Assignment-1

Software Tools and Techniques for CSE

Shardul Junagade (23110297)
Repository: cs202-stt

September 8, 2025

Contents

Lab 1: Introduction to Version Controlling, Git Workflows, and Actions

Repository Link: cs202-stt/lab1

1.1 Introduction, Setup, and Tools

1.1.1 Introduction

The objective of this lab was to gain hands-on experience with Version Control Systems (VCS), specifically Git, and to explore the integration of GitHub with automated workflows such as GitHub Actions. My aim was to understand fundamental concepts like repositories, commits, branches, and remotes, and to perform essential Git operations. Additionally, I set up a continuous integration workflow using GitHub Actions and Pylint to ensure code quality.

1.1.2 Setup and Tools

I logged in to my GitHub account in the web browser – ShardulJunagade – for version control. I installed Git and Visual Studio Code by downloading the installers from their official websites and following the installation instructions. After installation, I verified the installation by checking the versions of Git and Visual Studio Code.

The following is a list of the tools and technologies used in this lab:

- Operating System: Windows 11
- Terminal: PowerShell 7
- Git version: 2.42.0
- Code Editor: Visual Studio Code
- Python version: 3.13.7
- GitHub for remote repository hosting
- GitHub Actions for CI/CD pipelines

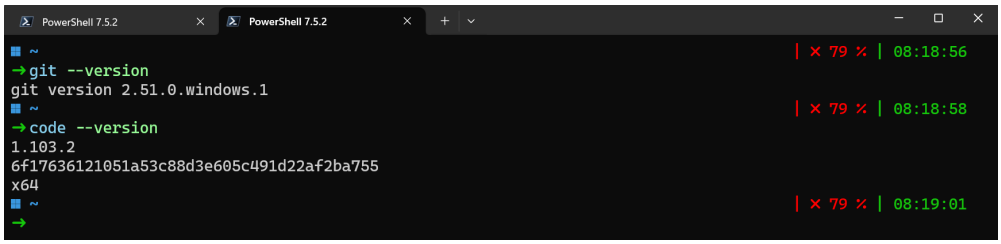


Figure 1: Verifying Git installation and version.

1.2 Methodology and Execution

1.2.1 Initializing a Local Repository

I created a new folder for the lab, named `Any_Name` using the command `mkdir Any_Name` and initialized the folder as a git repository using the command `git init`.

Figure 2: Initializing a new Git repository.

1.2.2 Local Git Configuration

Since my global Git configuration was already set up from previous work, I decided to configure my name and email specifically for this repository using the local configuration commands `git config user.name` and `git config user.email`. This ensures that all commits in this repository are attributed with the correct identity, regardless of the global settings.

Figure 3: Configuring local Git user details.

1.2.3 Adding and Committing Files

I created a new file named `README.md` and used the command `git status` to inspect the repository. The output indicated one untracked file. I then staged the file with `git add README.md` and confirmed the change with another `git status`, which now showed the file as staged.

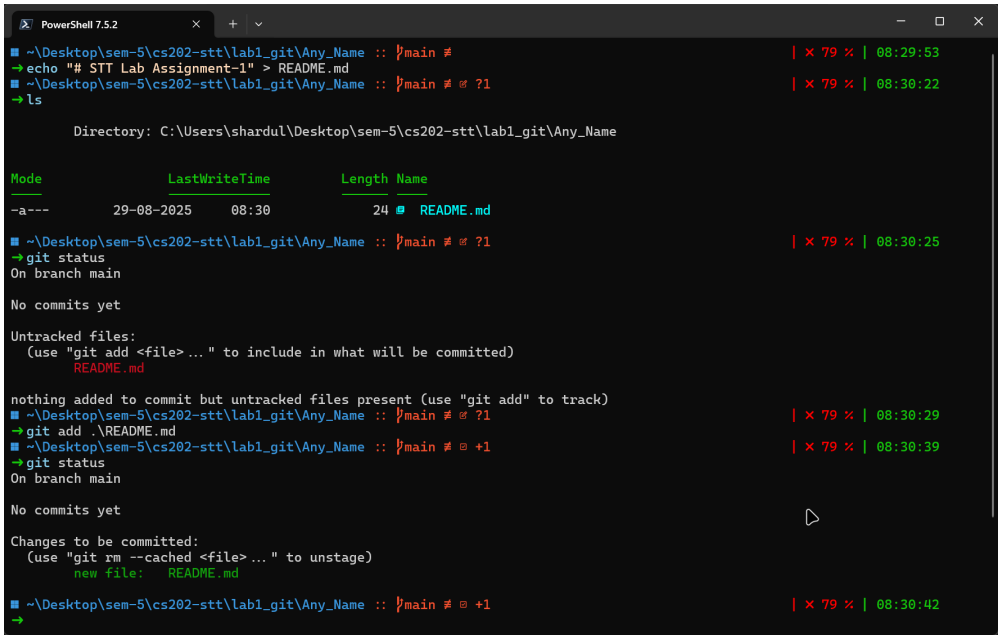


Figure 4: Staging the README.md file for commit.

Next, I committed the staged file using the command `git commit -m "Initial commit: Added README.md"`. To verify, I executed `git log`, which displayed the commit details along with the message and metadata.

Figure 5: Committing the file and viewing the commit history.

1.2.4 Working with Remote Repository

I created a remote repository on GitHub named `Any_Name`.

Repository Link: `https://github.com/ShardulJunagade/Any_Name`

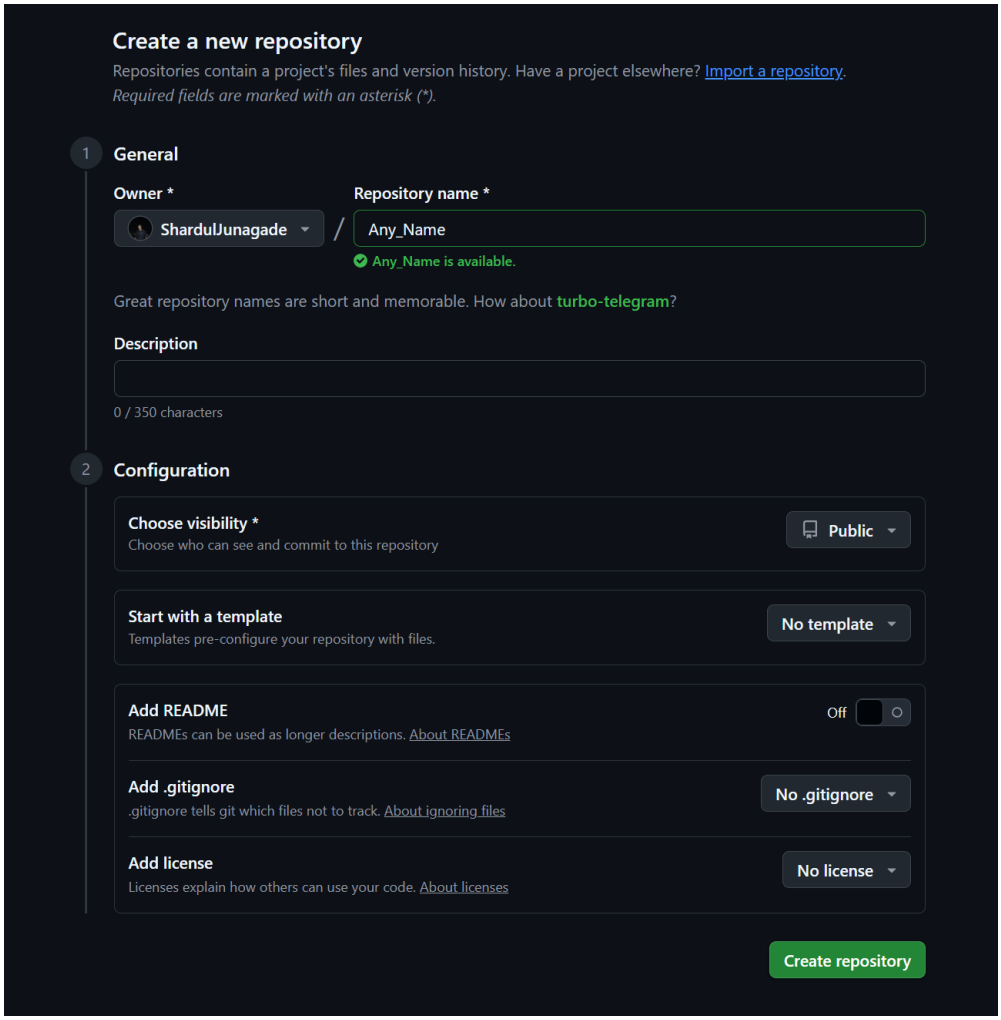


Figure 6: Creating a new repository on GitHub.

After that, I linked the remote repository to my local repository using the command `git remote add origin <URL>`. Once the connection was established, I pushed my local commits to GitHub with `git push -u origin main`.

Figure 7: Adding the remote and pushing local commits to GitHub.

To confirm that everything worked correctly, I opened the repository on GitHub in my browser and verified that all the changes were successfully reflected online. We can see that the README.md file is present.

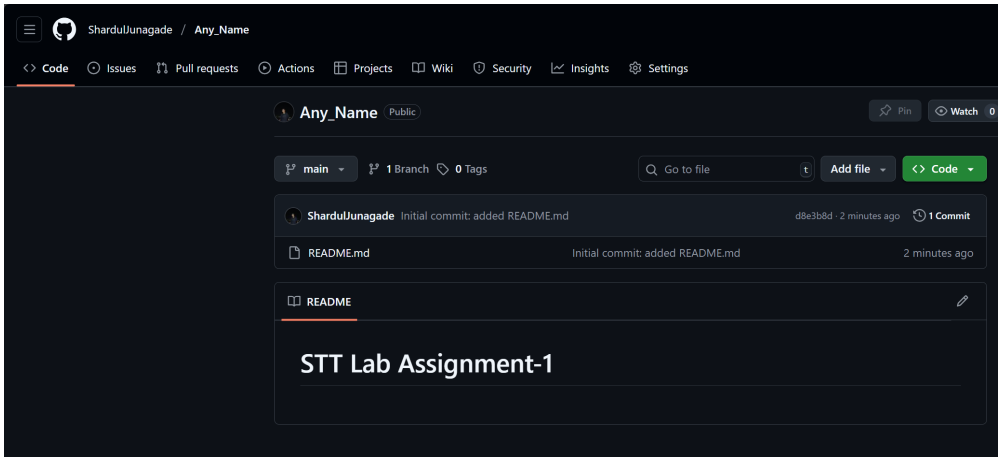


Figure 8: Verifying updates on GitHub.

For the next step, I cloned the repository using the command `git clone <URL>`, and we can see that the README.md file is present in the freshly cloned repo.

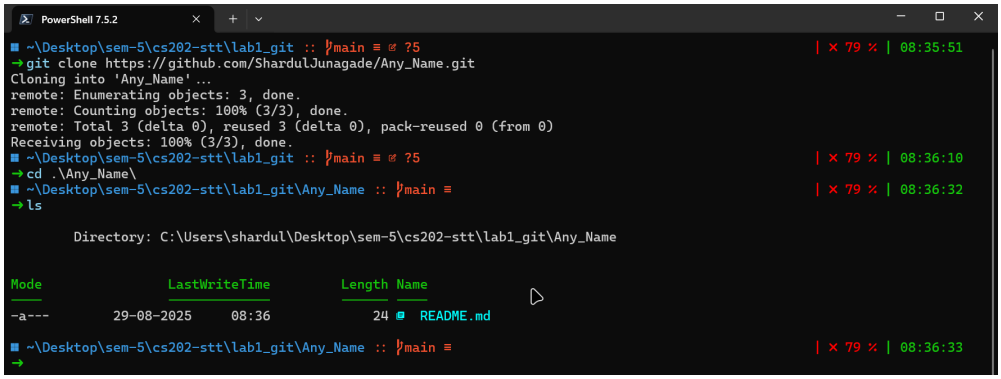


Figure 9: Cloning a repository from GitHub.

Then, I moved to GitHub in my web browser and added a new file named `main.py` that contains a simple Python program. I committed the changes directly on GitHub to the remote repository. This file was now visible in the remote repository on GitHub but not in my local repository.

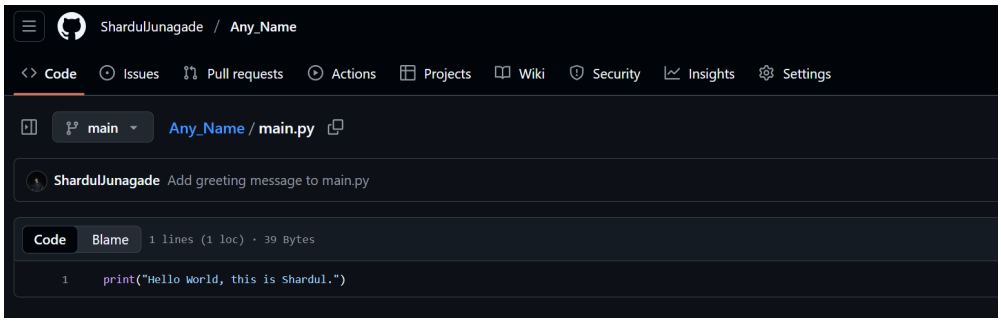


Figure 10: Adding a greeting message in `main.py` via GitHub.

Now, I had to pull these changes/commits from the remote repository to my local repository. To do this I executed the command `git pull origin main` to fetch and merge the changes. This pulled the new file into my local repository, keeping both versions in sync.

