

# ASSIGNMENT 1

## SOFTWARE TOOLS AND TECHNIQUES FOR CSE

Sai Keerthana Pappala

Roll No: 23110229

September 8, 2025

## Table of Contents

|                                     |    |
|-------------------------------------|----|
| <b>0.1 Laboratory Session 1</b>     | 3  |
| Introduction                        | 3  |
| Tools                               | 3  |
| Setup                               | 3  |
| Methodology and Execution           | 3  |
| Results and Analysis                | 8  |
| Discussion and Conclusion           | 9  |
| References                          | 9  |
| <br><b>0.2 Laboratory Session 2</b> | 10 |
| Introduction                        | 10 |
| Tools                               | 10 |
| Setup                               | 10 |
| Methodology and Execution           | 11 |
| Results and Analysis                | 12 |
| Discussion and Conclusion           | 12 |
| References                          | 13 |
| <br><b>0.3 Laboratory Session 3</b> | 14 |
| Introduction                        | 14 |
| Tools                               | 14 |
| Setup                               | 14 |
| Methodology and Execution           | 15 |
| Results and Analysis                | 15 |
| Discussion and Conclusion           | 16 |
| References                          | 17 |
| <br><b>0.4 Laboratory Session 4</b> | 18 |
| Introduction                        | 18 |
| Tools                               | 18 |
| Setup                               | 18 |
| Methodology and Execution           | 19 |
| Results and Analysis                | 19 |
| Discussion and Conclusion           | 20 |
| References                          | 21 |

## 0.1 LABORATORY SESSION 1

### INTRODUCTION

This lab aimed to teach the importance of Version Control Systems like Git, including basic operations such as initializing a repository, committing changes, and syncing with GitHub. We also learned to set up a Python linting workflow using GitHub Actions to ensure code quality and practice automated checks.

### TOOLS AND ENVIRONMENT

| Tool        | Version                 |
|-------------|-------------------------|
| Git         | 2.46.0                  |
| OS          | windows10               |
| code editor | VS code                 |
| Github      | already created my repo |
| Python      | 3.13.5                  |

### SETUP

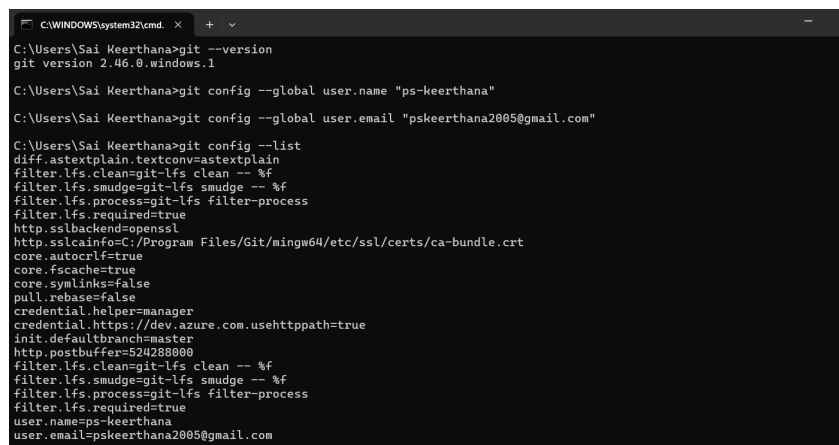
- I first installed Git and verified the installation by checking the version .Then, I configured Git with my username and email using

```
git config --global user.name "ps-keerthana"
git config --global user.email "pskeerthana2005@gmail.com"
```

and confirmed the configuration with

```
git config --list
```

- After that, I signed in to GitHub and completed the two-factor authentication process, which involved using both my password and the GitHub mobile app key. With all of this setup completed, I was ready to begin the lab tasks..



```
C:\WINDOWS\system32\cmd.exe
C:\Users\Sai Keerthana>git --version
git version 2.46.0.windows.1

C:\Users\Sai Keerthana>git config --global user.name "ps-keerthana"

C:\Users\Sai Keerthana>git config --global user.email "pskeerthana2005@gmail.com"

C:\Users\Sai Keerthana>git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fsckcache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
http.postbuffer=52428800
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
user.name=ps-keerthana
user.email=pskeerthana2005@gmail.com
```

Figure 1: setup

## METHODOLOGY AND EXECUTION

- Created a repository on public GitHub and named STT\_1\_1. Added a license and a README.md file.
- Cloned this GitHub repository to local system by Opening Git Bash terminal and ran:

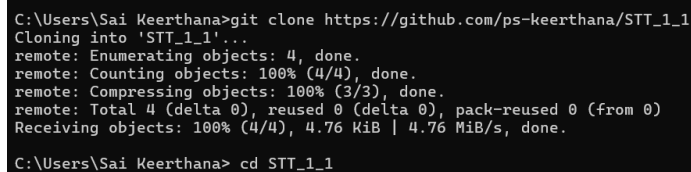
```
git clone https://github.com/ps-keerthana/STT_1_1
```

This created a local copy of the repository on my system in the path C:/Users/Sai Keerthana/STT\_1\_1

- Navigated into the local repository folder and Changed directory using

```
cd STT_1_1
```

and verified README.md exists.



```
C:\Users\Sai Keerthana>git clone https://github.com/ps-keerthana/STT_1_1
Cloning into 'STT_1_1'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (4/4), 4.76 KiB | 4.76 MiB/s, done.
C:\Users\Sai Keerthana> cd STT_1_1
```

Figure 2: Cloning of the repo

- Added a Python file to the local repository and Created my\_code.py with  $\geq 30$  lines of code for To-Do List program to add, show, and remove tasks.

```
C: > Users > Sai Keerthana > STT_1_1 > my_code.py > ...
1  tasks = []
2
3  def show_tasks():
4      if not tasks:
5          print("No tasks in the list.")
6      else:
7          print("Your Tasks:")
8          for i, task in enumerate(tasks, 1):
9              print(f"{i}. {task}")
10
11 def add_task():
12     task = input("Enter a new task: ")
13     tasks.append(task)
14     print(f"Task '{task}' added.")
15
16 def remove_task():
17     show_tasks()
18     if tasks:
19         try:
20             num = int(input("Enter task number to remove: "))
21             if 1 <= num <= len(tasks):
22                 removed = tasks.pop(num-1)
23                 print(f"Task '{removed}' removed.")
24             else:
25                 print("Invalid task number.")
26         except ValueError:
27             print("Please enter a valid number.")
28
29 def main():
30     while True:
31         print("\n--- To-Do List Menu ---")
32         print("1. Show Tasks")
33         print("2. Add Task")
34         print("3. Remove Task")
35         print("4. Exit")
36
37         choice = input("Enter your choice: ")
38         if choice == "1":
39             show_tasks()
40         elif choice == "2":
41             add_task()
42         elif choice == "3":
43             remove_task()
44         elif choice == "4":
45             print("Exiting To-Do List. Goodbye!")
46             break
47         else:
48             print("Invalid choice, try again.")
49
50 if __name__ == "__main__":
51     main()
52
```

Figure 3: mycode.py

- Pushed the Python file from local to GitHub by running :

```
git add my_code.py
git commit -m "PythonCode"
git push
```

This uploaded the my\_code.py file from the local system to GitHub.

```

C:\Users\Sai Keerthana\STT_1_1>git status
On branch main
Your branch is up to date with 'origin/main'.

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

nothing added to commit but untracked files present (use "git add" to track)
C:\Users\Sai Keerthana\STT_1_1>git add my_code.py
C:\Users\Sai Keerthana\STT_1_1>git status
On branch main
Your branch is up to date with 'origin/main'.

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
       new file:   my_code.py

C:\Users\Sai Keerthana\STT_1_1>git commit -m "Python code"
[main a25f77b] Python code
 1 file changed, 51 insertions(+)
 create mode 100644 my_code.py
C:\Users\Sai Keerthana\STT_1_1>git status
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
C:\Users\Sai Keerthana\STT_1_1>git push
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), 804 bytes | 804.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/ps-keerthana/STT_1_1
   cb82806..a25f77b  main -> main

```

Figure 4: Push to the repo

- When changes were made directly on GitHub (e.g :editing `my_code.py` online), they did not automatically reflect in the local repository. Used

```
git pull
```

to sync local repository with GitHub updates.

```

C:\Users\Sai Keerthana\STT_1_1>git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
C:\Users\Sai Keerthana\STT_1_1>git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 943 bytes | 62.00 KiB/s, done.
From https://github.com/ps-keerthana/STT_1_1
   a25f77b..1e8c068  main       -> origin/main
Updating a25f77b..1e8c068
Fast-forward
 my_code.py | 1 +
 1 file changed, 1 insertion(+)

```

Figure 5: Terminal showing successful pull operation

- Opened the Actions tab on GitHub for the repository and searched for Pylint and clicked Configure workflow, then GitHub automatically created a workflow file

```
.github/workflows/pylint.yml
```

with default steps for linting Python code.

