imbalance supports the conclusion from [1] that Histogram is generally better at producing hunks that match developer edits in practice, although the effect size is small. This shows that the choice between Myers and Histogram usually does not change the results, since they agree most of the time. When discrepancies do occur, Histogram is often better. However, there are limitations: I assumed that developers' logical edits are contiguous. That assumption fails when developers interleave unrelated edits in the same commit or when edits are semantically equivalent (for example, `[x for x in range(10)]` vs. `list(range(10))`). We could combine the methodology from the previous lab (refer to Lab 3) for detecting semantic equivalence with our current approach to determine the better diff algorithm.

References

[1] Y. S. Nugroho, H. Hata, and K. Matsumoto, "How different are different diff algorithms in git?" *Empirical Software Engineering*, vol. 25, no. 1, pp. 790–823, Jan 2020. [Online]. Available: https://doi.org/10.1007/s10664-019-09772-z

[2] Shouvick-Mondal, "Lab assignment document (on google classroom)," Google Classroom, accessed: 28 August 2025.

[3] "Git documentation," https://git-scm.com/docs, accessed: 2025-08-28.

[4] D. Spadini, M. Aniche, and A. Bacchelli, *PyDriller: Python Framework for Mining Software Repositories*, 2018.

[5] U. d. S. i. Software Institute, "Seart github search," https://seart-ghs.si.usi.ch/, accessed: 28 August 2025.