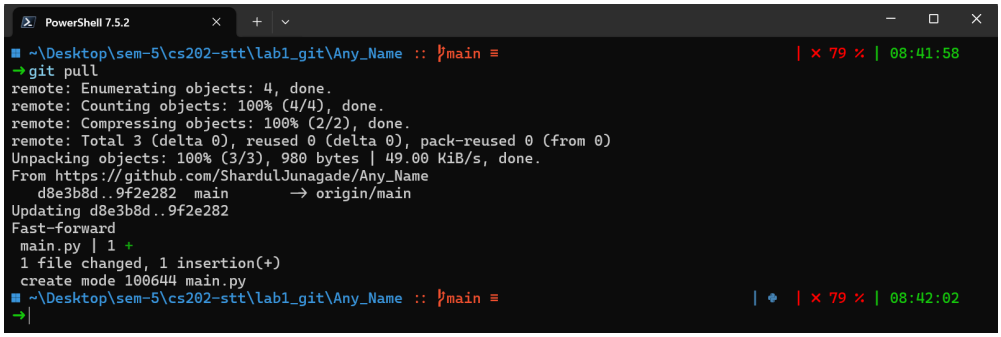


Figure 11: Pulling the latest changes from GitHub.

1.2.5 GitHub Actions – Pylint Workflow

The next task was to automate code quality checks using Pylint.

For this task, I wrote a Python file called `app.py` that contained a simple calculator program, which you can see below:

Figure 12: Initial version of `app.py` (calculator code).

After saving the file, I staged, committed, and pushed it to GitHub.

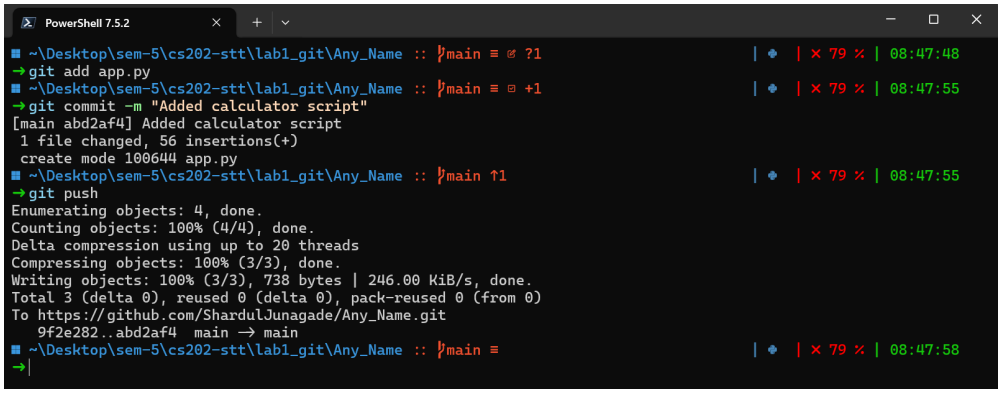


Figure 13: Pushing the calculator code to GitHub.

On GitHub, there was a section of suggested workflows for GitHub Actions. Here, I selected the Pylint workflow and clicked on `Configure`.

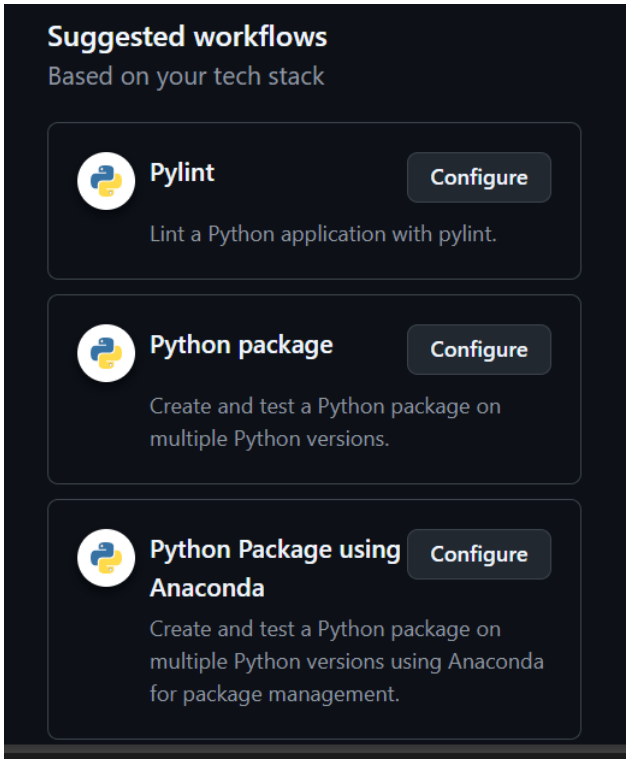


Figure 14: Suggested workflows for GitHub Actions.

Then, I created the `.github/workflows/Pylint.yml` file to enable linting on every push, as shown below:

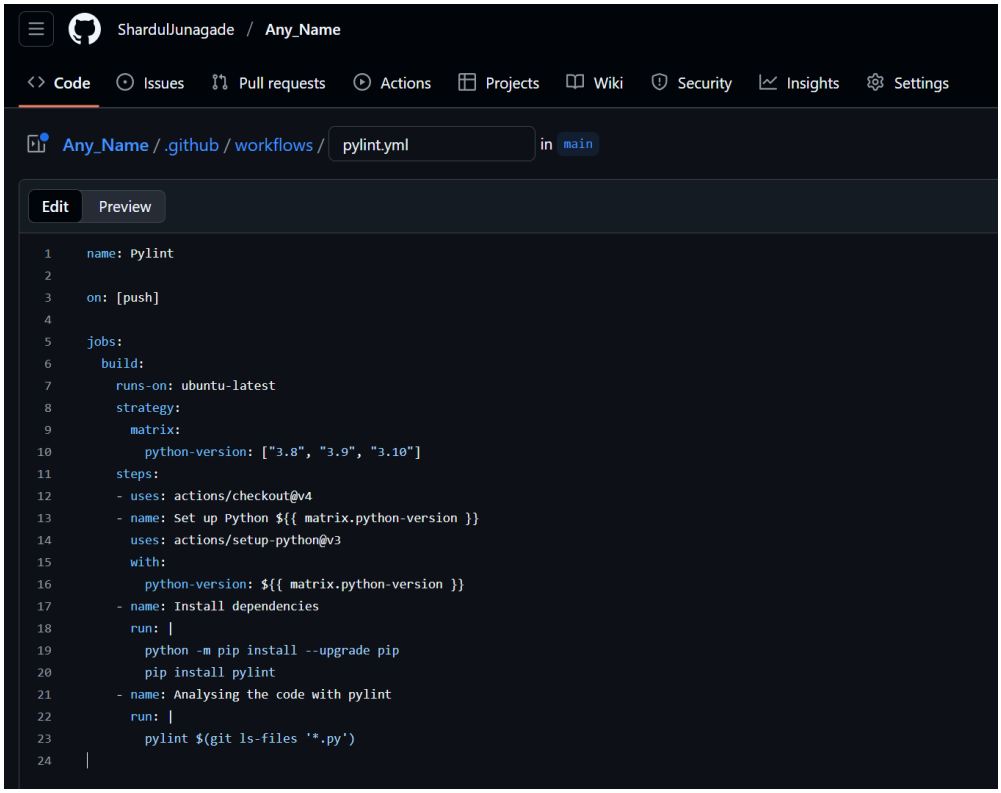


Figure 15: Creating the `Pylint.yml` workflow file.

As soon as I committed the workflow file, GitHub Actions automatically triggered the workflow. The first run failed because Pylint reported several errors in my code, which can be seen below:

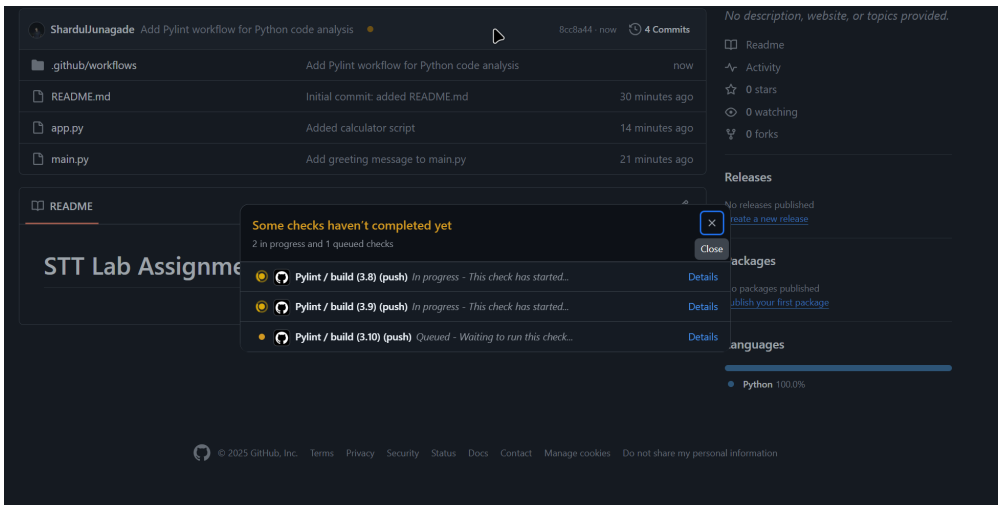


Figure 16: GitHub Actions running Pylint workflow.

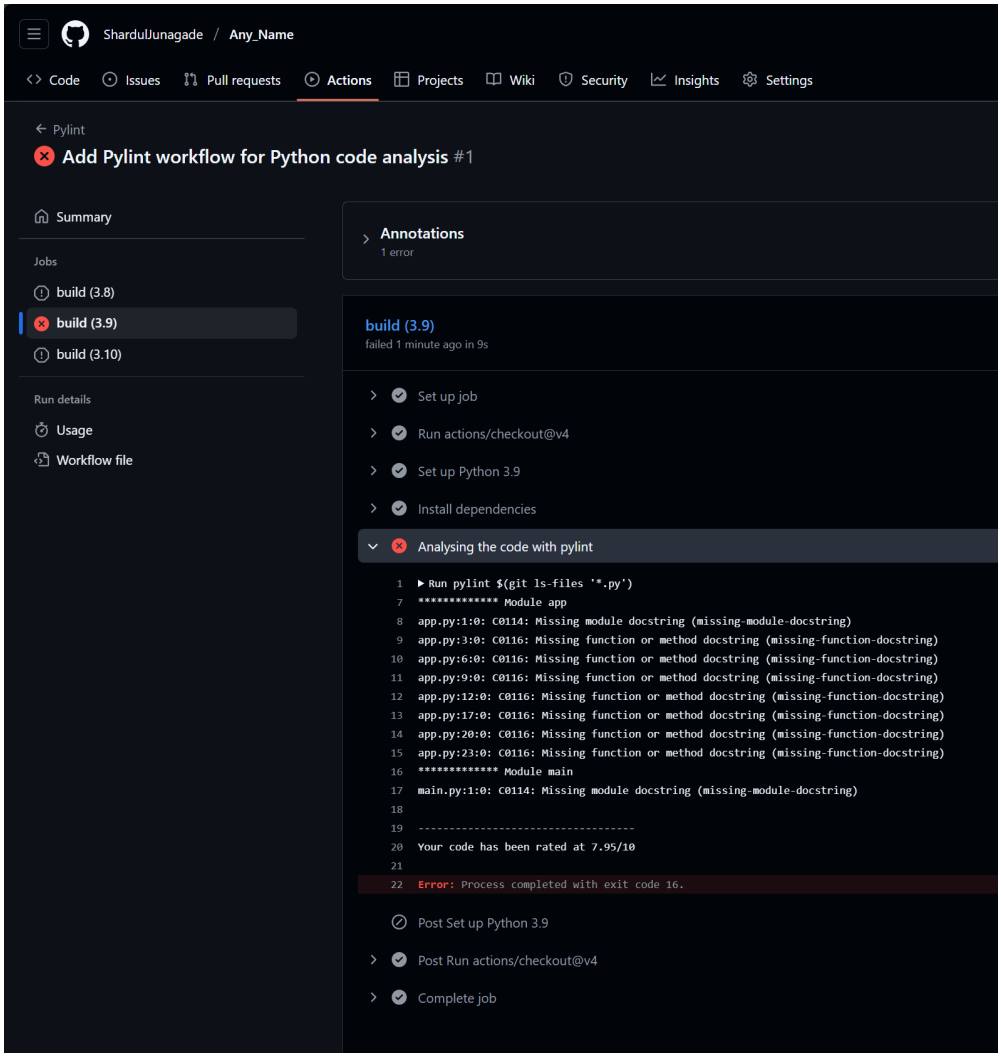


Figure 17: Initial Pylint errors detected.

After reviewing the error messages, I fixed the issues by adding proper docstrings and correcting other bugs. I committed and pushed the corrected code to the repository.

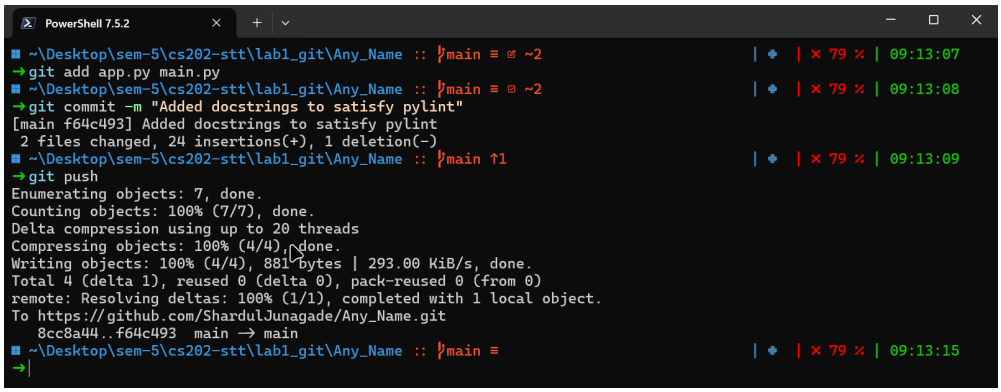


Figure 19: Pushing the corrected code.