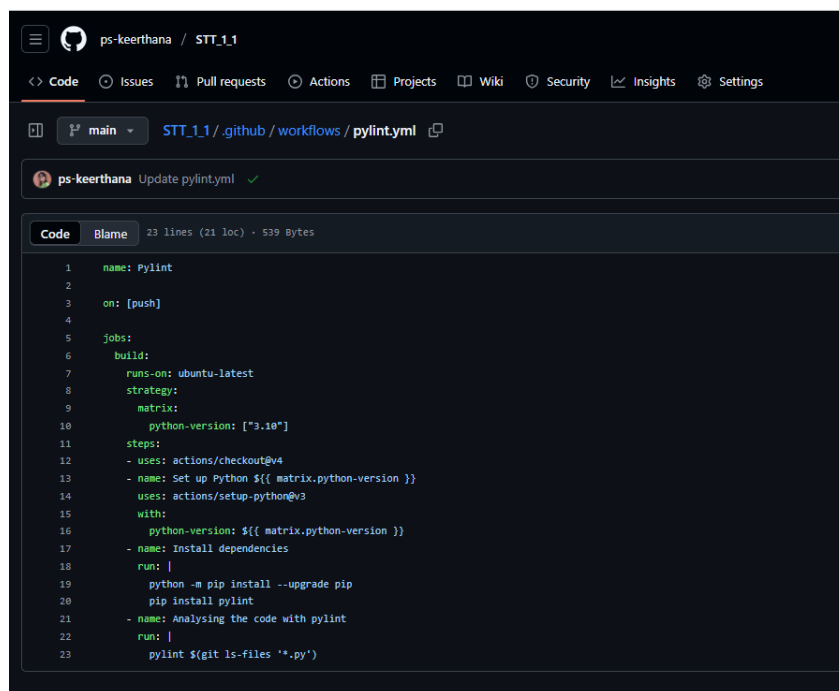


Figure 6: pylint.yml workflow

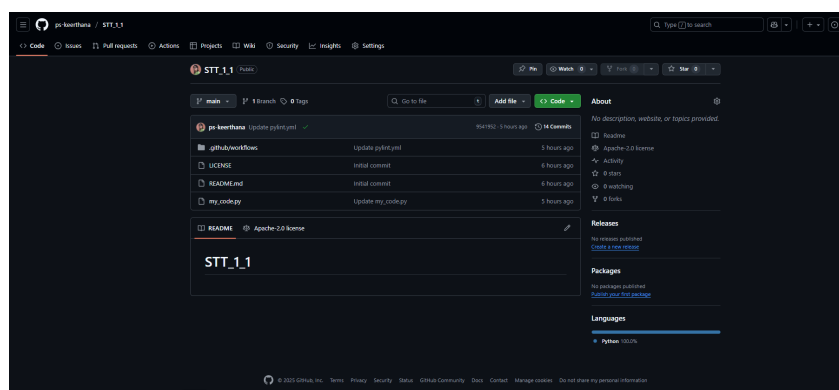


Figure 7: GitHub repository showing the pushed Python file along with the configured pylint workflow

- After the pylint workflow is setup, I committed the Python file `my_code.py` and ran the workflow few times then initially workflow failed (✗) due to syntax error in `pylint.yml`. Later, after fixing that, the workflow reported other issues such as trailing whitespace, missing module docstring, and missing function docstrings, as shown in the figures below.

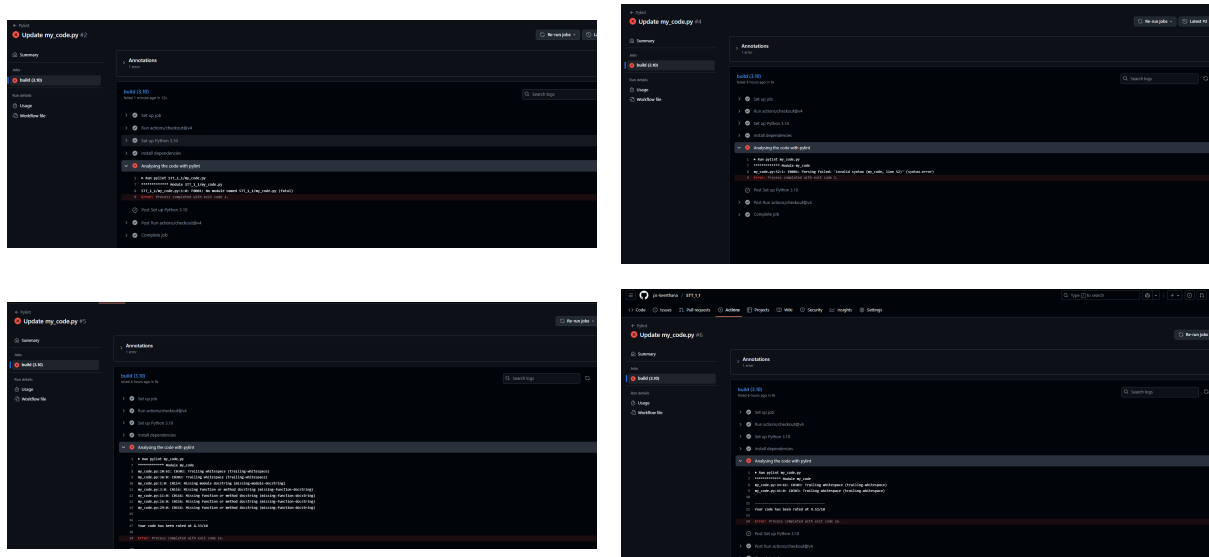


Figure 8: Pylint workflow showing errors such as syntax errors, trailing whitespace, missing module docstring, and missing function docstrings.

I then corrected the issues by removing trailing whitespace, adding a module docstring at the top of `my_code.py`, and including function docstrings for all functions. These fixes were committed directly on GitHub, after which the pylint workflow automatically re-ran and passed successfully (✓) as shown in the figure below.

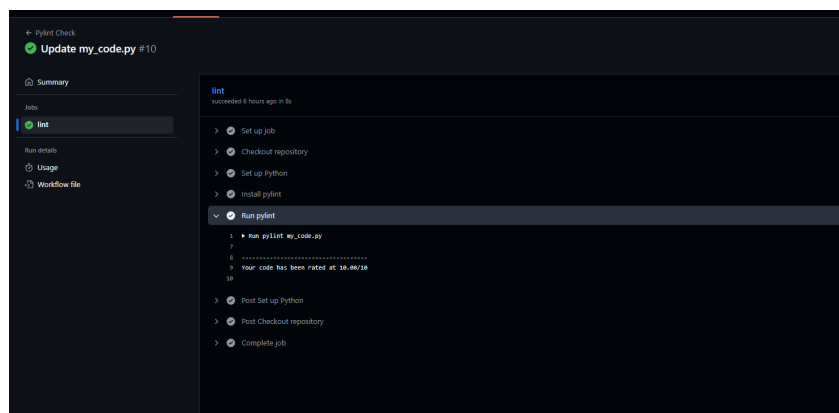


Figure 9: GitHub Actions showing green tick for successful pylint workflow



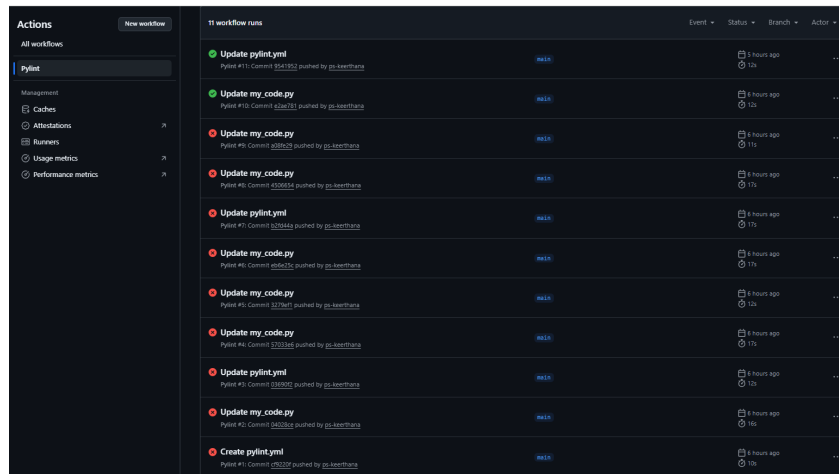
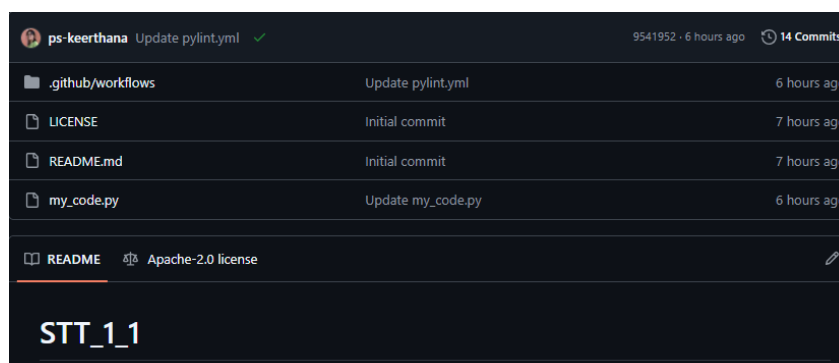


Figure 10: GitHub Actions tab showing all pylint workflow runs, including both failed (X) and successful (✓) executions.

## RESULTS AND ANALYSIS

So in this lab, I first cloned my GitHub repository `STT_1_1` to my laptop and added a Python file called `my_code.py`. I was able to commit my changes and push them to GitHub. Later, I noticed that if I made changes directly on GitHub, they didn't show up on my local repository automatically, so I had to run `git pull` to get everything synced.



Then I set up a Pylint workflow using GitHub Actions to check my code. At first, it failed because of minor issues like extra spaces at the end of lines and missing docstrings. After cleaning up the code and adding docstrings, the workflow ran successfully with a green tick.

This lab made me realize how Git keeps your local and online repositories in sync and how GitHub Actions can automatically check your code. It also taught me that writing clean and well-documented code matters just as much as making it run correctly.

## DISCUSSION AND CONCLUSION

This lab was super useful. I learned how to use Git from the command line creating repos, adding and committing files, pushing to GitHub, and pulling updates. Setting up Git was a bit tricky because of two-factor login, but I managed. The pylint workflow was cool too; it caught minor issues like extra spaces and missing docstrings, which helped make my code cleaner.

Overall,I realized the importance of version control for tracking changes and collaboration, and how GitHub Actions can help maintain code quality automatically.

## REFERENCES

- [1] Lab Document [Shared on Google Classroom]
- [2] Link to my Repo: [https://github.com/ps-keerthana/STT\\_1\\_1](https://github.com/ps-keerthana/STT_1_1)

## 0.2 LABORATORY SESSION 2

### INTRODUCTION

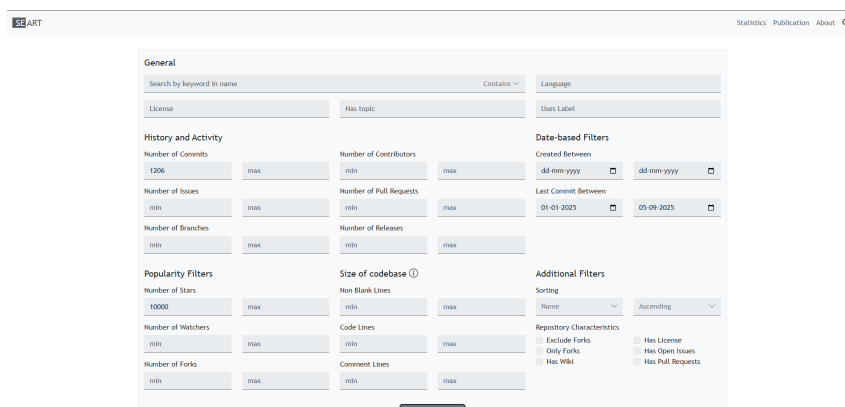
In this lab, we worked on commits in a git hub repo to find bug-fixing commits and improve their commit messages. Bugfixing commits are important because they fix errors in the code, and clear commit messages help developers understand what was changed. We used a real GitHub project to identify these commits, then applied a Large Language Model (LLM) to generate commit messages automatically. After that, we improved these messages further using a rectifier that makes them more precise and specific for each file. Finally, we compared the original developer messages, the LLM-generated messages, and the rectified messages using different evaluation methods that look at meaning, keywords, and overall quality.

### TOOLS AND ENVIRONMENT

I used Python 3.13.5 along with the latest versions of Git and key Python libraries for this experiment. PyDriller helped me mine Git repositories, while HuggingFace Transformers provided pre-trained LLMs for commit message generation. Specifically, I used mamiksik/CommitPredictorT5 as the main LLM for predicting commit messages and SEBIS/code\_trans\_t5\_base\_commit\_generation\_multitask\_finetune as a rectifier model to improve messages further. For semantic evaluation, I used BERTScore to quantify similarity between messages and code diffs. Matplotlib was used to plot evaluation results, and Pandas helped store and manipulate all data in CSV format. The experiments were run on a Kaggle environment with GPU support (NVIDIA CUDA), with CPU fallback enabled, and this setup works on Windows, Linux, or MacOS.

### SETUP

First, we needed to pick a medium-to-large GitHub repository. I used SEART Search Engine with the following filters:



The screenshot shows the SEART Search Engine interface with the following filters applied:

- General:** Search by keyword in name (empty), Contains (dropdown), Language (dropdown), License (dropdown), Has topic (checkbox).
- History and Activity:**
  - Number of Commits: 1200 (min), max
  - Number of Contributors: min, max
  - Number of Issues: min, max
  - Number of Pull Requests: min, max
  - Number of Branches: min, max
  - Number of Releases: min, max
- Popularity Filters:**
  - Number of Stars: 10000 (min), max
  - Number of Watchers: min, max
  - Number of Forks: min, max
- Size of codebase:**
  - Non Blank Lines: min, max
  - Code Lines: min, max
  - Comment Lines: min, max
- Date-based Filters:**
  - Created Between: dd-mm-yyyy (dropdown), dd-mm-yyyy (dropdown)
  - Last Commit Between: 01-01-2023 (dropdown), 05-09-2023 (dropdown)
- Additional Filters:**
  - Sorting: Name (dropdown), Ascending (dropdown)
  - Repository Characteristics:
    - Exclude Forks (checkbox)
    - Only Forks (checkbox)
    - Has License (checkbox)
    - Has Open Issues (checkbox)
    - Has Pull Requests (checkbox)

Figure 11: github seart

At first, I picked a repository with 22k commits, but running the LLM model took too long and the GPU crashed. So I went for one with fewer commits. I chose deepspeedai/DeepSpeed because:

- Popular: 40k+ stars, 4,500 forks, 2,906 commits

- Active: The repository has been updated as recently as July 31, 2025
- Relevant: Widely used deep learning library, perfect for bug-fix analysis

```

# --- Install dependencies ---
!pip install -q pydriller pandas transformers torch tqdm sentence-transformers bert-score matplotlib

# --- Imports ---
import os, re, gc
import pandas as pd
from tqdm.auto import tqdm
from pydriller import Repository
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch

# --- Setup device ---
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")

# --- Repository Selection ---
repo_name = "deepspeedai/DeepSpeed"
repo_path = f"/kaggle/working/{repo_name.split('/')[0]}"

if not os.path.exists(repo_path):
    !git clone https://github.com/{repo_name}.git {repo_path}

# --- Load Models ---
print("Loading CommitPredictorT5 (LLM)...")
llm_tokenizer = AutoTokenizer.from_pretrained("mamiksik/CommitPredictorT5")
llm_model = AutoModelForSeq2SeqLM.from_pretrained("mamiksik/CommitPredictorT5").to(device)

print("Loading SEBIS/code-trans-t5-base-commit-generation-multitask-finetune (Rectifier)...")
rect_tokenizer = AutoTokenizer.from_pretrained("SEBIS/code-trans-t5-base-commit-generation-multitask-finetune")
rect_model = AutoModelForSeq2SeqLM.from_pretrained("SEBIS/code-trans-t5-base-commit-generation-multitask-finetune").to(device)

```

Figure 12: Setup: Dependencies, repo, device, and models loaded.

## METHODOLOGY AND EXECUTION

- **Bug-Fix commits:** We need to check all commits and select the ones that fix bugs. To do this, we look for certain keywords in commit messages. Commits that modify test files are also included, as tests often fix issues. This code goes through the repository, finds bug-fix commits and details and saves them to a CSV file. If the CSV already exists, it simply loads the data.

```

# --- Stage 1: Bug-fix commits ---
commits_file = "/kaggle/working/bugfix_commits.csv"

if os.path.exists(commits_file):
    df_commits = pd.read_csv(commits_file)
    print(f"Loaded {len(df_commits)} bug-fix commits from checkpoint.")
else:
    commit_data = []
    for commit in tqdm(Repository(repo_path).traverse_commits(), desc="Scanning commits"):
        if is_bug_fix_commit(commit):
            # Save parent hashes
            parent_hashes = []
            for p in commit.parents:
                if isinstance(p, str):
                    parent_hashes.append(p)
                else:
                    parent_hashes.append(p.hexsha)
            is_merge = len(parent_hashes) > 1

            commit_data.append([
                commit.hash,
                commit.msg,
                parent_hashes,
                commit.merge,
                [mf.filename for mf in commit.modified_files]
            ])

    df_commits = pd.DataFrame(
        commit_data,
        columns=["Hash", "Message", "Parent Hashes", "Is Merge Commit", "Modified Files"]
    )
    df_commits.to_csv(commits_file, index=False)
    print(f"Saved {len(df_commits)} bug-fix commits.")

```

Figure 13: Identifying and Saving Bug-Fix Commits

| ID | Hash                                      | Message   | Parent Hashes                            | In Merge Commit? | Modified File   |
|----|---|---|--|------------------|---|
| 0  | d8226fb7aae074ce16aac9d808fde1be654892    | add two test files  | [8c4ea30071b86b81ebdbabddad7efbd0135c2]  | False            | [test_config.py, 'test_run.py<br>f_init_.py', 'ds.config_func_bsd_json'<br>ds.config_func_bsd_jon'<br>ds.config_func_bsd_no_zero.json'<br>ds.config_func_scheduler.json'<br>ds.config_perf_bsd_jon'.json',<br>ds.config_perf_bsd_jon'.gpdc_test.sh',<br>run_checkedout_test.py', run_func_test.py',<br>'run_per_baseline.py', 'run_perf_test.py', 'test_common.py',<br>'run_sanity_check.py'] |
| 1  | 98f513bb6bc52529a4ac3dfced382277312ae5    | add test model Megalon_GPT2   | [3bc21da024fa45bf67aalBcd3cf2d8358f]     | False            | 'ds.config_perf_bsd_jon'.gpdc_test.sh'  |
| 2  | e9k097H1afdc6894bcd0c56386344cdfMa7g47    | examples -> DeepSpeedExamples   | [681001fb644fd790780ca3ccr07a73abcb36]   | False            | 'ds.gpd_test.sh'  |
| 3  | caaee992ade348201277ba39930sc52678E2124c  | Model tests executable fix  | [e9b097H1afdc6894bcd0c56386344cdfMa7g47] | False            | 'ds.gpd_test.sh'  |
| 4  | b61a221710126949e0563331edfeba0a1ea711954 | Distributed testing @vinVinn<br>Add distributed test decorator and some unit tests.<br>@vinVinn Setting NCCL backend.<br>@vinVinn Parametrize test.vinVinn.rank = local_rank/vinVinn.Temporary disable CUDA initialization. | [978386114291232ba49G7F08I6I84r0686d3C2] | False            | ['common.py', 'test_dist.py']   |

Figure 14: bug-fix commit csv top 5 rows

Here i have observevd that is merge commit? is always false because there is no such merge in this repository , can have a look at repository stats in result section.

- **Diff Extraction and Analysis** This step extracts all filelevel changes from bugfixing commits, including the code before and after each change and the diff, and saves them in a CSV for further analysis.

```
# == Stage 2: Bugfix diffs ==
diffs_file = "loggit/marketing/bugfix_diffs.csv"
if os.path.exists(diffs_file):
    df_files = pd.read_csv(diffs_file)
    print(f"Loaded {len(df_files)} file-level diffs from checkpoint.")
else:
    file_data = []
    for commit in tqdm(Repository(repo_path).traverse_commits(), desc="Extracting diffs"):
        if commit.hash in set(df_commits['hash']):
            for mf in commit.modified_files:
                # Safely get source code before
                try:
                    mf_source_before = mf.source_code_before or ""
                except Exception as e:
                    print(f"Warning: Could not get source_code_before for {mf.filename} in commit {commit.hash}: {e}")
                    mf_source_before = ""

                # Safely get current source
                try:
                    mf_source_current = mf.source_code or ""
                except Exception as e:
                    print(f"Warning: Could not get source_code for {mf.filename} in commit {commit.hash}: {e}")
                    mf_source_current = ""

                # Safely get diff
                try:
                    mf_diff = mf.diff or ""
                except Exception as e:
                    print(f"Warning: Could not get diff for {mf.filename} in commit {commit.hash}: {e}")
                    mf_diff = ""

                file_data.append([
                    commit.hash,
                    commit.msg,
                    mf.filename,
                    mf_source_before,
                    mf_source_current,
                    mf_diff,
                ])

    df_files = pd.DataFrame(file_data, columns=[
        "hash", "Message", "Filename", "Source Code (before)", "Source Code (current)", "Diff"
    ])

    df_files.to_csv(diffs_file, index=False)
    print(f"Saved {len(df_files)} file-level diffs.")
```

Figure 15: Filelevel bug-fix diffs extraction.