This time, the workflow passed successfully, and GitHub showed a green tick, which confirmed that my code met the required standards.

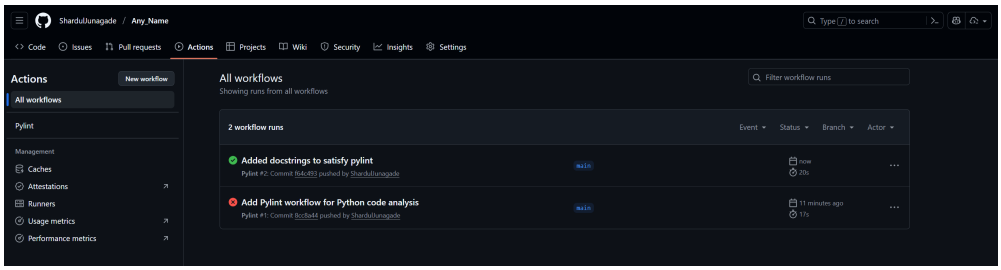


Figure 20: Successful Pylint workflow run.

All three workflow runs completed successfully, as indicated by the green tick on each build in the GitHub Actions tab.

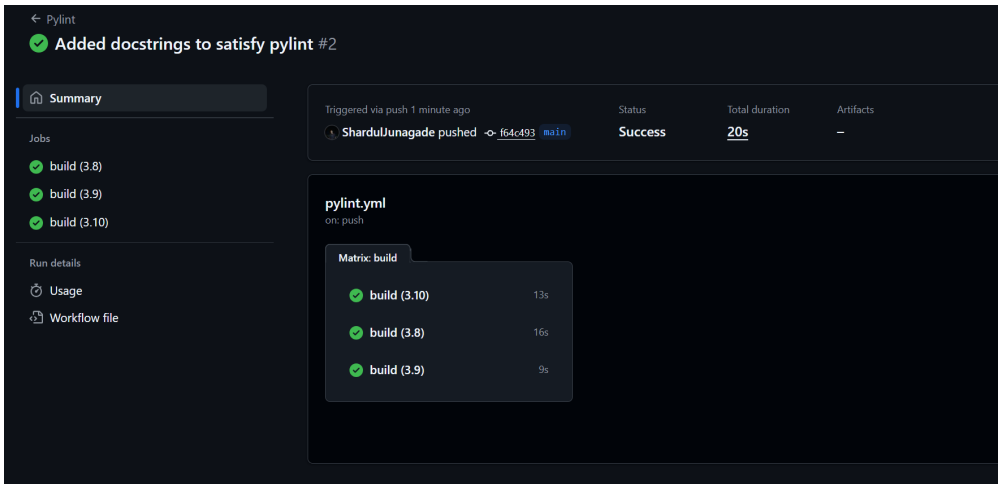


Figure 21: Workflow completion with a green tick.

1.3 Results and Analysis

Through this lab, I successfully created and managed a Git repository, synchronized changes with GitHub, and demonstrated cloning and pulling operations. The GitHub Actions workflow validated my Python code using Pylint, ensuring it followed coding standards and confirming the results with a green tick on GitHub.

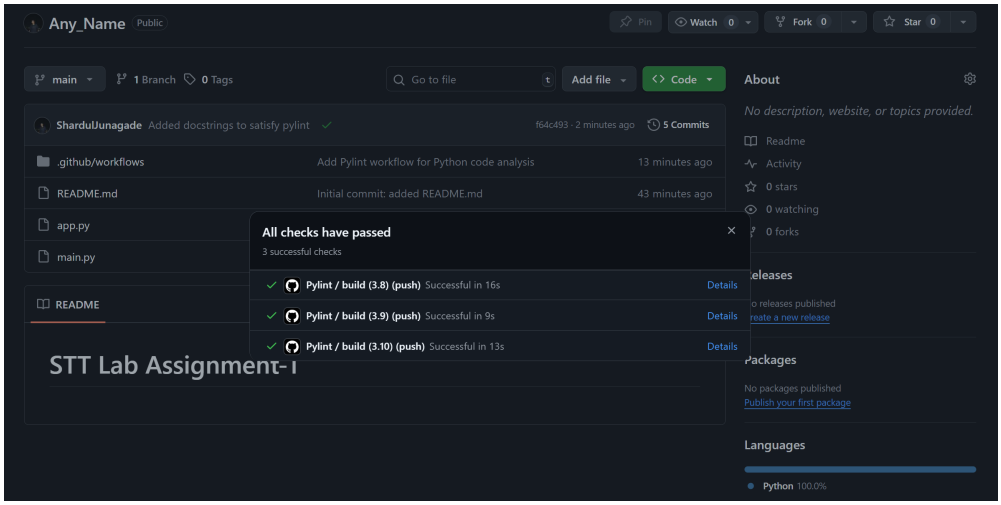


Figure 22: Successful Pylint validation and workflow status.

The final commit history below shows the whole process – from the first commit to fixing errors and reaching the successful workflow run.

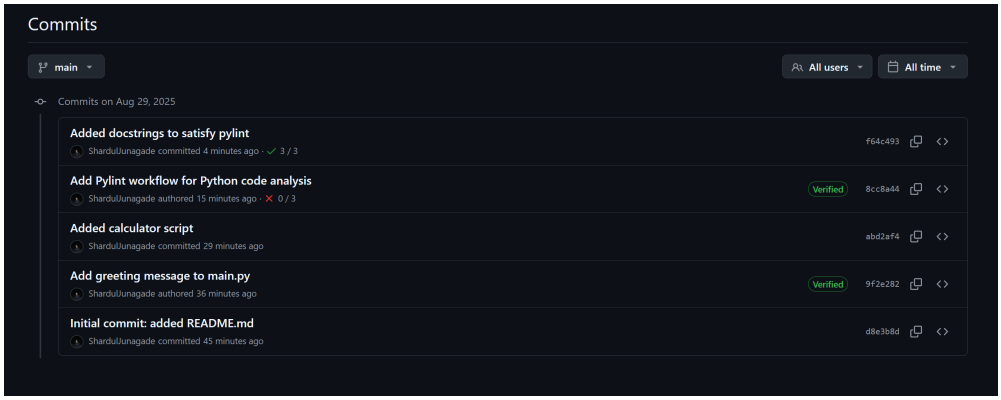


Figure 23: Final commit history showing all changes.

Task	**Outcome**
Initialize repository	Repository created successfully (Figure 2-3)
Push to GitHub	Changes pushed and visible online (Figures 4–8)
Clone and Pull	Repository cloned and synced correctly (Figures 9–11)
GitHub Actions (Pylint)	Failed first due to docstring issues, later passed with green tick (Figures 12–21)

1.4 Discussion and Conclusion

This lab gave me hands-on practice with Git, starting from creating an empty repository to adding files, staging, committing, pushing to GitHub, cloning repositories, and pulling updates. One of the main challenges I faced was the branch mismatch issue (main vs. master), which caused errors during my first push.

I also found interpreting the Pylint workflow's yaml file slightly tricky at first and even though the Python code was running fine, the workflow still failed due to docstring and formatting issues which were caught by Pylint. Going through the error messages and fixing them was a valuable learning experience. The most satisfying moment was finally seeing the green tick on GitHub after the corrected code passed Pylint.

Overall, this lab gave me a nice overview of how Git and GitHub work together and showed me how CI/CD pipelines can give instant feedback on code quality, something that is very useful in real-world projects where many developers work together.

1.5 References

[1] Git Documentation

[2] GitHub Guides

[3] Git Cheat Sheet

[4] Pylint Documentation

[5] Lab Document (Shared on Google Classroom)

Lab 2: Mining Bug-Fixing Commits, LLM Inference, Rectifier, and Evaluation

Repository Link: cs202-stt/lab2

2.1 Introduction, Setup, and Tools

2.1.1 Introduction

This lab focused on mining open-source repositories to study bug-fixing commits and commit message alignment. I implemented a pipeline to:

1. Identify bug-fixing commits from a real-world project.
2. Extract file-level diffs from those commits.
3. Use a pre-trained LLM to generate concise summaries for each file-level change.
4. Rectify those messages to make them more precise and context-aware.
5. Evaluate the quality of developer, LLM, and rectified messages using semantic similarity with CodeBERT.

The motivation behind this pipeline was that commit messages are not always reliable indicators of what a change actually fixes. Developers may batch multiple fixes, write vague messages, or skip details. Automated tools and rectifiers can help make these messages more consistent and useful.

2.1.2 Environment and Tools

- **Operating System:** Windows 11
- **Terminal:** Powershell 7
- **Python:** 3.13.7
- **PyTorch:** 2.8 (with CUDA 12.9)
- **Transformers:** 4.56
- **PyDriller:** 2.8
- **Models used:**
  - mamiksik/CommitPredictorT5 (LLM Inference)
  - codellama:7b via Ollama (Rectifier)
  - microsoft/codebert-base (Evaluation)
- **Repository analyzed:** 3b1b/manim

```
Python version: 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)]
Using device: cuda
PyTorch version: 2.8.0+cu129
CUDA version: 12.9
Device name: NVIDIA GeForce RTX 4060 Laptop GPU
PyDriller version: 2.8
Transformers version: 4.56.0
```

Environment Details

2.2 Methodology and Execution

2.2.1 Repository Selection

For this lab, I chose the repository 3b1b/manim. Manim (short for Mathematical Animation Engine) is a Python library that started with Grant Sanderson's 3Blue1Brown channel and has since grown into a large open-source project. It is mainly used to create mathematical animations and visualizations, and because of its popularity it now has a wide contributor base and frequent updates.

I felt Manim was a good choice because it's not just an academic toy project but a tool actually used by educators, researchers, and content creators. That also means its commit history has plenty of real bug fixes to study, which fits well with the aim of this assignment.

Selection Criteria

Repository Statistics

While narrowing down the repository, I kept the following points in mind:

1. **Number of commits:** Manim has more than **6300 commits**, which is large enough to give me enough bug-fixing commits for analysis.
2. **Popularity:** With around **78k stars** and **6.7k forks**, the project has a huge user base and community involvement, so the data is representative of real usage.
3. **Programming language:** The project is written in **Python**, which works well since the lab tools like PyDriller and radon are also Python-based.
4. **Relevance:** Since Manim deals with mathematical visualization and graphics, correctness and stability are very important. That makes bug-fixing commits here especially meaningful to analyze.
5. **Active development:** The repo is still very active, with the last commit in June 2025 and contributions from over **160 developers**, showing that the project is maintained and evolving.

Based on these reasons, Manim seemed like a balanced and practical choice for carrying out this lab. Then I cloned the repository locally using the `git clone <URL>` command.

```
# clone the repositories if not already cloned
if not os.path.exists(repo_name):
    print(f"Cloning {repo_name} from {REPO_URL}...")
    subprocess.run(["git", "clone", REPO_URL])
else:
    print(f"Repository {repo_name} already exists. Skipping clone.")
```

```
Repository manim already exists. Skipping clone.
```

Git Clone

2.2.2 Identifying Bug-Fixing Commits

Notebook Link: bugfix_commits.ipynb

I first defined a heuristic to detect bug-fixing commits. I scanned commit messages for keywords such as:

fix, bug, patch, error, issue, defect, crash, flaw, repair, resolve, solve, fail, leak, vulnerability

This simple keyword filter is fast and transparent. The downside is that it may miss commits where developers did not explicitly mention a bug (false negatives), or capture irrelevant commits where the keyword appeared casually (false positives).

Using PyDriller, I traversed the commit history of the **manim** repository and stored each matching commit in a CSV (`bugfix_commits.csv`) with the following fields:

- Commit hash
- Commit message
- Parent hashes
- Is merge commit?
- Modified files

The following code snippet shows the implementation:

```
bug_keywords = ['fix', 'bug', 'patch', 'error', 'issue', 'defect', 'crash', 'fault', 'flaw',
                'glitch', 'mistake', 'repair', 'resolve', 'solve', 'fail', 'break', 'broke',
                'overflow', 'leak', 'vulnerability']

def is_bugfix(msg):
    msg = msg.lower()
    return any(word in msg for word in bug_keywords)
```

```
fields = ['commit_hash', 'commit_message', 'parent_hashes', 'is_merge_commit', 'modified_files']

with open(f"{output_folder}/{output_csv}", 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=fields)
    writer.writeheader()

    commits_list = list(Repository(repo_name).traverse_commits())
    for commit in tqdm(commits_list, desc="Processing commits"):
        if is_bugfix(commit.msg):
            writer.writerow({
                'commit_hash': commit.hash,
                'commit_message': commit.msg.replace('\n', ' ').replace('\r', ' '),
                'parent_hashes': '; '.join(commit.parents),
                'is_merge_commit': commit.merge,
                'modified_files': '; '.join([mf.filename for mf in commit.modified_files])
            })

print(f"Bug-fixing commits written to {output_folder}/{output_csv}")
```

```
Processing commits: 100%|██████████| 6344/6344 [02:48<00:00, 37.66it/s]
Bug-fixing commits written to bugfix_commits.csv
```

Code for extracting Bug Fixing Commits

I traversed 6,344 commits, out of which, the keyword filter flagged 1358 as bug-fix candidates (21%).