[illegible]

Figure 16: bugfix-diffs csv top 5 rows with devmsg

- **LLM Commit Message Inference:**the bug-fix file changes (diffs) are processed using this LLM([mamiksik/CommitPredictorT5](#))to automatically generate precise and descriptive commit messages better than the dev message.

```

# --- Stage 3: LLM Inference ---
llm_file = "/kaggle/working/llm_commits.csv"
checkpoint_llm = "/kaggle/working/llm_checkpoint.csv"

def extract_keywords(diff_text, top_k=5):
    tokens = re.findall(r'\w+', str(diff_text).lower())
    freq = {}
    for t in tokens:
        freq[t] = freq.get(t, 0) + 1
    sorted_tokens = sorted(freq.items(), key=lambda x: x[1], reverse=True)
    return [t for t, _ in sorted_tokens[:top_k]]

# Load bugfix diffs
df_diffs = pd.read_csv("/kaggle/working/bugfix_diffs.csv")

if os.path.exists(checkpoint_llm):
    df_ckpt = pd.read_csv(checkpoint_llm)
    processed = set(zip(df_ckpt["Hash"], df_ckpt["Filename"]))
    print(f'Resuming Stage 3: (llm(df_ckpt)). rows processed: ')
else:
    df_ckpt = pd.DataFrame(columns=["Hash", "Message", "Filename", "LLM Inference"])
    processed = set()

batch_size = 4

for i in tqdm(range(0, len(df_diffs), batch_size), desc="LLM Inference"):
    batch = df_diffs.iloc[i:i+batch_size]
    batch = batch[batch["Hash"] != "" & batch["Filename"] != ""]
    if batch.empty:
        continue

    prompts = []
    for fname, diff_text in zip(batch["Filename"], batch["Diff"]):
        keywords = extract_keywords(diff_text)
        prompts.append(f'''
You are a software developer writing a commit message.

File: {fname}
Diff:
{diff_text}

Instructions:
- Use imperative style (Fix, Add, Update, Remove).
- 18-25 words, concise but clear.
- Must include at least 3 of these keywords: {', '.join(keywords)}
- Focus only on this file's changes.
- Avoid vague messages like 'fix bug'.

Commit message:
''')

    inputs_tok = llm_tokenizer(prompts, return_tensors='pt', padding=True, truncation=True, max_length=1024).to(device)

    with torch.no_grad():
        outputs = llm_model.generate(
            inputs_tok,
            max_new_tokens=100,
            num_beams=1,
            early_stopping=True,
            no_repeat_ngram_size=2
        )

    llm_msgs = [llm_tokenizer.decode(o, skip_special_tokens=True).strip() for o in outputs]

    # Apply length floor (append filename if too short)
    final_llm_msgs = []
    for msg, fname in zip(llm_msgs, batch["Filename"]):
        if len(msg.split()) < 10:
            msg += f' in {fname}'
        final_llm_msgs.append(msg)

    batch_out = batch[["Hash", "Message", "Filename"]].copy()
    batch_out["LLM Inference"] = final_llm_msgs

    # Save checkpoint
    batch_out.to_csv(checkpoint_llm, mode="a", index=False, header=not os.path.exists(checkpoint_llm))
    processed.update(zip(batch["Hash"], batch["Filename"]))

    gc.collect()
    torch.cuda.empty_cache()

# Save final LLM file
if os.path.exists(checkpoint_llm):
    os.rename(checkpoint_llm, llm_file)
    print(f'Final LLM CSV saved: {llm_file}')

```

Figure 17: LLM working code with a prompt

|   | Hash                                    | Message                      | Filename                | LLM Inference   |
|---|---|------------------------------|-------------------------|---|
| 0 | dc226f6b7aae074ce16aa68d088dfe1be654892 | add two test files           | test_config.py          | add regression test for in test_config.py               |
| 1 | dc226f6b7aae074ce16aa68d088dfe1be654892 | add two test files           | test_run.py             | add more tests for parse_resource_filter in test_run.py |
| 2 | 98f5131bb6b252294a63cfed3822773112ee5   | add test model Megatron_GPT2 | __init__.py             | add note about gpt2 function test case in __init__.py   |
| 3 | 98f5131bb6b252294a63cfed3822773112ee5   | add test model Megatron_GPT2 | ds_config_func_bs4.json | add add_fp16_to_config.js in ds_config_func_bs4.json    |
| 4 | 98f5131bb6b252294a63cfed3822773112ee5   | add test model Megatron_GPT2 | ds_config_func_bs8.json | add add_fp16_to_ds_config.js in ds_config_func_bs8.json |

Figure 18: llm csv 5rows with llm message

- Rectifier Formulation** The rectifier is needed because sometimes a single commit changes multiple files or multiple parts of a file, and the original commit message might not describe everything clearly. Even if the LLM generates a message, it may still miss some details.

My rectifier SEBIS/code\_trans\_t5\_base\_commit\_generation\_multitask\_finetune makes commit messages clearer and more accurate for each file. It looks at the original developer message, the LLM message, and the code changes, then writes a better message that explains the changes properly. If the rectifier message misses anything, it adds information from the LLM and developer messages to make sure nothing important is left out.

```

# Build rectifier prompts
prompts = [
    f"""
You are an expert software developer. Improve the commit message below so that it is:
- Longer and more descriptive than the LLM version
- Concise, precise, and human-readable
- Specific to the file: {fname}
- Includes all important keywords from the diff: {' '.join(extract_keywords(diff_text))}
- Emphasizes the main change

Diff of file:
{diff_text}

Current LLM commit message:
{llm_msg}

Final improved commit message:
"""
    for fname, diff_text, llm_msg in zip(
        batch["Filename"].fillna(""),
        batch["Diff"].fillna(""),
        batch["LLM Inference"].fillna("")
    )
]

# Tokenize + generate
inputs_tok = rect_tokenizer(
    prompts, return_tensors="pt", padding=True, truncation=True, max_length=1024
).to(device)

with torch.no_grad():
    outputs = rect_model.generate(
        *inputs_tok,
        max_new_tokens=200,
        num_beams=5,
        early_stopping=True,
        no_repeat_ngram_size=2
    )

rect_msgs = [rect_tokenizer.decode(o, skip_special_tokens=True).strip() for o in outputs]
final_msgs = []
for rect_msg, llm_msg, dev_msg, diff_text in zip(
    rect_msgs,
    batch["LLM Inference"].astype(str),
    batch["Message"].astype(str),
    batch["Diff"].astype(str)
):
    rect_tokens = set(rect_msg.lower().split())
    llm_tokens = set(llm_msg.lower().split())
    diff_keywords = set(extract_keywords(diff_text))

    # If rectifier is weaker, merge with LLM
    if (len(rect_msg.split()) < len(llm_msg.split())) or \
        (len(rect_tokens & diff_keywords) < len(llm_tokens & diff_keywords)):
        rect_msg = rect_msg + " | " + llm_msg

    # Always include LLM and dev messages (strong guarantee)
    if llm_msg not in rect_msg:
        rect_msg += " | " + llm_msg
    if dev_msg not in rect_msg:
        rect_msg += " | " + dev_msg

    final_msgs.append(rect_msg)

```

Figure 19: Rectifier logic

[illegible]

Figure 20: commit messgaes generated by my rectifier

Here we can we look how better rectifier commit message is comapred to llm and dev message

- **Evaluation** i calculated hit rates by measring how closely each message generated by dev,llm,rectifier matches teh actual code change (diff),using then taking the average across all commits for each type of mesage

```
# --- Prepare references and predictions ---
refs = df_final["Diff"].astype(str).tolist()
devs = df_final["Message"].astype(str).tolist()
llms = df_final["LLM Inference"].astype(str).tolist()
rects = df_final["Rectified Message"].astype(str).tolist()
```

```
bert_scores = {
    "Dev_vs_Diff": bert_score_batches(devs, refs),
    "LLM_vs_Diff": bert_score_batches(llms, refs),
    "Rectified_vs_Diff": bert_score_batches(rects, refs)
}
```

```
values = [np.mean(scores[k]) for k in x_keys]
```

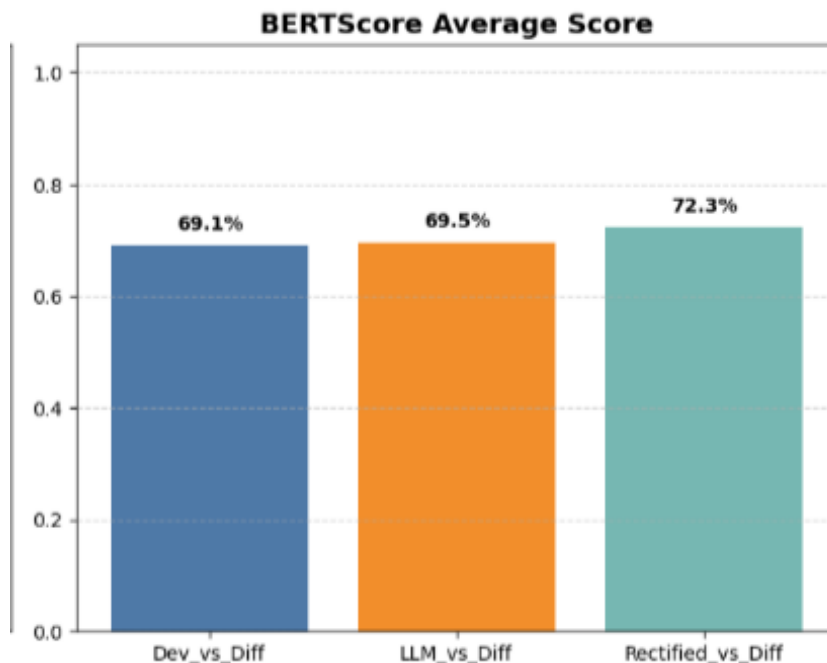


Figure 21: hit rates for dev,llm,rect

## RESULTS AND ANALYSIS

As previously seen from the hit rates obtained using BERTScore in the evaluation step, we can conclude the following:

- **RQ1 (developer eval):** Hit rate: 69.1% (Dev vs Diff) Developer commit messages were somewhat precise. On average, Some messages were vague, especially when multiple files were changed in a single commit.
- **RQ2 (LLM eval):** Hit rate: 69.5% (LLM vs Diff) The LLM generated more precise messages than developers in most cases. However, it sometimes missed contextspecific details due to truncation that a human developer included.



- **RQ3 (rectifier eval):** Hit rate: 72.3% (Rectifier vs Diff) The rectifier made messages stronger. It combined Dev and LLM messages and added missing context. Messages became clearer and matched the code better..

And from my analysis i could say that Hit rates vary with the evaluation metric. BERTScore ranks Rectifier > LLM > Developer, while CodeBERT and ROUGE-L sometimes rank Developer > LLM, though Rectifier stays on top. This happens because my Rectifier merges keywords from Dev, LLM, and Diff, but LLM can miss some keywords due to truncation, and developers naturally include most Diff keywords.

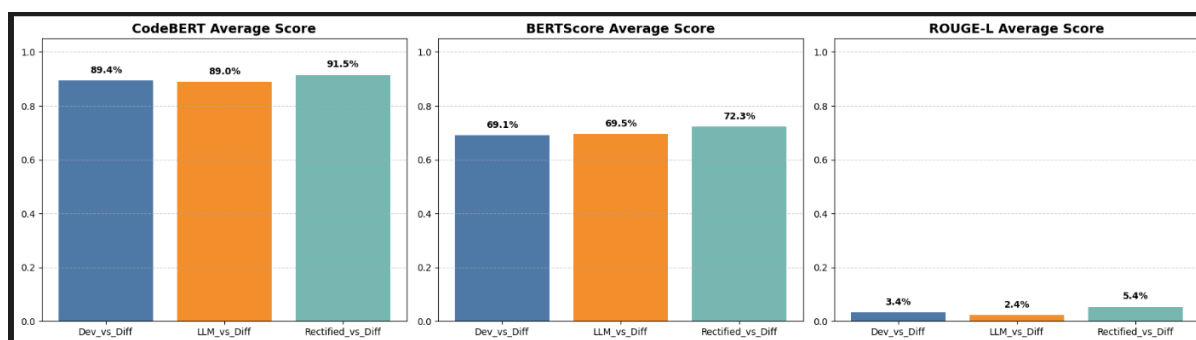


Figure 22: Hit rates comparison for different evaluation metrics

- And below here are my repository stats

```
# --- Compute Repository Stats ---
total_commits = len(list(Repository(repo_path).traverse_commits()))
total_bugfix_commits = len(df_commits)
total_bugfix_files = df_files["Filename"].nunique()
avg_files_per_bugfix_commit = len(df_files) / len(df_commits)
merge_pct = df_commits["Is Merge Commit?"].mean() * 100

# handle NaNs safely
df_files["Lines Changed"] = df_files["Diff"].fillna("").astype(str).apply(lambda x: x.count("\n"))
avg_lines_per_file = df_files["Lines Changed"].mean()

repo_stats = {
    "Total commits": total_commits,
    "Total bug-fix commits": total_bugfix_commits,
    "Unique files in bug-fix commits": total_bugfix_files,
    "Average files per bug-fix commit": avg_files_per_bugfix_commit,
    "Merge commits percentage": merge_pct,
    "Average lines changed per file": avg_lines_per_file
}

pd.DataFrame([repo_stats]).to_csv('/kaggle/working/repo_stats.csv', index=False)
print('\n== Repository Stats ==')
for k,v in repo_stats.items():
    print(f'{k}: {v}')

== Repository Stats ==
Total commits: 2900
Total bug-fix commits: 1481
Unique files in bug-fix commits: 1386
Average files per bug-fix commit: 6.088453747467927
Merge commits percentage: 0.0
Average lines changed per file: 67.2977708772319
```

## DISCUSSION AND CONCLUSION

In this lab, we worked with big realworld repositories. There were thousands of commits, so it was hard to handle everything. Sometimes diff extraction failed, especially for binary or deleted files. Running the LLM and embeddings was tricky. The GPU sometimes disconnected because of memory issues. A few times the developer's original message was better because it used exact keywords.

I learned how developers write commit messages and how to find bug-fixing commits. LLMs can make good messages, but they can miss details. Using a rectifier helps make the messages more correct.

In the end, we built a system that improves commit messages. The messages are clearer, more exact, and match the code changes better. This can help keep software clean and also help with automatic bug fixing or security checks.

## REFERENCES

- [1] Lab Document [Shared on Google Classroom]
- [2] Link to my Repo: [https://github.com/ps-keerthana/STT\\_1\\_2](https://github.com/ps-keerthana/STT_1_2)
- [3] <https://pydriller.readthedocs.io/en/latest/index.html>
- [4] <https://huggingface.co/mamiksik/CommitPredictorT5>

## 0.3 LABORATORY SESSION 3

### INTRODUCTION

The aim of this lab is to study bugfix commits from lab2 using two types of measures: structural code metrics and similarity metrics. The structural metrics, calculated using Radon, include Maintainability Index (MI) which tells how easy the code is to maintain, Cyclomatic Complexity (CC) which shows how complex the control flow is, and Lines of Code (LOC) which measures the length of the code. For similarity, we use CodeBERT (microsoft/codebert-base) to measure semantic similarity and SacreBLEU to measure token similarity. Using these values, we classify bug fixes as either Major or Minor, and then compare the results to see if both approaches agree with each other.

### TOOLS AND ENVIRONMENT

| Category            | Details  |
|---------------------|--|
| Python Version      | 3.13.5   |
| Structural Metrics  | radon → MI, CC, LOC  |
| Token Similarity    | sacrebleu → BLEU score (token-level similarity)  |
| Semantic Similarity | transformers, torch, sentence-transformers → Embeddings & similarity using <b>CodeBERT</b> |
| Data Handling       | pandas   |
| Visualization       | matplotlib, seaborn  |
| Pre-trained Model   | microsoft/codebert-base → for semantic embeddings  |

Table 2: Libraries, tools, and models used in the experiment

```
# --- Install dependencies ---
!pip install -q radon sacrebleu transformers torch tqdm sentence-transformers pandas matplotlib seaborn

# --- Imports ---
import os, warnings
import pandas as pd
import numpy as np
from tqdm.auto import tqdm
from radon.complexity import cc_visit
from radon.metrics import mi_visit
from radon.raw import analyze
from sentence_transformers import SentenceTransformer
import sacrebleu
import torch
import matplotlib.pyplot as plt
import seaborn as sns
```

### SETUP

I used Google Colab to run this lab, where it automatically checks if a GPU (CUDA) is available and otherwise runs on CPU. This made the model (CodeBERT) run faster. I also added checkpoints in the code to save progress while running big loops, and then I loaded the Lab 2 dataset to proceed further.

```
# --- Setup ---
device = "cuda" if torch.cuda.is_available() else "cpu"
print(f"Using device: {device}")
warnings.filterwarnings("ignore")
sns.set(style="whitegrid")

# --- Directories ---
CHECKPOINT_DIR = "/content/lab3_checkpoints"
os.makedirs(CHECKPOINT_DIR, exist_ok=True)

# --- Load Lab 2 CSV dataset ---
lab2_file = "/content/rectified_commits.csv" # Upload your file
df = pd.read_csv(lab2_file)
```

Figure 23: setup

## METHODOLOGY AND EXECUTION

- after loading lab2 dataset i checked for some basic info

```
# --- Baseline statistics ---
print(f"Total commits: {df['Hash'].nunique()}")
print(f"Total files: {len(df)}")
print(f"Avg files per commit: {df.groupby('Hash')['Filename'].count().mean():.2f}")
print(f"Fix type distribution:\n", df["LLM Inference"].value_counts().head(10))
print(f"Most common file extensions:\n", df["Filename"].apply(lambda x: os.path.splitext(x)[1]).value_counts().head(10))
```

```
Using device: cuda
Total commits: 1481
Total files: 8420
Avg files per commit: 5.69
Fix type distribution:
LLM Inference
update engine.py in engine.py          59
update config.py in config.py          26
update checkpointing.py in checkpointing.py 19
update cpu_adam.py in cpu_adam.py      17
update auto_tp.py in auto_tp.py        16
update pipeline_engine.py in engine.py  15
add missing description in __init__.py  15
update replace module.py in replace_module.py 14
add note about imperative style in requirements.txt 12
add tests for deepspeed in engine.py    12
Name: count, dtype: int64
Most common file extensions:
Filename
.py      5708
.md      661
.yml     578
.h       389
.cpp     255
.cu      230
.png     97
.txt     95
.rst     72
72
Name: count, dtype: int64
```

- For each Python file,i used **radon** to calculate the Maintainability Index (MI), Cyclomatic Complexity (CC), and Lines of Code (LOC) before and after the bug fix.I then added new columns to save these values as and checkpointed them