Number of bug-fixing commits found: 1358

	commit_hash	commit_message	parent_hashes	is_merge_commit	modified_files
0	014a277a97759bbc0e6ec8fba588bc6	A few fixes to initial point_thickness i	c0994ed0a53f06a555ea8d42903c203	False	constants.py; displayer.py; mobject.p
1	2e074afb60d13262ce1e42e83bcf0ed	middle of massive restructure, everyt	096c5c1890e326e67ee387c921901f8	False	__init__.py; __init__.py; animation.py;
2	ac930952f151acf284f6e01e98e8f725	Beginning transform KA article, many	d294d0f951f01b307805d18679509c4	False	animation.py; meta_animations.py; si
3	2322018875b218cc156f50d30d865a4	quick rgb-should-be-numpy-array b	c7389e008d3dc79f61e8270ff187582	False	mobject.py
4	7ae5a0eccb13713b3439d9c0a0b79b	Slightly faster sort_points method, ar	7f45044bafb27c94ae1b3f900014b3f	False	mobject.py
5	f21f6619a5150d193d85ed31ef60ed5	Fix to Stars Mobject	7ae5a0eccb13713b3439d9c0a0b79b	False	geometry.py; three_dimensions.py
6	68160140233b8f7ebf00aa2894ea579	Bug fix to bug fix on Mobject.fade m	8f8eeea870c2f780cc93950a6de6bd6	False	mobject.py
7	4f57551344e2655e53c9dcf72ee7d17	Bug in Arrow buffer	68160140233b8f7ebf00aa2894ea579	False	geometry.py
8	5964d5206ac15fc0544325728e232b6	Fixed ShowCreation	84054286824d0ca17219f7d3e11a5f2	False	simple_animations.py
9	9e54a5a6802d12994b007434b98c9f3	Fixed PiCreature mouth, but more w	c0009064d6ee0183f2d36fd15d7442b	False	characters.py

20 rows x 5 cols 10 per page « ‹ Page 1 of 2 › »

Bug Fixing Commits

2.2.3 Extracting File-level Diffs

Notebook Link: diff_extract_and_llm_infer.ipynb

Since commits often modify multiple files, I processed each file separately. For each bug-fixing commit, I extracted the before and after source codes for each modified file and also stored metadata such as filename, change type, and the git diff.

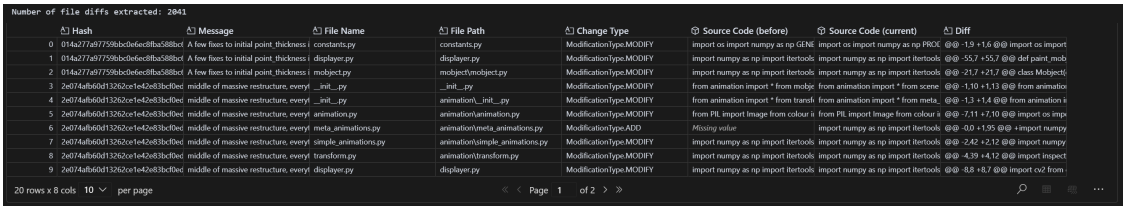
The following image shows the code implementation:

```python
diffs_per_file_csv = 'diffs_per_file.csv'

fields = [
    'Hash','Message', 'File Name','File Path',
    'Change Type','Source Code (before)','Source Code (current)','Diff',
]

with open(f"{output_folder}/{diffs_per_file_csv}", 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=fields)
    writer.writeheader()

    commits_list = list(Repository(repo_name).traverse_commits())
    for commit in tqdm(commits_list, desc="Processing commits"):
        if not is_bugfix(commit.msg):
            continue
        for m in commit.modified_files:
            if m.diff is None or m.diff.strip() == '':
                continue
            writer.writerow({
                'Hash': commit.hash,
                'Message': commit.msg.replace('\n', ' ').replace('\r', ' '),
                'File Name': m.filename,
                'File Path': m.new_path or m.old_path,
                'Change Type': str(m.change_type),
                'Source Code (before)': m.source_code_before or '',
                'Source Code (current)': m.source_code or '',
                'Diff': m.diff,
            })
print(f"Diffs per file written to {output_folder}/{diffs_per_file_csv}")
```

```
Processing commits: 100%|██████████| 6344/6344 [02:45<00:00, 38.33it/s]
Diffs per file written to results/diffs_per_file.csv
```

Code for extracting Per-File Diffs

After running the code, I extracted 2041 file-level diffs and saved these entries to `diffs_per_file.csv`.



Number of file diffs extracted: 2041

	Hash	Message	File Name	File Path	Change Type	Source Code (before)	Source Code (current)	Diff
0	014a277a97759bbc0e6ec8fba588bcf	A few fixes to initial point_thickness i	constants.py	constants.py	ModificationType.MODIFY	import os import numpy as np GENE	import os import numpy as np PROC	@@ -1,9 +1,6 @@ import os import
1	014a277a97759bbc0e6ec8fba588bcf	A few fixes to initial point_thickness i	displayer.py	displayer.py	ModificationType.MODIFY	import numpy as np import itertools	import numpy as np import itertools	@@ -55,7 +55,7 @@ def paint_mob
2	014a277a97759bbc0e6ec8fba588bcf	A few fixes to initial point_thickness i	mobject.py	mobject\mobject.py	ModificationType.MODIFY	import numpy as np import itertools	import numpy as np import itertools	@@ -21,7 +21,7 @@ class Mobject(
3	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	__init__.py	__init__.py	ModificationType.MODIFY	from animation import * from mobje	from animation import * from scene	@@ -1,10 +1,13 @@ from animatio
4	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	__init__.py	animation__init__.py	ModificationType.MODIFY	from animation import * from transfo	from animation import * from meta_	@@ -1,3 +1,4 @@ from animation i
5	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	animation.py	animation\animation.py	ModificationType.MODIFY	from PIL import Image from colour i	from PIL import Image from colour i	@@ -7,11 +7,10 @@ import os imp
6	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	meta_animations.py	animation\meta_animations.py	ModificationType.ADD	*Missing value*	import numpy as np import itertools	@@ -0,0 +1,95 @@ +import numpy
7	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	simple_animations.py	animation\simple_animations.py	ModificationType.MODIFY	import numpy as np import itertools	import numpy as np import itertools	@@ -2,42 +2,12 @@ import numpy
8	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	transform.py	animation\transform.py	ModificationType.MODIFY	import numpy as np import itertools	import numpy as np import itertools	@@ -4,39 +4,12 @@ import inspect
9	2e074afb60d13262ce1e42e83bcf0ec	middle of massive restructure, everyt	displayer.py	displayer.py	ModificationType.MODIFY	import numpy as np import itertools	import numpy as np import itertools	@@ -8,8 +8,7 @@ import cv2 from

20 rows x 8 cols 10 per page Page 1 of 2

File-level Diffs

2.2.4 LLM Inference of "fix type"

Notebook Link: diff_extract_and_llm_infer.ipynb

I used the Hugging Face model **CommitPredictorT5** to infer the type of fix from the diff. I gave the following prompt template to the pretrained model.

```
File: <filename>
Diff: <diff>
```

The model generated concise commit-style summaries (max length = 64 tokens). These were appended to the CSV as an extra column: *LLM Inference (fix type)* and I saved this dataset to `diffs_per_file_with_llm_infer.csv`. This allowed direct comparison between the developer written commit messages and the LLM predictions.

The following code snippet shows the implementation:

```python
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
diffs_per_file_llm_infer_csv = 'diffs_per_file_with_llm_infer.csv'

MODEL = "mamiksik/CommitPredictorT5"
tokenizer = AutoTokenizer.from_pretrained(MODEL)
model = AutoModelForSeq2SeqLM.from_pretrained(MODEL).to(device)

def prepare_prompt(file_path, diff):
    if not isinstance(diff, str) or diff.strip() == '':
        diff = '<NO DIFF AVAILABLE>'
    return f"File: {file_path}\nDiff:\n{diff}"

def llm_infer(filepath, diff):
    prompt = prepare_prompt(filepath, diff)
    inputs = tokenizer(prompt, return_tensors='pt', truncation=True, max_length=2048).to(device)
    with torch.no_grad():
        gen = model.generate(**inputs, max_length=64)
    llm_msg = tokenizer.decode(gen[0], skip_special_tokens=True)
    return llm_msg

with open(f"{output_folder}/{diffs_per_file_llm_infer_csv}", 'w', newline='', encoding='utf-8') as f:
    writer = csv.DictWriter(f, fieldnames=fields + ['LLM Inference (fix type)'])
    writer.writeheader()

    for _, row in tqdm(diffs_per_file_df.iterrows(), total=len(diffs_per_file_df), desc="LLM Inference"):
        llm_msg = llm_infer(row['File Path'], row['Diff'])
        writer.writerow({
            **row.to_dict(),
            'LLM Inference (fix type)': llm_msg
        })

print(f"Diffs with LLM inference written to {output_folder}/{diffs_per_file_llm_infer_csv}")
```

```
LLM Inference:   0%|          | 0/2041 [00:00<?, ?it/s]
LLM Inference: 100%|██████████| 2041/2041 [20:19<00:00,  1.67it/s]
Diffs with LLM inference written to results/diffs_per_file_with_llm_infer.csv
```

Code for LLM Inference

Number of file diffs with LLM inference: 2041

	Hash	Message	File Name	File Path	Change Type	Source Code (before)	Source Code (current)	Diff	LLM Inference (fix type)
0	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	constants.py	constants.py	ModificationType.MODIFY	import os\nimport numpy as np\n\n\nGENERALLY_B...	import os\nimport numpy as np\n\n\nPRODUCTION_QU...	@@ -1,9 +1,6 @@\n import os\n import numpy as ...	add missing constants
1	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	displayer.py	displayer.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -55,7 +55,7 @@ def paint_mobjects(mobjects,...	add nudge to displayer.py
2	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	mobject.py	mobject\mobject.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -21,7 +21,7 @@ class Mobject(object):\n ...	add missing docstring
3	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	__init__.py	__init__.py	ModificationType.MODIFY	from animation import *\nfrom mobject import *...	from animation import *\nfrom scene import *\n...	@@ -1,10 +1,13 @@\n from animation import *\n+...	add missing import statements
4	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	__init__.py	animation__init__.py	ModificationType.MODIFY	from animation import *\nfrom transform import...	from animation import *\nfrom meta_animations ...	@@ -1,3 +1,4 @@\n from animation import *\n-fr...	add missing newline
5	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	animation.py	animation\animation.py	ModificationType.MODIFY	from PIL import Image\nfrom colour import Colo...	from PIL import Image\nfrom colour import Colo...	@@ -7,11 +7,10 @@\n import os\n import copy\n im...	add missing config to missing color animation
6	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	meta_animations.py	animation\meta_animations.py	ModificationType.ADD	NaN	import numpy as np\nimport itertools as it\nfr...	@@ -0,0 +1,95 @@\n+import numpy as np\n+import...	add tests for animation.update and animation.u...
7	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	simple_animations.py	animation\simple_animations.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nfr...	import numpy as np\nimport itertools as it\nfr...	@@ -2,42 +2,12 @@\n import numpy as np\n import ...	add some more classes to the animation class

LLM Inference Samples

2.2.5 Rectifier Formulation

Notebook Link: ollama_rectifier.ipynb

Developer messages and LLM outputs can still be vague or misaligned, especially when multiple files are involved. Many times developers may not clearly specify the bug or issue being addressed. Developers often combine multiple changes/fixes in a single commit and the LLM may not capture all relevant context. To improve clarity, I designed a rectifier with the following rules:

- Input: file name, change type, diff, and optionally developer + LLM messages.
- Output style: `[file]: Fix <bug/issue> in <component> by <specific action>`
- Keep messages short (< 20 words) and avoid vague verbs.

I implemented this rectifier with **Ollama + codellama:7b** for local inference on my NVIDIA RTX 4060 machine and added 1 more column named *Rectified Message* to the CSV for the rectified messages. I saved the results to `ollama_rectified_commits.csv`.

The following image shows the code implementation for the rectifier:

```python
def rectify_message(row):
    file_name = str(row.get("File Name", "file"))
    dev_msg = str(row.get("Message", ""))
    llm_msg = str(row.get("LLM Inference (Fix Type)", ""))
    diff = str(row.get("Diff", ""))
    change_type = str(row.get("Change Type", ""))

    prompt = f"""
You are refining commit messages for bug-fixing commits.

File: {file_name}
Change type: {change_type}

Git Diff:
{diff}

Developer's commit message: {dev_msg if dev_msg else "N/A"}
LLM inference: {llm_msg if llm_msg else "N/A"}

Task:
Generate a **detailed rectified commit message** in this style:
[{file_name}]: Fix [bug/issue/etc] in [function/component] by [specific action].
Keep under 20 words, precise, and avoid vague terms like 'update' or 'change'.
"""

    try:
        response = ollama.chat(model="codellama:7b", messages=[
            {"role": "user", "content": prompt}
        ])
        rectified = response["message"]["content"].strip()
        return rectified if rectified else f"{file_name}: Minor {change_type}"
    except Exception as e:
        return f"{file_name}: Minor update"
```

Code for Rectifier

Hash	Message	File Name	File Path	Change Type	Source Code (before)	Source Code (current)	Diff	LLM Inference (fix type)	Rectified Message
3c6e6de65a86	A few fixes to initial point_thickness impleme...	constants.py	constants.py	ModificationType.MODIFY	import os\nimport numpy as np\n\n\nGENERALLY_B...	import os\nimport numpy as np\n\nPRODUCTION_QU...	@@ -1,9 +1,6 @@\n import os\n import numpy as ...	add missing constants	[constants.py]: Fix default point thickness to...
3c6e6de65a86	A few fixes to initial point_thickness impleme...	displayer.py	displayer.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -55,7 +55,7 @@ def paint_mobjects(mobjects,...	add nudge to displayer.py	[displayer.py]: Fix potential offset issue in ...
3c6e6de65a86	A few fixes to initial point_thickness impleme...	mobject.py	mobject\mobject.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -21,7 +21,7 @@ class Mobject(object):\n ...	add missing docstring	[mobject.py]: Fix inconsistent point_thickness...
1ed28d95ad82	middle of massive restructure, everything stil...	__init__.py	__init__.py	ModificationType.MODIFY	from animation import *\nfrom mobject import *...	from animation import *\nfrom scene import *\n...	@@ -1,10 +1,13 @@\n from animation import *\n-...	add missing import statements	[__init__.py]: Fix [bug/issue/etc] in [functio...
1ed28d95ad82	middle of massive restructure, everything stil...	__init__.py	animation__init__.py	ModificationType.MODIFY	from animation import *\nfrom transform import...	from animation import *\nfrom meta_animations ...	@@ -1,3 +1,4 @@\n from animation import *\n-fr...	add missing newline	Fix regression in transform module by adding m...
1ed28d95ad82	middle of massive restructure, everything stil...	animation.py	animation\animation.py	ModificationType.MODIFY	from PIL import Image\nfrom colour import Colo...	from PIL import Image\nfrom colour import Colo...	@@ -7,11 +7,10 @@\n import os\n import copy\n im...	add missing config to missing color animation	[animation.py]: Fix bug in writeGif function b...
1ed28d95ad82	middle of massive restructure, everything stil...	meta_animations.py	animation\meta_animations.py	ModificationType.ADD	NaN	import numpy as np\nimport itertools as it\nfr...	@@ -0,0 +1,95 @@\n+import numpy as np\n+import...	add tests for animation.update and animation.u...	[meta_animations.py]: Fix bug in DelayByOrder ...