MI\_Before, MI\_After, MI\_Change

CC\_Before, CC\_After, CC\_Change

LOC\_Before, LOC\_After, LOC\_Change

```

# --- Stage (c) Radon Metrics for Python files only ---
def compute_radon_metrics(code, filename):
    """Compute MI, CC, LOC safely for Python files"""
    if not str(filename).endswith(".py") or not str(code).strip():
        return np.nan, np.nan, np.nan
    try:
        code = str(code)
        mi = mi_visit(code, True)
        cc_items = cc_visit(code)
        cc_sum = sum([c.complexity for c in cc_items]) if cc_items else 0
        loc = analyze(code).loc
        return mi, cc_sum, loc
    except:
        return np.nan, np.nan, np.nan

radon_checkpoint = os.path.join(CHECKPOINT_DIR, "radon_metrics.csv")
if os.path.exists(radon_checkpoint):
    print("Loading Radon metrics from checkpoint...")
    df_radon = pd.read_csv(radon_checkpoint)
    for col in ["MI_Before", "CC_Before", "LOC_Before", "MI_After", "CC_After", "LOC_After",
               "MI_Change", "CC_Change", "LOC_Change"]:
        if col in df_radon.columns:
            df[col] = df_radon[col]
else:
    print("Computing Radon metrics for Python files...")
    mi_before, cc_before, loc_before = [], [], []
    mi_after, cc_after, loc_after = [], [], []
    for _, row in tqdm(df.iterrows(), total=len(df)):
        mi_b, cc_b, loc_b = compute_radon_metrics(row["Source Code (before)"], row["Filename"])
        mi_a, cc_a, loc_a = compute_radon_metrics(row["Source Code (current)"], row["Filename"])
        mi_before.append(mi_b); cc_before.append(cc_b); loc_before.append(loc_b)
        mi_after.append(mi_a); cc_after.append(cc_a); loc_after.append(loc_a)
    df["MI_Before"], df["CC_Before"], df["LOC_Before"] = mi_before, cc_before, loc_before
    df["MI_After"], df["CC_After"], df["LOC_After"] = mi_after, cc_after, loc_after
    df["MI_Change"] = df["MI_After"] - df["MI_Before"]
    df["CC_Change"] = df["CC_After"] - df["CC_Before"]
    df["LOC_Change"] = df["LOC_After"] - df["LOC_Before"]
    df.to_csv(radon_checkpoint, index=False)
    print(f"Radon metrics saved: {radon_checkpoint}")

```

- to measure the similarity between old and new codes, Semantic similarity was computed with CodeBERT embeddings and cosine similarity, and Token similarity was measured with SacreBLEU.

```

# --- Stage (d) Semantic & Token Similarity ---
def safe_tokenize(code):
    code_str = str(code).strip()
    return code_str.split() if code_str else ["<EMPTY>"]

def compute_token_similarity(before, after):
    try:
        return sacrebleu.sentence_bleu(" ".join(safe_tokenize(after)),
                                         [" ".join(safe_tokenize(before))]).score / 100.0
    except:
        return 0.0

embedding_checkpoint = os.path.join(CHECKPOINT_DIR, "embeddings_similarity.csv")
BATCH_SIZE = 16
sbert_model = SentenceTransformer('microsoft/codebert-base', device=device)

if os.path.exists(embedding_checkpoint):
    print("Loading embeddings & similarity from checkpoint...")
    df_emb = pd.read_csv(embedding_checkpoint)
    df["Semantic_Similarity"] = df_emb["Semantic_Similarity"]
    df["Token_Similarity"] = df_emb["Token_Similarity"]
else:
    print("Computing Semantic & Token Similarity...")
    semantic_sims, token_sims = [], []
    for i in tqdm(range(0, len(df), BATCH_SIZE)):
        batch = df.iloc[i:i+BATCH_SIZE]
        codes = []
        for _, row in batch.iterrows():
            codes.append(str(row["Source Code (before)"]))
            codes.append(str(row["Source Code (current)"]))
        embeddings = sbert_model.encode(codes, convert_to_tensor=True)
        for j in range(0, len(embeddings), 2):
            cos_sim = torch.nn.functional.cosine_similarity(embeddings[j], embeddings[j+1], dim=0)
            semantic_sims.append(float(cos_sim))
        for _, row in batch.iterrows():
            token_sims.append(compute_token_similarity(row["Source Code (before)"],
                                                       row["Source Code (current)"]))
    df["Semantic_Similarity"] = semantic_sims
    df["Token_Similarity"] = token_sims
    df.to_csv(embedding_checkpoint, index=False)

```

Figure 24: Cloning of the repo

- Each commit was classified as Minor or Major based on thresholds: Semantic similarity  $\geq 0.80 \rightarrow$  Minor,  $< 0.80 \rightarrow$  Major; Token similarity  $\geq 0.75 \rightarrow$  Minor,  $< 0.75 \rightarrow$  Major. New columns **Semantic\_Class** and **Token\_Class** were added.

```

SEM_THRESHOLD, TOKEN_THRESHOLD = 0.80, 0.75
df["Semantic_Class"] = df["Semantic_Similarity"].apply(lambda x: "Minor" if x >= SEM_THRESHOLD else "Major")
df["Token_Class"] = df["Token_Similarity"].apply(lambda x: "Minor" if x >= TOKEN_THRESHOLD else "Major")
print("\nSemantic Class counts:\n", df["Semantic_Class"].value_counts())
print("\nToken Class counts:\n", df["Token_Class"].value_counts())

```

```

Semantic Class counts:
Semantic_Class
Minor      7157
Major      1263
Name: count, dtype: int64

Token Class counts:
Token_Class
Minor      6616
Major      1804

```

Figure 25: Commit classification based on semantic and token similarity thresholds and counts observed

- checked whether **Semantic\_Class** and **Token\_Class** agree for each commit, a new column added **Classes\_Agree**, with YES if they match and NO if they differ.

```
df['Classes_Agree'] = df.apply(lambda row: "YES" if row["Semantic_Class"] == row["Token_Class"] else "NO", axis=1)
print("\nAgreement counts:\n", df['Classes_Agree'].value_counts())
```

```
Agreement counts:
Classes_Agree
YES      7879
NO       541
Name: count, dtype: int64
```

Figure 26: agreement based on semantic and token similarity thresholds and counts observed

## RESULTS AND ANALYSIS

- observed range of changes of MI,CC,LOC were calculated and ranges were shown below along with the Histograms of MI Change, CC Change, and LOC Change were shown

```
MI Change: min = -100.0 , max = 82.50989689839646
CC Change: min = -187.0 , max = 522.0
LOC Change: min = -1423.0 , max = 2831.0
```

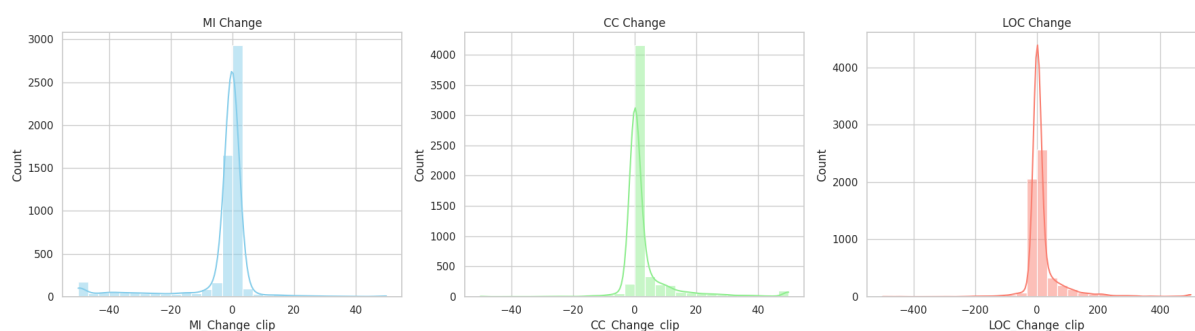


Figure 27: Histograms of MI Change, CC Change, and LOC Change

what could the changes mean?

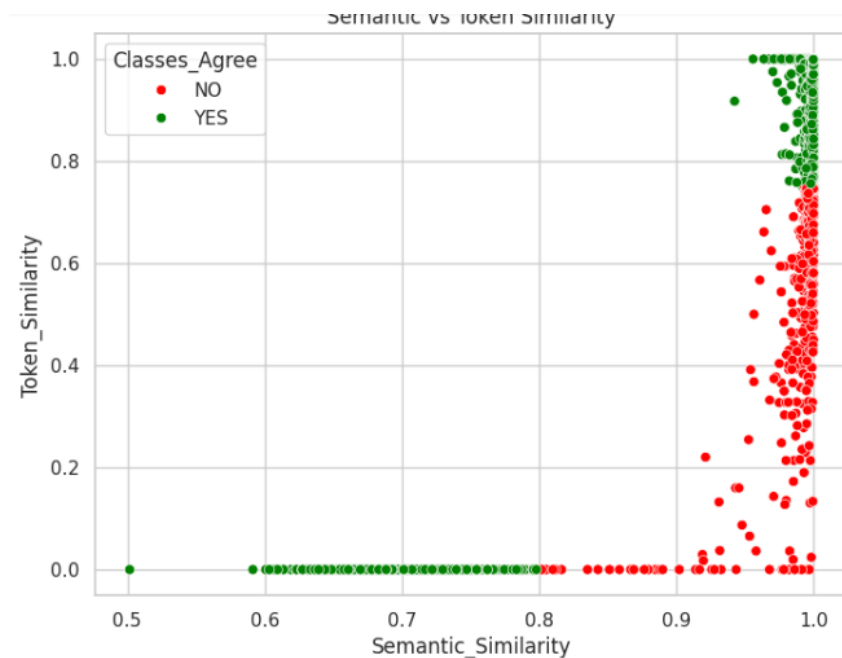
**MI Change** measures how maintainable the code became; **CC Change** measures change in complexity; **LOC Change** measures code size change.