Rectifier Sample Results

2.2.6 Evaluation with CodeBERT

Notebook Link: ollama_evaluation.ipynb

To measure how well each message aligned with its code change, I used **microsoft/codebert-base**. First, I generated embeddings for both the diff text and the corresponding messages (developer, LLM, and rectified). Then, I computed the cosine similarity between these embeddings to quantify how semantically close each message was to the actual code change. In this setup, a higher similarity score indicates a stronger alignment between the text and the code.

I decided to keep the threshold for precision at 0.9. Any score greater than 0.9 was considered "precise." Using this rule, I could compute the hit rate for each category of message and answer the three research questions (RQ1–RQ3). The results were saved to `ollama_scores_codebert.csv`. The following image shows the code implementation for the evaluation:

```python
from transformers import RobertaTokenizer, RobertaModel

MODEL_NAME = "microsoft/codebert-base"
tokenizer = RobertaTokenizer.from_pretrained(MODEL_NAME)
model = RobertaModel.from_pretrained(MODEL_NAME).to(device)


def get_code_embedding(code_snippet):
    tokens = tokenizer(code_snippet, return_tensors="pt", truncation=True, padding=True, max_length=512)
    inputs = {key: val.to(device) for key, val in tokens.items()}
    with torch.no_grad():
        outputs = model(**inputs)
    # Use the [CLS] token representation as the embedding
    cls_embedding = outputs.last_hidden_state[:, 0, :]
    return cls_embedding

def cosine_sim(vec1, vec2):
    return F.cosine_similarity(vec1, vec2).item()

def score(code, msg):
    if not msg.strip() or not code.strip():
        return 0
    code_emb = get_code_embedding(code)
    msg_emb = get_code_embedding(msg)
    return cosine_sim(code_emb, msg_emb)

def evaluate_with_codebert(df):
    print("Evaluating with CodeBERT...")
    print("\nScoring original commit messages...")
    df["dev_score"] = df.progress_apply(lambda r: score(r["Diff"], r["Message"]), axis=1)
    print("\nScoring LLM inference messages...")
    df["llm_inference_score"] = df.progress_apply(lambda r: score(r["Diff"], r["LLM Inference (fix type)"]), axis=1)
    print("\nScoring rectified messages...")
    df["rectifier_score"] = df.progress_apply(lambda r: score(r["Diff"], r["Rectified Message"]), axis=1)
    return df
```

Code for Evaluation

	Hash	File Name	dev_score	llm_inference_score	rectifier_score
0	014a277a97759bbc0e6ec8fba588bc6e6de65a86	constants.py	0.962658	0.942259	0.976323
1	014a277a97759bbc0e6ec8fba588bc6e6de65a86	displayer.py	0.918488	0.902997	0.941550
2	014a277a97759bbc0e6ec8fba588bc6e6de65a86	mobject.py	0.937761	0.921378	0.959316
3	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	__init__.py	0.942789	0.933025	0.963519
4	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	__init__.py	0.970627	0.962571	0.969677
5	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	animation.py	0.953300	0.954102	0.969746
6	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	meta_animations.py	0.897986	0.911280	0.977757
7	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	simple_animations.py	0.892042	0.898379	0.903630
8	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	transform.py	0.889994	0.870787	0.966823
9	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	displayer.py	0.911511	0.924990	0.937381
10	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	extract_scene.py	0.913572	0.917471	0.941413
11	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	helpers.py	0.889769	0.906124	0.927119
12	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	image_mobject.py	0.907444	0.911569	0.917670
13	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	images2gif.py	0.883028	0.872721	0.963090
14	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	mobject.py	0.959424	0.955415	0.977553
15	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	__init__.py	0.968253	0.959519	0.970230
16	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	complex_multiplication_article.py	0.900738	0.912112	0.918707
17	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	generate_logo.py	0.955927	0.955502	0.972877
18	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	moser_main.py	0.975104	0.965233	0.985425
19	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	region.py	0.954561	0.950039	0.982617

CodeBERT Scores

```
RQ1 - Developer precise hit rate: 91.28%
RQ2 - LLM precise hit rate: 93.39%
RQ3 - Rectifier precise hit rate: 100.00%
```

Mean Hit Rates

We can see that the developer messages had a precision hit rate of 91%, the LLM inference messages had a hit rate of 92%, and the rectified messages improved the hit rate to 100%. This indicates that the rectifier was effective in enhancing the alignment between the messages and the code changes.

2.3 Results and Analysis

Bug-fixing commits found: 1358
File-level diffs extracted: 2041

Following are few examples of developer messages, LLM inferences, and my rectified messages:

Developer Message	LLM Inference	Rectified Message
A few fixes to initial point_thickness implementation	add missing constants	[constants.py]: Fix default point thickness to be 4 in DEFAULT_POINT_THICKNESS by specifying it explicitly.
A few fixes to initial point_thickness implementation	add nudge to displayer.py	[displayer.py]: Fix potential offset issue in point_thickness by adjusting the thickness of a plus-sign-shaped pixel arrangement to ensure correct rendering on high-quality displays.
A few fixes to initial point_thickness implementation	add missing docstring	[mobject.py]: Fix inconsistent point_thickness implementation in Mobject1D and Mobject2D by specifying a default value for the attribute.

RQ	Message Type	Hit Rate (threshold = 0.9)
RQ1	Developer messages	∼91%
RQ2	LLM inference	∼93%
RQ3	Rectified messages	100%

Developer messages were often short and lacked detail, lowering their alignment scores. The LLM inference was generally good but sometimes missed context that the rectifier captured.

My rectifier is able to consistently produce high-quality, precise messages. This is mainly because of 3 reasons:

1. **Better Large Language Model (LLM):** The use of a more advanced LLM for inference likely contributed to the improved message quality. The LLM was able to better understand the context and nuances of the code changes, resulting in more accurate and relevant messages.
2. **Better Context:** By using the specific file name and change type as part of the input, the rectifier can generate messages that are more closely aligned with the actual code changes being made. This helps to reduce ambiguity and improve precision.
3. **Structured Output:** The output format of the rectifier is designed to be concise and specific, which helps to ensure that the messages are clear and actionable.

2.4 Discussion and Conclusion

2.4.1 Challenges faced

During the lab, I faced a few challenges that slowed me down initially. PyDriller, for example, was a new library for me, and I needed some time to get comfortable with its API and how to extract the right commit-level information. Another difficulty came from the keyword-based heuristic I used to identify bug-fixing commits. While it worked reasonably well, it sometimes missed commits where developers didn't explicitly mention bug-related terms, and on the other hand it also pulled in a few extra commits that weren't true bug fixes. Finally, token limits posed a practical issue – some of the larger diffs had to be truncated before being passed to the model, and this occasionally hurt the accuracy of the LLM's generated summaries.

2.4.2 Lessons learned

Working through these issues taught me a few important lessons. I found that analyzing changes at the file level and then applying rectification added real value, because it made commit messages more precise and easier to interpret. I also realized that building a pipeline by combining several tools – PyDriller for mining, Hugging Face models for inference, Ollama for rectification, and CodeBERT for evaluation – can be powerful, but it also demands a lot of care in data handling. Even small oversights, like inconsistent CSV column names, can break later steps in the workflow.

2.4.3 Conclusion

Overall, the end-to-end pipeline – from mining commits to generating diffs, running LLM inference, rectifying the outputs, and finally evaluating them – proved to be quite workable. The rectifier in particular helped improve the quality of commit messages, often making them more specific and useful than both the original developer-written messages and the raw LLM predictions.

2.5 References

[1] PyDriller
[2] Hugging Face Transformers
[3] CodeBERT (microsoft/codebert-base)
[4] CommitPredictorT5 (mamiksik/CommitPredictorT5)
[5] Ollama
[6] Repository analyzed (manim)
[7] Lab Document: Google Doc

Lab 3: Multi-Metric Bug Context Analysis and Agreement Detection in Bug-Fix Commits

Repository Link: cs202-stt/lab3

3.1 Introduction, Setup, and Tools

3.1.1 Introduction

In Lab 2, I had prepared a per-file dataset of bug-fix commits with extracted diffs and model-generated summaries. The purpose of this lab was to analyse the relation between structural code quality and magnitude of changes in bug-fix commits. Specifically, I aimed to investigate:

- Structural metrics around each fix (Maintainability Index, Cyclomatic Complexity, and Lines of Code) using radon.
- Change magnitude metrics (Semantic similarity with CodeBERT and Token similarity with BLEU).
- Classify each fix as Major or Minor from both lenses and check where they agree (or don't).

By combining these, I classified the bug fixes as Major or Minor and checked where the structural and semantic lenses agreed or conflicted. This is important because commit messages or diffs alone rarely capture how "big" or "complex" a change really is.

3.1.2 Environment and Tools

- OS: Windows 11, Terminal: PowerShell 7
- Code Editor: Visual Studio Code - Insiders
- Python: 3.13.7
- Key packages: radon, nltk, transformers, torch, scikit-learn
- Models: microsoft/codebert-base (for embeddings)
- Hardware: NVIDIA RTX 4060 Laptop GPU

```
Python version: 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)]
Using device: cuda
PyTorch version: 2.8.0+cu129
CUDA version: 12.9
Device name: NVIDIA GeForce RTX 4060 Laptop GPU

radon: 6.0.1
transformers: 4.56.0
scikit-learn: 1.7.1
numpy: 2.3.2
pandas: 2.3.2
tqdm: 4.67.1
nltk: 3.9.1
```

Environment Setup

3.2 Methodology and Execution

Notebook Link: lab3.ipynb

3.2.1 Starting point (Lab 2 dataset)

- Input CSV: `lab3/lab2_diffs.csv`
- Columns included: Commit Hash, Message, File Name, Source Code (before), Source Code (current), Diff, LLM Inference (fix type), Rectified Message

I first loaded the dataset, checked the first 10–20 rows, and verified that there were no missing critical columns. Then, I checked for NaNs in key columns in the dataset.

	Hash	Message	File Name	File Path	Change Type	Source Code (before)	Source Code (current)	Diff	LLM Inference (fix type)	Rectified Message
	ɔc6e6de65a86	A few fixes to initial point_thickness impleme...	constants.py	constants.py	ModificationType.MODIFY	import os\nimport numpy as np\n\n\nGENERALLY_B...	import os\nimport numpy as np\n\nPRODUCTION_QU...	@@ -1,9 +1,6 @@\n import os\n import numpy as ...	add missing constants	[constants.py]: Fix default point thickness to...
	ɔc6e6de65a86	A few fixes to initial point_thickness impleme...	displayer.py	displayer.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -55,7 +55,7 @@ def paint_mobjects(mobjects,...	add nudge to displayer.py	[displayer.py]: Fix potential offset issue in ...
	ɔc6e6de65a86	A few fixes to initial point_thickness impleme...	mobject.py	mobject\mobject.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -21,7 +21,7 @@ class Mobject(object):\n ...	add missing docstring	[mobject.py]: Fix inconsistent point_thickness...
	led28d95ad82	middle of massive restructure, everything stil...	__init__.py	__init__.py	ModificationType.MODIFY	from animation import *\nfrom mobject import *...	from animation import *\nfrom scene import *\n...	@@ -1,10 +1,13 @@\n from animation import *\n- ...	add missing import statements	[__init__.py]: Fix [bug/issue/etc] in [functio...
	led28d95ad82	middle of massive restructure, everything stil...	__init__.py	animation__init__.py	ModificationType.MODIFY	from animation import *\nfrom transform import...	from animation import *\nfrom meta_animations ...	@@ -1,3 +1,4 @@\n from animation import *\n- fr...	add missing newline	Fix regression in transform module by adding m...
	led28d95ad82	middle of massive restructure, everything stil...	animation.py	animation\animation.py	ModificationType.MODIFY	from PIL import Image\nfrom colour import Colo...	from PIL import Image\nfrom colour import Colo...	@@ -7,11 +7,10 @@ import os\n import copy\n im...	add missing config to missing color animation	[animation.py]: Fix bug in writeGif function b...
	led28d95ad82	middle of massive restructure, everything stil...	meta_animations.py	animation\meta_animations.py	ModificationType.ADD	NaN	import numpy as np\nimport itertools as it\nfr...	@@ -0,0 +1,95 @@\n+import numpy as np\n+@@\n+import ...	add tests for animation.update and animation.u...	[meta_animations.py]: Fix bug in DelayByOrder ...
	led28d95ad82	middle of massive restructure, everything stil...	simple_animations.py	animation\simple_animations.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nfr...	import numpy as np\nimport itertools as it\nfr...	@@ -2,42 +2,12 @@\n import numpy as np\n+ import ...	add some more classes to the animation class	[simple_animations.py]: Refactor commit messag...
	led28d95ad82	middle of massive restructure, everything stil...	transform.py	animation\transform.py	ModificationType.MODIFY	import numpy as np\nimport itertools as it\nim...	import numpy as np\nimport itertools as it\nim...	@@ -4,39 +4,12 @@\n import inspect\n import copy...	add new animation classes	[transform.py]: Fix broken behavior in path_al...

Dataset Preview

```
# check for nan values in the df
print("Checking for NaN values in each column:")
print(df.isna().sum())
```

[39]

```
Checking for NaN values in each column:
Hash                          0
Message                       0
File Name                     0
File Path                     0
Change Type                   0
Source Code (before)         52
Source Code (current)        35
Diff                          0
LLM Inference (fix type)      0
Rectified Message             0
dtype: int64
```

NaN Counts

3.2.2 Baseline Descriptive Stats

From the dataset, I computed the following statistics to understand the dataset:

- Total unique commits and total file entries.
- Average modified files per commit.
- Distribution of fix types from `LLM Inference (fix type)`.
- Most frequently modified filenames and extensions.

The following images show the code and output for these computations:

```python
total_commits = df['Hash'].nunique()        # total number of unique commits
total_files = df.shape[0]                   # total number files
avg_files_per_commit = total_files / total_commits      # average number of files per commit
# avg_files_per_commit_pd = df.groupby('Hash').size().mean()

# Distribution of fix types
fix_type_distribution = df["LLM Inference (fix type)"].value_counts(dropna=False)

# Most frequently modified filenames and extensions
most_modified_filenames = df['File Name'].value_counts()
df['file_extension'] = df['File Name'].apply(
    lambda x: '.' + x.split('.')[-1] if pd.notna(x) and '.' in x else 'no_extension'
)
file_extension_distribution = df['file_extension'].value_counts()


print(f"Total number of unique commits: {total_commits}")
print(f"Total number of files: {total_files}")
print(f"Average number of modified files per commit (manual calc): {avg_files_per_commit:.2f}")
# print(f"Average number of modified files per commit (pandas calc): {avg_files_per_commit_pd:.2f}")
print("\nDistribution of fix types:")
print(fix_type_distribution.head(10))
print("\nMost frequently modified filenames:")
print(most_modified_filenames.head(10))
print("\nDistribution of file extensions:")
print(file_extension_distribution.head(10))
```

Baseline Stats Code

```
Total number of unique commits: 1121
Total number of files: 2041
Average number of modified files per commit (manual calc): 1.82

Distribution of fix types:
LLM Inference (fix type)
add missing docstring           161
add missing import               95
add missing imports              39
add missing docstrings           29
add missing class attributes     28
add missing comments             27
add missing comment              18
add missing config file          17
update camera.py                 15
add missing config               13
Name: count, dtype: int64
```

Baseline Stats

```
Most frequently modified filenames:
File Name
vectorized_mobject.py    120
mobject.py               113
scene.py                  87
geometry.py               83
camera.py                 60
svg_mobject.py            56
tex_mobject.py            47
config.py                 44
constants.py              36
coordinate_systems.py     36
Name: count, dtype: int64

Distribution of file extensions:
file_extension
.py           1809
.glsl           78
.rst            47
.yml            34
.md             27
.txt             9
.mp4             8
.svg             7
.gitignore       6
.tex             6
Name: count, dtype: int64
```

Baseline Stats

3.2.3 Structural metrics with radon

For each file, I ran radon on both "before" and "current" versions of the source code and recorded the following structural metrics:

- **Maintainability Index (MI)** – A composite score that combines factors like lines of code, complexity, and comments to indicate how easy a piece of code is to maintain. A higher MI usually means the code is more readable and maintainable.
- **Cyclomatic Complexity (CC)** – A measure of how many independent paths exist through the code, essentially capturing the decision points (like if/else, loops). Higher CC means the code is more complex and harder to test thoroughly.
- **Lines of Code (LOC)** – The raw number of lines in the code. While simple, this metric is a direct measure of the size of the code and often correlates with the effort required to understand or modify it.

I then computed their deltas: `MI_Change`, `CC_Change`, `LOC_Change`. I caught any parsing exceptions and recorded them as NaN (those propagate to "Unknown" later). I saved these results to `results/structural_metrics.csv`.

The following image show the code implementation for structural metrics computation:

```python
def compute_radon_metrics(code):
    """Compute radon metrics: MI, CC, LOC for given source code."""
    if not isinstance(code, str) or code.strip() == "":
        return (np.nan, np.nan, np.nan)

    try:
        # Maintainability Index
        mi = float(mi_visit(code, multi=True))
    except Exception as e:
        mi = np.nan

    try:
        # Cyclomatic Complexity
        cc_results = cc_visit(code)
        cc = np.mean([block.complexity for block in cc_results]) if cc_results else 0.0
    except Exception as e:
        cc = np.nan

    try:
        # Lines of Code
        loc = radon_analyze(code).loc
    except Exception as e:
        loc = np.nan

    return (mi, cc, loc)


# Compute radon metrics for each file before and after the commit
df[['MI_Before', 'CC_Before', 'LOC_Before']] = df['Source Code (before)'].progress_apply(
    lambda x: pd.Series(compute_radon_metrics(x))
)
df[['MI_After', 'CC_After', 'LOC_After']] = df['Source Code (current)'].progress_apply(
    lambda x: pd.Series(compute_radon_metrics(x))
)

# Compute changes in metrics
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(f"{output_folder}/structural_metrics.csv", index=False)
```

Code snippet for structural metrics computation

	Hash	Message	MI_Before	MI_After	MI_Change	CC_Before	CC_After	CC_Change	LOC_Before	LOC_After	LOC_Change
0	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	56.611516	56.611516	0.000000	0.000000	0.000000	0.0	104.0	103.0	-1.0
1	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	NaN	NaN	NaN	NaN	NaN	NaN	230.0	238.0	8.0
2	014a277a97759bbc0e6ec8fba588bc6e6de65a86	A few fixes to initial point_thickness impleme...	NaN	NaN	NaN	NaN	NaN	NaN	455.0	455.0	0.0
3	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	100.000000	100.000000	0.000000	0.000000	0.000000	0.0	10.0	13.0	3.0
4	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	100.000000	100.000000	0.000000	0.000000	0.000000	0.0	3.0	4.0	1.0
5	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	58.276524	58.495906	0.219382	1.714286	1.714286	0.0	122.0	120.0	-2.0
6	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	58.959139	NaN	NaN	3.100000	NaN	NaN	95.0	NaN
7	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	59.449158	NaN	NaN	2.176471	NaN	233.0	102.0	-131.0
8	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	51.775973	NaN	NaN	1.961538	NaN	235.0	171.0	-64.0
9	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	198.0	184.0	-14.0
10	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	181.0	189.0	8.0
11	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	264.0	296.0	32.0
12	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	49.664046	50.088220	0.424174	2.500000	2.900000	0.4	157.0	137.0	-20.0
13	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	845.0	NaN	NaN
14	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	463.0	461.0	-2.0
15	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	100.000000	NaN	NaN	0.000000	NaN	NaN	6.0	NaN	NaN
16	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	392.0	NaN	NaN
17	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	66.0	62.0	-4.0
18	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	NaN	NaN	NaN	NaN	NaN	NaN	1706.0	1701.0	-5.0
19	2e074afb60d13262ce1e42e83bcf0ed28d95ad82	middle of massive restructure, everything stil...	64.537035	64.742479	0.205444	2.153846	2.153846	0.0	128.0	128.0	0.0

Preview of structural_metrics.csv

3.2.4 Change magnitude: semantic vs token similarity

To understand how much the code changed between the *before* and *after* versions, I measured change magnitude using two complementary metrics:

- **Semantic similarity** – Computed using CodeBERT embeddings with cosine similarity. This captures whether the two versions of the code still mean the same thing, even if the surface-level tokens look different.
- **Token similarity** – Measured using BLEU with NLTK's tokenizer (with smoothing). This focuses on how closely the literal tokens match between the two code snippets, making it sensitive to formatting and small textual edits.

I added 2 columns for these values to the dataframe and saved the dataset to `results/change_magnitude_metrics.csv`.

The following images show the code implementation for calculating the semantic similarity and token similarity:

```python
import math
from transformers import AutoTokenizer, AutoModel
from sklearn.metrics.pairwise import cosine_similarity
import nltk

nltk.download("punkt")
nltk.download("punkt_tab")

# Load CodeBERT model and tokenizer
tokenizer = AutoTokenizer.from_pretrained("microsoft/codebert-base")
model = AutoModel.from_pretrained("microsoft/codebert-base").to(device)
model.eval()

def safe_str(s):
    if s is None:
        return ""
    if isinstance(s, float) and math.isnan(s):
        return ""
    return str(s)

def mean_pooling(model_output, attention_mask):
    token_embeddings = model_output.last_hidden_state
    input_mask_expanded = attention_mask.unsqueeze(-1).expand(token_embeddings.size()).float()
    return torch.sum(token_embeddings * input_mask_expanded, 1) / torch.clamp(input_mask_expanded.sum(1), min=1e-9)

def compute_codebert_embedding(code):
    """Compute CodeBERT embedding for given source code."""
    if not isinstance(code, str) or code.strip() == "":
        return np.zeros((768,))  # Return zero vector for empty code

    inputs = tokenizer(code, return_tensors="pt", truncation=True, padding=True, max_length=512)
    inputs = {k: v.to(device) for k, v in inputs.items()}
    with torch.no_grad():
        outputs = model(**inputs)
    # embeddings = outputs.last_hidden_state[:, 0, :].squeeze().cpu().numpy()
    embeddings = mean_pooling(outputs, inputs['attention_mask']).squeeze().cpu().numpy()
    return embeddings

def compute_semantic_similarity(row):
    """Compute semantic similarity between before and after code using CodeBERT embeddings."""
    before_code = safe_str(row['Source Code (before)'])
    after_code = safe_str(row['Source Code (current)'])

    before_embedding = compute_codebert_embedding(before_code)
    after_embedding = compute_codebert_embedding(after_code)

    if np.linalg.norm(before_embedding) == 0 or np.linalg.norm(after_embedding) == 0:
        return np.nan  # Return NaN if either embedding is zero vector

    similarity = cosine_similarity([before_embedding], [after_embedding])[0][0]
    return similarity
```

Code snippet for semantic similarity

```python
def compute_token_similarity(row):
    """Compute token similarity between before and after code using BLEU score."""
    before_code = safe_str(row['Source Code (before)'])
    after_code = safe_str(row['Source Code (current)'])

    if not before_code or not after_code:
        return np.nan  # Return NaN if either code is empty

    before_tokens = nltk.word_tokenize(before_code)
    after_tokens = nltk.word_tokenize(after_code)

    if len(before_tokens) == 0 or len(after_tokens) == 0:
        return np.nan  # Return NaN if tokenization results in empty lists

    smooth = nltk.translate.bleu_score.SmoothingFunction()
    bleu_score = nltk.translate.bleu_score.sentence_bleu([before_tokens], after_tokens, smoothing_function=smooth.method1)
    return bleu_score
```

Code snippet for token similarity

	Message	File Name	Semantic_Similarity	Token_Similarity
0	A few fixes to initial point_thickness impleme...	constants.py	0.999283	0.986973
1	A few fixes to initial point_thickness impleme...	displayer.py	1.000000	0.952839
2	A few fixes to initial point_thickness impleme...	mobject.py	0.999760	0.998018
3	middle of massive restructure, everything stil...	__init__.py	0.997906	0.660184
4	middle of massive restructure, everything stil...	__init__.py	0.996067	0.701206
5	middle of massive restructure, everything stil...	animation.py	0.999739	0.980059
6	middle of massive restructure, everything stil...	meta_animations.py	NaN	NaN
7	middle of massive restructure, everything stil...	simple_animations.py	0.994631	0.252018
8	middle of massive restructure, everything stil...	transform.py	0.993418	0.636833
9	middle of massive restructure, everything stil...	displayer.py	0.999916	0.894411
10	middle of massive restructure, everything stil...	extract_scene.py	0.999588	0.918873
11	middle of massive restructure, everything stil...	helpers.py	1.000000	0.894109
12	middle of massive restructure, everything stil...	image_mobject.py	0.999869	0.868611
13	middle of massive restructure, everything stil...	images2gif.py	NaN	NaN
14	middle of massive restructure, everything stil...	mobject.py	0.999880	0.996650
15	middle of massive restructure, everything stil...	__init__.py	NaN	NaN
16	middle of massive restructure, everything stil...	complex_multiplication_article.py	0.997389	0.655118
17	middle of massive restructure, everything stil...	generate_logo.py	0.999659	0.943530
18	middle of massive restructure, everything stil...	moser_main.py	1.000000	0.995177
19	middle of massive restructure, everything stil...	region.py	0.999849	0.989585

Table preview showing Semantic_Similarity and Token_Similarity

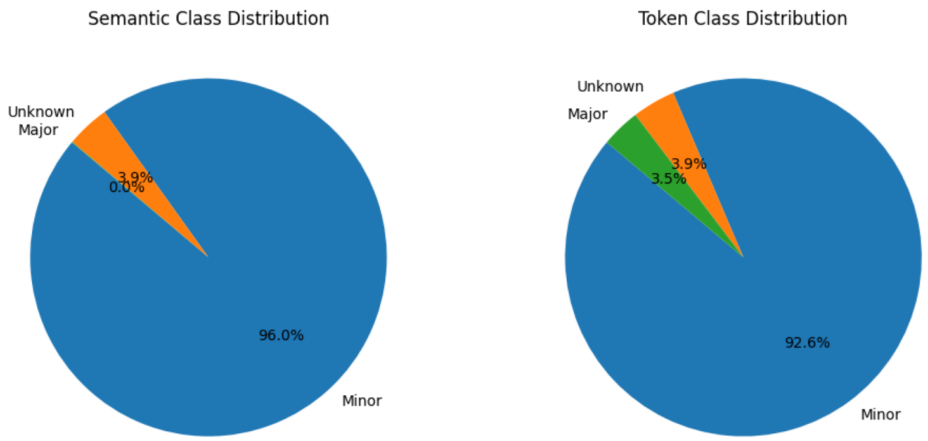
3.2.5 Classification and agreement

After computing the similarity scores, I mapped each bug-fix commit into categories of *Major* or *Minor* using simple threshold rules. This step helped in comparing how the two metrics align in their judgment of the same change.