- observed range of semantic and token similarities were calculated and shown below along with Scatterplot of Semantic vs Token Similarity

```
Semantic Similarity: min = 0.5016905069351196 , max = 1.000000238418579
Token Similarity: min = 0.0 , max = 1.0000000000000004
```

| Metric            | What it Means  |
|-------------------|--|
| <b>MI Change</b>  | Negative → code became harder to maintain (messy code added)     |
|                   | Positive → code became easier to maintain (refactoring/cleanup)  |
|                   | Big drop = drastic worsening, big increase = major improvement   |
| <b>CC Change</b>  | Negative → simplified logic, easier to understand                |
|                   | Positive → more complex code added                               |
|                   | Big increase = large function with many branches/loops           |
| <b>LOC Change</b> | Negative → lines of code deleted (cleanup or removed modules)    |
|                   | Positive → lines of code added (big features or major bug fixes) |

Table 3: Explanation of code metrics changes



Semantic Similarity (**CodeBERT**) measures meaning similarity, and Token Similarity (**BLEU**) measures surface-level textual similarity.

| Metric                     | What it Means                                    |
|----------------------------|--|
| <b>Semantic Similarity</b> | Low (0.5) → meaning of code completely changed   |
|                            | High (1.0) → almost no change, only small tweaks |
| <b>Token Similarity</b>    | 0.0 → code completely rewritten                  |
|                            | 1.0 → code almost identical                      |

Table 4: Explanation of similarity metrics

- after looking at the semantic and token similarity counts based on the thresholds as mentioned in methodology we could say that Both methods show that most bug fixes (80–85%) are **Minor**, involving small changes or tweaks, while a smaller portion (15–20%) are **Major**, representing significant rewrites or largescale changes to the code.
- after checking whether Semantic Class and Token Class agree ,the pie chart looks like



### Agreement between Semantic and Token Classification

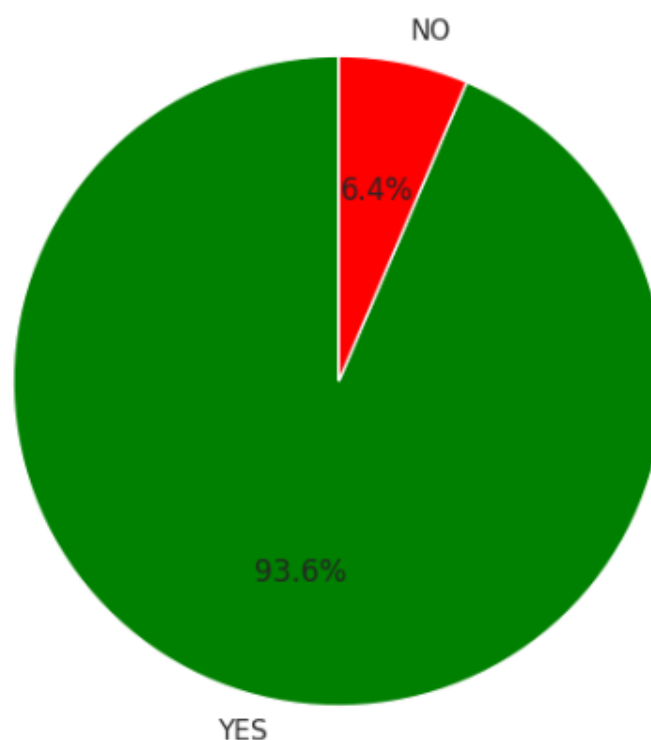


Figure 28: Pie chart of Agreement vs Disagreement.

Disagreements here might have occurred when code looks different but means the same (high semantic, low token similarity, like renaming/formatting) or looks similar but behaves differently (low semantic, high token similarity, like changing an operator). This shows the need for both semantic and token analysis.

## DISCUSSION AND CONCLUSION

I faced challenges such as Radon not working on nonPython files( i safely skipped those) and token similarity being sensitive to formatting. Computing embeddings for many files was slow, but using batching and checkpoints helped manage the process.

From this work, we learned that structural metrics (MI, CC, LOC) reveal how bug fixes affect code quality. Semantic and token similarity mostly agree, though token similarity is more sensitive to style changes. Using both metrics together helps classify fixes as Major or Minor, providing a clear view of bug-fix impacts.

## REFERENCES

- [1] Lab Document [Shared on Google Classroom]
- [2] Link to my Repo: [https://github.com/ps-keerthana/STT\\_1.3](https://github.com/ps-keerthana/STT_1.3) [3]

## 0.4 LABORATORY SESSION 4

### INTRODUCTION

The main goal of this lab is to study how different diff algorithms show changes in files for real open-source projects. A diff algorithm compares two versions of a file and shows what has been added, removed, or changed. In this lab, we focus on two diff algorithms:

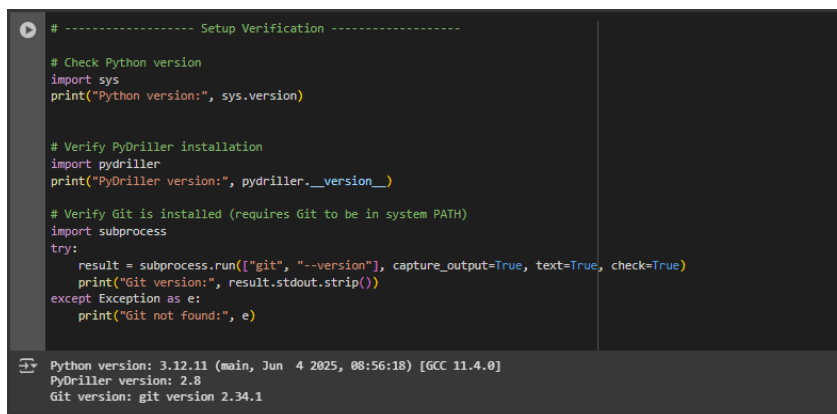
- **Myers diff** – used by the **PyDriller**. It finds differences between file versions in a straightforward way.
- **Histogram diff** – This is provided by **Git** and uses a different method to detect changes, which can sometimes give slightly different results.

We want to see how these two algorithms behave when looking at different kinds of files in a project, such as source code files(main program), test code files(codes to test),README,LICENSE files

### TOOLS AND ENVIRONMENT

I operated this on Windows 10 using Python 3.15.3. The tools and libraries used included PyDriller (for analyzing Git repositories), Git (for cloning repositories and running histogram diffs), Pandas (for data handling), and Matplotlib (for generating plots and visualizing results).

### SETUP



```
# ----- Setup Verification -----  
  
# Check Python version  
import sys  
print("Python version:", sys.version)  
  
# Verify PyDriller installation  
import pydriller  
print("PyDriller version:", pydriller.__version__)  
  
# Verify Git is installed (requires Git to be in system PATH)  
import subprocess  
try:  
    result = subprocess.run(["git", "--version"], capture_output=True, text=True, check=True)  
    print("Git version:", result.stdout.strip())  
except Exception as e:  
    print("Git not found:", e)  
  
Python version: 3.12.11 (main, Jun 4 2025, 08:56:18) [GCC 11.4.0]  
PyDriller version: 2.8  
Git version: git version 2.34.1
```

Figure 29: setup

### METHODOLOGY AND EXECUTION

- **Repo selection**First, we needed to pick a medium-to-large GitHub repository. I used SEART Search Engine with the following filters:

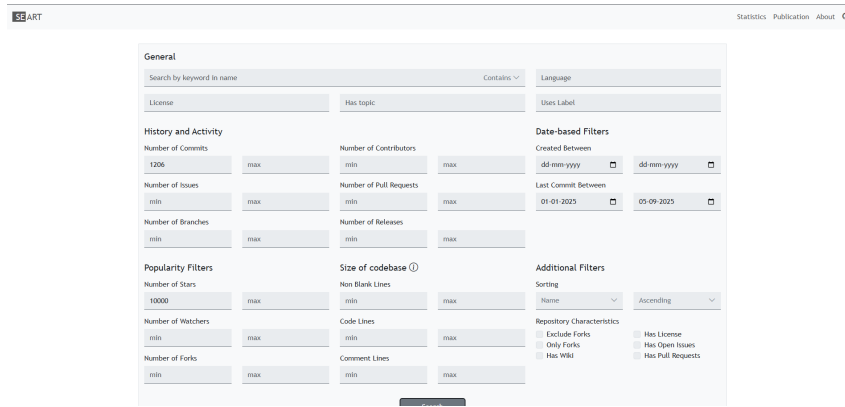


Figure 30: github seart

Selected three medium to large open source repositories CHOSEN REPOSITORIES

### **gpt-researcher**

Stars: 19,000

Forks: 2,100

Description: An open deep research agent designed for both web and local research on any given task. The agent produces detailed, factual, and unbiased reports.

Reason for Selection: Its large number of stars and forks shows strong community interest and active development, making it good for analyzing diff outputs.

### **elegantrl**

Stars: 4,200

Forks: 942

Description: An open-source massively parallel library for deep reinforcement learning algorithms, implemented in PyTorch.

Reason for Selection: The project's focus on deep reinforcement learning and good community engagement provides a diverse dataset for diff analysis.

### **maxtext**

Stars: 1,900

Forks: 400

Description: A high-performance, highly scalable, open-source large language model written in pure Python/JAX, targeting Google Cloud TPUs and GPUs for training and inference.

Reason for Selection: Its advanced use of JAX and scalability features give a unique perspective for diff algorithm analysis.

- First i cloned the repository and defined paths for saving csv files(raw diffs and final dataset)

```
repo_url = "https://github.com/AI-Hypercomputer/maxtext"
local_path = "maxtext"
csv_raw = "maxtext_diffs.csv"
csv_final = "maxtext_final.csv"
```

```
extract_repo_diffs(repo_url, local_path, csv_raw)
df_final = add_discrepancy(csv_raw, csv_final)
```

```
repo_url = "https://github.com/ai4finance-foundation/elegantrl"
local_path = "elegantrl"
csv_raw = "elegantrl_diffs.csv"
csv_final = "elegantrl_final.csv"
```

```
repo_url = "https://github.com/AI-Hypercomputer/maxtext"
local_path = "maxtext"
csv_raw = "maxtext_diffs.csv"
csv_final = "maxtext_final.csv"
```

Figure 31: Cloning of the repo

- For each commit in the repository, traverse its modified files and extract both the Myers diff using PyDriller and the Histogram diff using `git diff --histogram`.

```
# Extract commit diffs
def extract_repo_diffs(repo_url, local_path, output_csv):
    # Clone repo locally if not present
    if not os.path.exists(local_path):
        subprocess.run(["git", "clone", repo_url, local_path])

    with open(output_csv, '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_myers", "diff_hist"
        ])

    for commit in Repository(local_path).traverse_commits():
        parent_commits = commit.parents if commit.parents else ["None"]
        parent_str = ";".join(parent_commits)

        for modified_file in commit.modified_files:
            diff_myers = modified_file.diff if modified_file.diff else ""
            diff_hist = get_hist_diff(local_path, commit.hash, modified_file.new_path)

            writer.writerow([
                modified_file.old_path,
                modified_file.new_path,
                commit.hash,
                parent_str,
                commit.msg,
                diff_myers,
                diff_hist
            ])
    )
```

Figure 32: extract commit diffs

- Helper function runs Git histogram diff for a given file in a commit.

```
# Helper: Run histogram diff
def get_hist_diff(repo_path, commit_hash, file_path):
    if file_path is None:
        return ""
    try:
        cmd = [
            "git", "-C", repo_path, "diff", "--histogram",
            f"{commit_hash}^", commit_hash, "--", file_path
        ]
        result = subprocess.run(cmd, capture_output=True, text=True, check=True)
        return result.stdout
    except Exception as e:
        return str(e)
```