- Semantic similarity $\geq 0.80 \Rightarrow$ *Minor*, else *Major*
- Token similarity $\geq 0.75 \Rightarrow$ *Minor*, else *Major*
- Unknown: if the metric could not be computed (NaN), the classification was recorded as *Unknown*.

```python
df = pd.read_csv(f"{output_folder}/change_magnitude_metrics.csv")

# taken from lab pdf
SEMANTIC_THRESHOLD = 0.80
TOKEN_THRESHOLD = 0.75

def classify_change(val, threshold):
    # Semantic class
    if not (isinstance(val, float) or isinstance(val, int)) or np.isnan(val):
        return 'Unknown'
    elif val >= threshold:
        return 'Minor'
    else:
        return 'Major'

df['Semantic_Class'] = df['Semantic_Similarity'].apply(lambda x: classify_change(x, SEMANTIC_THRESHOLD))
df['Token_Class'] = df['Token_Similarity'].apply(lambda x: classify_change(x, TOKEN_THRESHOLD))

df.to_csv(f"{output_folder}/final_metrics.csv", index=False)
```

Code snippet for classification



Class Distribution of Semantic_Class and Token_Class

Then, I compared the two classifications:

- If both matched, Classes_Agree = YES
- If they differed, Classes_Agree = NO
- If either was Unknown, then agreement was also Unknown

```python
def check_agreement(row):
    if row['Semantic_Class'] == 'Unknown' or row['Token_Class'] == 'Unknown':
        return 'Unknown'
    return 'YES' if row['Semantic_Class'] == row['Token_Class'] else 'NO'
df['Classes_Agree'] = df.apply(check_agreement, axis=1)

df.to_csv(f"{output_folder}/final_metrics.csv", index=False)
```

Code snippet for agreement check

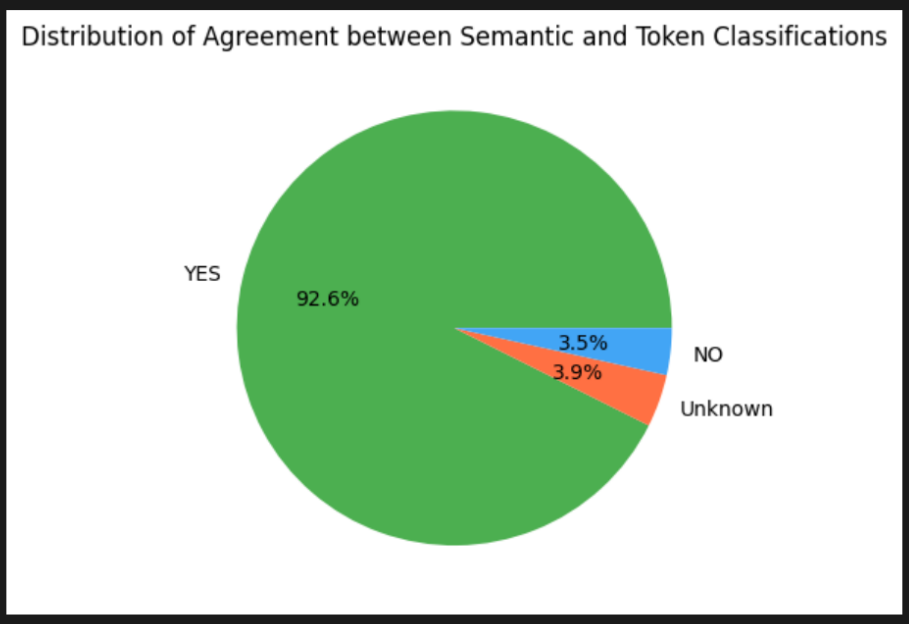
```
df = pd.read_csv(f"{output_folder}/final_metrics.csv")
display(df[['Message', 'File Name',
            'Semantic_Class', 'Token_Class', 'Classes_Agree']
           ].head(20))
```

	Message	File Name	Semantic_Class	Token_Class	Classes_Agree
0	A few fixes to initial point_thickness impleme...	constants.py	Minor	Minor	YES
1	A few fixes to initial point_thickness impleme...	displayer.py	Minor	Minor	YES
2	A few fixes to initial point_thickness impleme...	mobject.py	Minor	Minor	YES
3	middle of massive restructure, everything stil...	__init__.py	Minor	Major	NO
4	middle of massive restructure, everything stil...	__init__.py	Minor	Major	NO
5	middle of massive restructure, everything stil...	animation.py	Minor	Minor	YES
6	middle of massive restructure, everything stil...	meta_animations.py	Unknown	Unknown	Unknown
7	middle of massive restructure, everything stil...	simple_animations.py	Minor	Major	NO
8	middle of massive restructure, everything stil...	transform.py	Minor	Major	NO
9	middle of massive restructure, everything stil...	displayer.py	Minor	Minor	YES
10	middle of massive restructure, everything stil...	extract_scene.py	Minor	Minor	YES
11	middle of massive restructure, everything stil...	helpers.py	Minor	Minor	YES
12	middle of massive restructure, everything stil...	image_mobject.py	Minor	Minor	YES
13	middle of massive restructure, everything stil...	images2gif.py	Unknown	Unknown	Unknown
14	middle of massive restructure, everything stil...	mobject.py	Minor	Minor	YES
15	middle of massive restructure, everything stil...	__init__.py	Unknown	Unknown	Unknown
16	middle of massive restructure, everything stil...	complex_multiplication_article.py	Minor	Major	NO
17	middle of massive restructure, everything stil...	generate_logo.py	Minor	Minor	YES
18	middle of massive restructure, everything stil...	moser_main.py	Minor	Minor	YES
19	middle of massive restructure, everything stil...	region.py	Minor	Minor	YES

Table showing Agreement column

I exported the final table to `results/final_metrics.csv` and plotted pie chart for the distribution of agreement column.

```
Distribution of agreement between Semantic and Token classifications:
Classes_Agree
YES        1890
Unknown      80
NO           71
Name: count, dtype: int64
```



Class Distribution of Agreement

3.3 Results and Analysis

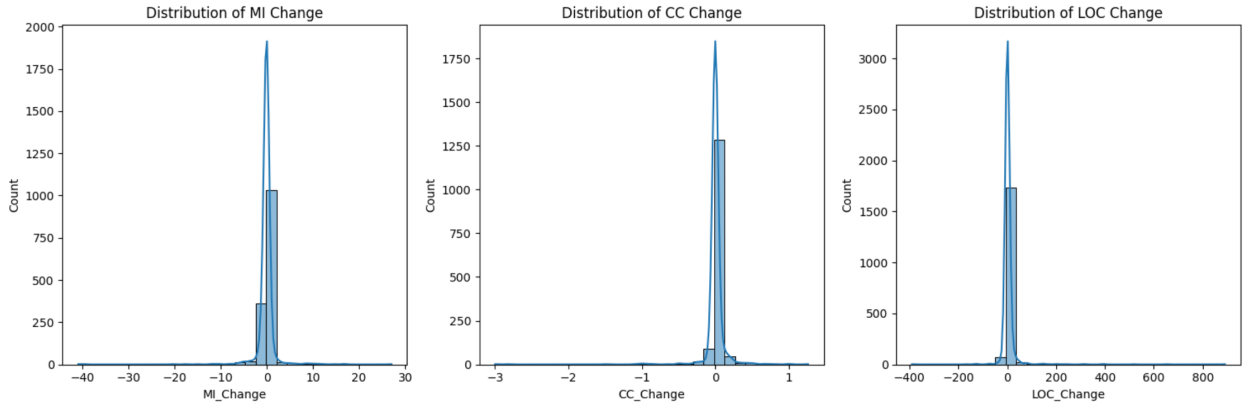
3.3.1 Final Results

Final table link: `final_metrics.csv`

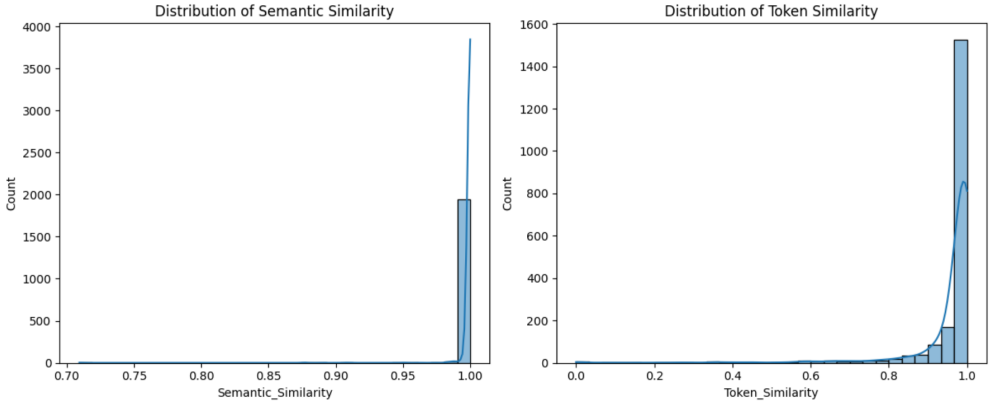
Summary of key metrics like structural changes, semantic similarity, and classification agreement:

Metric	Value
Mean MI_Change	-0.13
Mean CC_Change	0.02
Mean LOC_Change	4.5
Mean Semantic Similarity	0.9992
Mean Token Similarity	0.9596
Semantic Classification (Minor)	96.0%
Semantic Classification (Major)	0.1%
Semantic Classification (Unknown)	3.9%
Token Classification (Minor)	92.6%
Token Classification (Major)	3.5%
Token Classification (Unknown)	3.9%
Agreement (YES)	92.6%
Agreement (NO)	3.5%

3.3.2 Visualizations



Bar plots for Distribution of structural metrics



Bar plots for Distribution of Semantic and Token Similarity

3.4 Discussion and Conclusion

During this lab, I encountered a few challenges that slowed me down at first. One issue was that Radon sometimes failed when analyzing code written in older versions of Python, which meant I had to either skip those snippets or handle errors gracefully. Another challenge was that Radon itself was a completely new library for me, so I had to spend time going through its documentation and experimenting before I could use it confidently. I also ran into problems with the NLTK tokenizer setup – the lab notebook would throw runtime errors until I figured out that the punkt package needed to be downloaded separately.

This lab helped me learn a lot. I now have a much better understanding of **structural metrics** like Maintainability Index (MI), Cyclomatic Complexity (CC), and Lines of Code (LOC), and how they can reflect code quality changes. On the other hand, exploring **semantic similarity with CodeBERT** and **token similarity with BLEU** showed me how different perspectives can highlight different aspects of the same bug fix.

Overall, this lab felt like a natural extension of Lab 2. It pushed me to look beyond raw diffs and I learnt how we can classify a change/bugfix as "major" or "minor" by combining structural and semantic metrics.

3.5 References

[1] Radon documentation

[2] CodeBERT model (Hugging Face)

[3] NLTK tokenizer

[4] Lab Document (Google Doc)

Lab 4: Exploration of Different Diff Algorithms on Open-Source Repositories

Repository Link: cs202-stt/lab4

4.1 Introduction, Setup, and Tools

4.1.1 Introduction

In this lab, I compared two diff algorithms – **Myers** and **Histogram** – to see how they behave on real-world open-source repositories. The main task was to extract per-file diffs for each modified file in the commit history, identify where the two algorithms produced different results, and analyze whether these mismatches were more common in code files, test files, or documentation.

Myers Algorithm: The Myers algorithm is the **default in Git** and is based on finding the shortest edit script between two versions of a file. It is efficient and works well in general, but sometimes produces diffs that are harder to read, especially when code blocks are moved or reordered.

Histogram Algorithm: The Histogram algorithm, on the other hand, tries to anchor diffs on *rare lines first* (lines that appear less frequently), which often makes the changes more meaningful and easier to follow.

The motivation for this lab was to understand whether the choice of diff algorithm makes a practical difference for developers, reviewers, and automated tools.