Figure 33: mycode.py

- cleaned the diff by removing metadata lines like `diff --git, index, ---, +++, @@` and ignored blank lines and extra spaces.

```
# Clean diff: remove metadata lines, normalize whitespace
def clean_diff(diff_str):
    cleaned = []
    for line in str(diff_str).splitlines():
        line = line.strip()
        if not line:
            continue
        # Skip diff metadata lines
        if line.startswith("diff --git") or line.startswith("index ") \
           or line.startswith("---") or line.startswith("+++") \
           or line.startswith("@@"):
            continue
        cleaned.append(line)
    return "\n".join(cleaned)
```

Figure 34: cleaning the diff

- Compared cleaned Myers and Histogram diffs and labeled Discrepancy column = Yes if they differ, else No.

```
# Add discrepancy column (now using clean_diff)
def add_discrepancy(input_csv, output_csv):
    df = pd.read_csv(input_csv)

    df["Discrepancy"] = [
        "No" if clean_diff(row["diff_myers"]) == clean_diff(row["diff_hist"]) else "Yes"
        for _, row in df.iterrows()
    ]

    df.to_csv(output_csv, index=False)
    return df
```

- plotted the count of mismatches of different file types, by looping through the final dataset.

```
# Generate stats + plot
def generate_stats_plot(df, repo_name):
    categories = {"source": 0, "test": 0, "readme": 0, "license": 0}

    for _, row in df.iterrows():
        if row["Discrepancy"] == "Yes":
            file_path = str(row["new_file_path"])
            if file_path is None or file_path == "nan":
                continue
            fp_lower = file_path.lower()

            if "test" in fp_lower:
                categories["test"] += 1
            elif "readme" in fp_lower:
                categories["readme"] += 1
            elif "license" in fp_lower:
                categories["license"] += 1
            else:
                categories["source"] += 1

    plt.bar(categories.keys(), categories.values())
    plt.title(f"Diff Mismatches in {repo_name}")
    plt.xlabel("File Type")
    plt.ylabel("Mismatches (#)")
    plt.show()
```

## RESULTS AND ANALYSIS

- Discrepancy Counts for Repositories:

| Repository                  | No   | Yes  | Total | % Discrepancy |
|-----------------------------|------|------|-------|---------------|
| <code>gpt-researcher</code> | 3701 | 2256 | 5957  | 38%           |
| <code>ElegantRL</code>      | 3472 | 6734 | 10206 | 66%           |
| <code>MaxText</code>        | 6669 | 3058 | 9727  | 31%           |

Table 5: Discrepancy counts across repositories

Myers and Histogram diffs mostly agree, but some files differ due to formatting differences. Source code files show the most discrepancies, while `test`, `README`, and `LICENSE` files show fewer mismatches. `ElegantRL` shows the highest percentage of discrepancies, likely due to large commits or complex code changes.

- Barplot Observations:

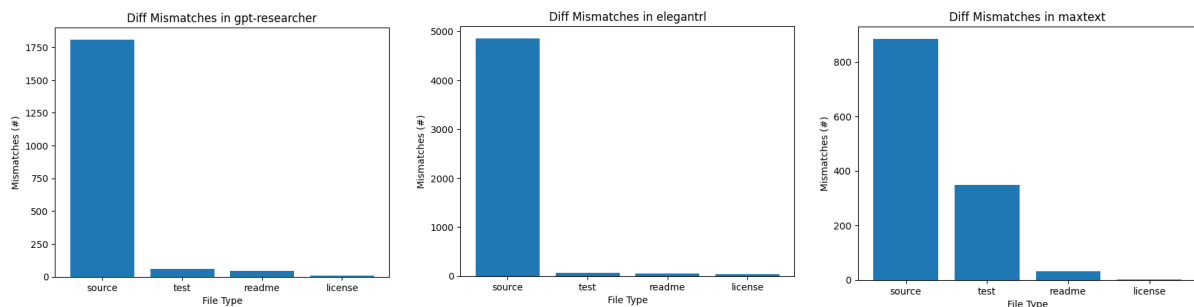
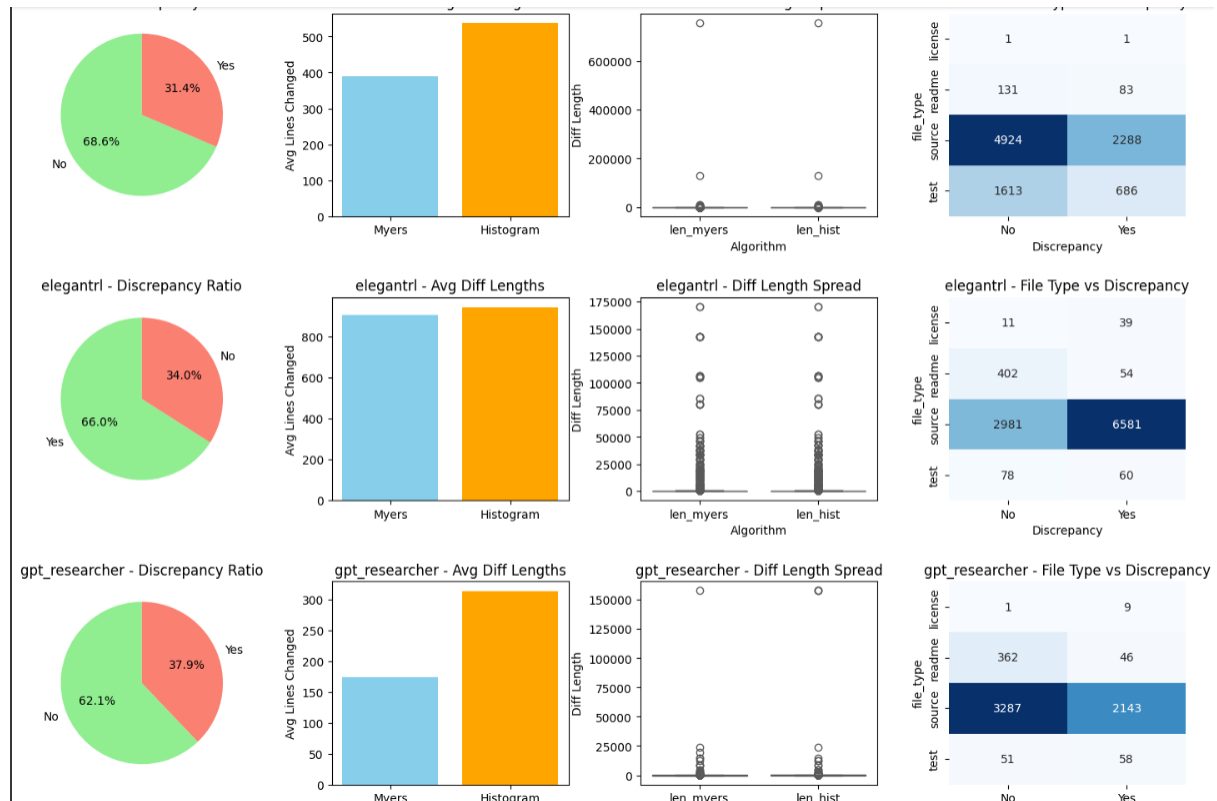


Figure 35: Counts of mismatches across repositories

Most of the mismatches come from changes in the main source code. In `ElegantRL` and `gpt-researcher`, this is because the main code is being updated and improved a lot. In `MaxText`, the source code also has many mismatches, but the `test` files show more changes too, probably because they are often used for experiments and checking the code.

- Visualization of Myers vs Histogram diffs across repositories, showing mismatch ratio, average and spread of diff lengths, and filetype discrepancies.



- and coming to which algorithm is better i checked the files where Myers and Histogram diffs do not match (Discrepancy = Yes) and Counted the number of lines in each diff, Shorter diffs usually show the real changes more clearly so The algorithm with shorter average lines in mismatched commits is usually better.

| Repo           | Myers lines | Histogram lines | Better algorithm |
|----------------|-------------|-----------------|------------------|
| maxtext        | 390.43      | 537.61          | Myers            |
| elegantl       | 904.05      | 942.19          | Myers            |
| gpt-researcher | 173.34      | 312.88          | Myers            |

Table 6: Comparison of lines processed by Myers and Histogram algorithms

this is just an easier implimentation of how it works on these 3 repos so we could see that by this basic approach myers always worked better.

For a more better approach ,To find which algorithm is better, we look only at mismatches (cases where Myers and Histogram give different results). Then we check simple measures: number of lines in the diff (shorter is better), number of hunks or change blocks (fewer is cleaner), and sometimes semantic similarity (how close the diff is to real code changes). After that, we take averages of these measures for each algorithm and compare them. The algorithm with shorter lines, fewer hunks, and higher similarity is considered better. We can also check this by file type (source, test, README, LICENSE). To make it automatic, we can use a scoring formula:

$$Score = w_1 \times (lines) + w_2 \times (hunks) - w_3 \times (similarity)$$

The algorithm with the lowest score is the better one.

## DISCUSSION AND CONCLUSION

The histogram diff sometimes showed differences just because of formatting, which created false discrepancies. We also had to clean the diffs to remove metadata and whitespace. Processing large repositories took more time, which is challenging.

From this work, I learned how to use **PyDriller** to go through commits and extract diffs. I also learned how to compare different diff algorithms, categorize files, and analyze the results.

Overall, the Myers and Histogram algorithms gave mostly similar results, but the Myers works better using the diff length in my 3 repos. but there were still some mismatches, around 31% to 66% depending on the repository. Most of these mismatches happened in source code files.

## REFERENCES

- [1] Lab Document [Shared on Google Classroom]
- [2] Link to my Repo: [https://github.com/ps-keerthana/STT\\_1\\_4](https://github.com/ps-keerthana/STT_1_4)
- [3] <https://git-scm.com/docs/git-diff>
- [4] <https://github.com/ishepard/pydriller>
- [5] <https://doi.org/10.1007/s10664-019-09772-z>
- [6] <https://github.com/yusufsn/DifferentDiffAlgorithms>
- [7] SEART GitHub Search Engine