4.1.2 Environment and Tools

- **Operating System:** Windows 11
- **Terminal:** PowerShell 7
- **Code Editor:** Visual Studio Code - Insiders
- **Python version:** 3.13.7
- **PyDriller version:** 2.8
- **SEART GitHub:** `https://seart-ghs.si.usi.ch/`
- **Notebooks:**
  - `lab4/diff_extract.ipynb` (data extraction)
  - `lab4/diff_analyse.ipynb` (analysis and plotting)

```
Python version: 3.13.7 (tags/v3.13.7:bcee1c3, Aug 14 2025, 14:15:11) [MSC v.1944 64 bit (AMD64)]
pydriller: 2.8
pandas: 2.3.2
matplotlib: 3.10.6
tqdm: 4.67.1
```

Environment Details

4.2 Methodology and Execution

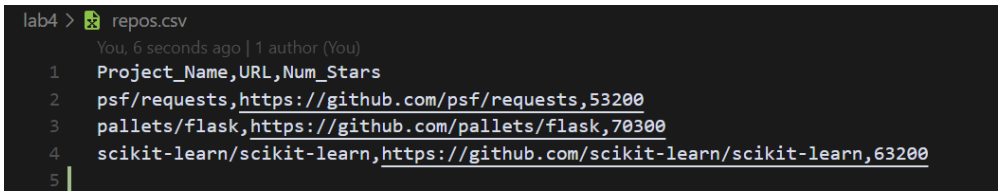
4.2.1 Repository Selection and Criteria

My selection was based on:

- **Activity:** Projects with regular commits and long histories.
- **Popularity:** Well-recognized repositories with high stars and forks.
- **Variety of files:** Containing source code, tests, and documentation.
- **Language consistency:** All three projects are Python-based, making analysis simpler.

Normally, I would start with a larger set of repositories using tools like the SEART GitHub Search Engine and then narrow down based on these criteria. The chosen projects already satisfied these requirements.

I selected three well-known Python projects to get varied histories and file types:



Repository Selection

1. psf/requests – requests is a widely used Python library for making HTTP requests. It is popular because of its simple API, reliability, and extensive use in both academic and industry projects. With over 30,000 commits, 10,000 forks, and 50,000 stars on GitHub, it is considered a highly influential and well-maintained project. Its rich commit history makes it suitable for analyzing diff algorithm discrepancies. The project was selected directly from GitHub due to its popularity and long development history.

GitHub metrics for psf/requests

2. pallets/flask – Flask is a lightweight Python web framework known for its simplicity and flexibility. It is widely adopted in both small and large-scale web applications. Flask has more than 15,000 commits, 6,000 forks, and 65,000 stars on GitHub, reflecting its popularity and active maintenance. The repository contains extensive test suites and documentation, which makes it an ideal candidate for this lab. Its selection was done using GitHub search and project metrics.

GitHub metrics for pallets/flask

3. scikit-learn/scikit-learn – scikit-learn is one of the most important machine learning libraries in Python. It is heavily used in both research and industry for tasks such as classification, regression, clustering, and model evaluation. The repository has over 27,000 commits, 25,000 forks, and 60,000 stars on GitHub, which highlights its active community and long development history. Due to its diverse codebase and well-documented commit history, it was selected as a representative project for this study.

GitHub metrics for scikit-learn/scikit-learn

These repositories represent some of the most widely adopted projects in the Python ecosystem. Each of them has tens of thousands of stars on GitHub (Requests: 53k, Flask: 71k, Scikit-learn: 63k), thousands of commits (ranging from 5k to over 32k), and a large contributor base of around 400 developers each. Their popularity, long commit history, active development, and extensive documentation make them excellent candidates for studying diff algorithm behavior for the purpose of this lab.

4.2.2 Diff Extraction Pipeline and Discrepancy Handling

Notebook link: lab4/diff_extract.ipynb

CSV Files: lab4/results

I cloned each repository under `lab4/repos/` using the `git clone` command.

```
REPO_URLS = [
    "https://github.com/psf/requests.git",
    "https://github.com/pallets/flask.git",
    "https://github.com/scikit-learn/scikit-learn.git",
]

# clone the repositories if not already cloned
for url in REPO_URLS:
    repo_name = url.split("/")[-1].replace(".git", "")
    repo_path = os.path.join(REPO_FOLDER, repo_name)
    if not os.path.exists(repo_path):
        print(f"🔄 Cloning {url}...")
        subprocess.run(["git", "clone", url, repo_path])
    else:
        print(f"📦 Repository {repo_name} already exists. Skipping clone.")
```

Git Clone

Then, I traversed each commit and:

- Extracted modified files (excluding newly added/deleted files).
- Computed two diffs per file:
  - Myers (`git diff`)
  - Histogram (`git diff --histogram`)
- Ignored whitespace and blank line changes using flags `-w` and `--ignore-blank-lines`. I also handled edge cases like if a commit is a root commit, i.e., has no parent.

For each file-modifying commit I stored the following data:

- commit_sha, parent_commit_sha
- old_file_path, new_file_path
- commit_message
- diff_myers (plain text)
- diff_hist (plain text)
- Discrepancy - A discrepancy was marked as **"Yes"** if the Myers and Histogram diffs were different, otherwise **"No"**.

```python
def extract_diffs_from_repo(repo_url):
    repo_name = repo_url.split('/')[-1].replace('.git', '')
    repo_path = os.path.join(REPO_FOLDER, repo_name)

    # Use the Git object for direct command execution
    git_repo = Git(repo_path)
    rows = []

    commits_list = list(Repository(repo_path).traverse_commits())
    for commit in tqdm(commits_list, desc=f"Traversing commits in {repo_name}"):
        if not commit.parents:
            continue
        parent_commit_sha = commit.parents[0]

        for modified_file in commit.modified_files:
            # Process only file modifications, not new files or deletions
            if modified_file.old_path and modified_file.new_path:
                # Define the common flags required by the assignment
                # -w ignores whitespace
                # --ignore-blank-lines ignores differences in blank lines
                common_flags = [
                    '-w',
                    '--ignore-blank-lines',
                    parent_commit_sha,
                    commit.hash,
                    '--',
                    modified_file.new_path
                ]

                # 1. Myers Diff (standard git diff)
                diff_myers_output = git_repo.repo.git.diff(*common_flags)
                # 2. Histogram Diff
                diff_hist_output = git_repo.repo.git.diff('--histogram', *common_flags)
                # Compare Diff Outputs
                discrepancy = "Yes" if diff_myers_output != diff_hist_output else "No"

                rows.append({
                    "old_file_path": modified_file.old_path,
                    "new_file_path": modified_file.new_path,
                    "commit_sha": commit.hash,
                    "parent_commit_sha": parent_commit_sha,
                    "commit_message": commit.msg.strip(),
                    "diff_myers": diff_myers_output,
                    "diff_hist": diff_hist_output,
                    "Discrepancy": discrepancy
                })
    return rows
```

Code Snippet for Diff Extraction

```
--- Repository 1/3 : requests ---
Traversing commits in requests: 100%|██████████| 6373/6373 [33:48<00:00,  3.14it/s]
📊 Collected 7019 records from this repository
💾 Data from requests written to results/requests_diff_analysis.csv

--- Repository 2/3 : flask ---
Traversing commits in flask: 100%|██████████| 5463/5463 [33:14<00:00,  2.74it/s]
📊 Collected 8380 records from this repository
💾 Data from flask written to results/flask_diff_analysis.csv

--- Repository 3/3 : scikit-learn ---
Traversing commits in scikit-learn: 100%|██████████| 32801/32801 [7:16:53<00:00,  1.25it/s]
📊 Collected 80107 records from this repository
💾 Data from scikit-learn written to results/scikit-learn_diff_analysis.csv

🎉 Analysis complete. Data saved to results\diff_analysis.csv.
```

Code Output for Diff Extraction

```
# Load dataset
df = pd.read_csv("results/diff_analysis.csv", encoding='utf-8', encoding_errors='surrogateescape')
df["new_file_path"] = df["new_file_path"].astype(str)
print(f"Total records in dataset: {len(df)}")
```

```
Total records in dataset: 95506
```

Number of rows extracted

This gave me a dataset with 95,506 entries. Then, I saved the extracted data into `lab4/results/diff_analysis.csv`. Since scikit-learn has a huge number of commits, the resulting CSV file exceeded 1GB in size, and could not be pushed to GitHub, so I uploaded the results folder to OneDrive instead.

```
df = pd.read_csv("results/diff_analysis.csv", encoding='utf-8', encoding_errors='surrogateescape')
display(df)
```

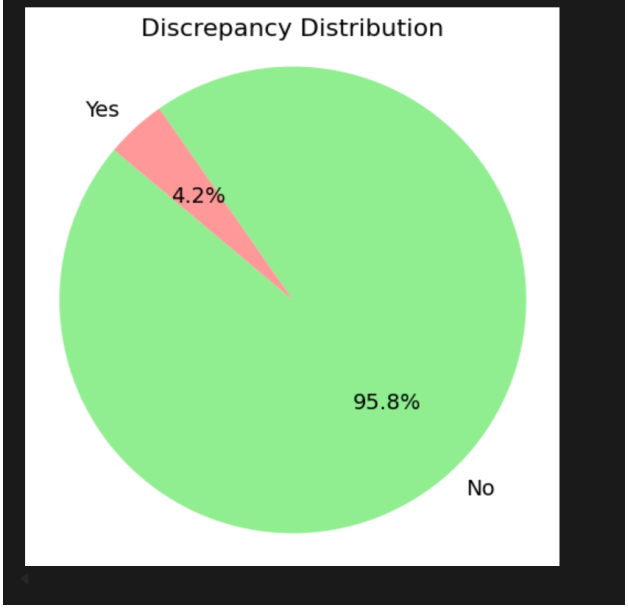
Python

	old_file_path	new_file_path	commit_sha	parent_commit_sha	commit_message	diff_myers	diff_hist	Discrepancy
0	.gitignore	.gitignore	4ec7d2a0d8eac4f915dc0d38a886cd57045bb0c4	b15ad394279fc3b7f998fa56857f334a7c0156f6	Started working on documentation.\n\nSo far ju...	diff --git a/.gitignore b/.gitignore\nindex 21...	diff --git a/.gitignore b/.gitignore\nindex 21...	No
1	examples\minitwit\minitwit.py	examples\minitwit\minitwit.py	4ec7d2a0d8eac4f915dc0d38a886cd57045bb0c4	b15ad394279fc3b7f998fa56857f334a7c0156f6	Started working on documentation.\n\nSo far ju...	diff --git a/examples/minitwit/minitwit.py b/e...	diff --git a/examples/minitwit/minitwit.py b/e...	No
2	flask.py	flask.py	4ec7d2a0d8eac4f915dc0d38a886cd57045bb0c4	b15ad394279fc3b7f998fa56857f334a7c0156f6	Started working on documentation.\n\nSo far ju...	diff --git a/flask.py b/flask.py\nindex 83d8a8...	diff --git a/flask.py b/flask.py\nindex 83d8a8...	No
3	docs\api.rst	docs\api.rst	3b36bef2e6165bb4dad73d17f23ee1879e99f497	44b42e0fbd93d86e0f4e929bda8e5fb63e81035d	Improved documentation, added a contextmanager...	diff --git a/docs/api.rst b/docs/api.rst\ninde...	diff --git a/docs/api.rst b/docs/api.rst\ninde...	No
4	docs\conf.py	docs\conf.py	3b36bef2e6165bb4dad73d17f23ee1879e99f497	44b42e0fbd93d86e0f4e929bda8e5fb63e81035d	Improved documentation, added a contextmanager...	diff --git a/docs/conf.py b/docs/conf.py\ninde...	diff --git a/docs/conf.py b/docs/conf.py\ninde...	No
...	...	...	...	...	...	...	...	...
8375	uv.lock	uv.lock	55c62556571ee46a94da174643b50ece06edead4	d8259eb11900285af9b80b0fa47f841174c054e3	update dev dependencies	diff --git a/uv.lock b/uv.lock\nindex 206c2d53...	diff --git a/uv.lock b/uv.lock\nindex 206c2d53...	Yes
8376	.github\workflows\publish.yaml	.github\workflows\publish.yaml	4dd52ca9c768c9b6d04180f0547d6f4b6e34f211	55c62556571ee46a94da174643b50ece06edead4	Update GitHub Actions workflow for artifact ha...	diff --git a/.github/workflows/publish.yaml b/...	diff --git a/.github/workflows/publish.yaml b/...	No
8377	CHANGES.rst	CHANGES.rst	2c1b30d0503cfb064f1cb252e6614a06915a362a	1292419ddfc6a14fc7f85b5ed7efcc2d215f1ad3	release version 3.1.2	diff --git a/CHANGES.rst b/CHANGES.rst\nindex ...	diff --git a/CHANGES.rst b/CHANGES.rst\nindex ...	No
8378	pyproject.toml	pyproject.toml	2c1b30d0503cfb064f1cb252e6614a06915a362a	1292419ddfc6a14fc7f85b5ed7efcc2d215f1ad3	release version 3.1.2	diff --git a/pyproject.toml b/pyproject.toml\n...	diff --git a/pyproject.toml b/pyproject.toml\n...	No
8379	uv.lock	uv.lock	2c1b30d0503cfb064f1cb252e6614a06915a362a	1292419ddfc6a14fc7f85b5ed7efcc2d215f1ad3	release version 3.1.2	diff --git a/uv.lock b/uv.lock\nindex 3f8ba3e8...	diff --git a/uv.lock b/uv.lock\nindex 3f8ba3e8...	No

Dataset Preview

Out of the 95,506 entries, around 4,000 (4.2%) had discrepancies between Myers and Histogram diffs, which can be seen in the following image.

```
Discrepancy Counts:
Discrepancy
No      91508
Yes      3998
Name: count, dtype: int64
```



Discrepancy Distribution

4.2.3 File Type Categorization and Statistics

Notebook link: lab4/diff_analyse.ipynb

I categorized files into:

- **Source Code** (extensions: .py, .java, .c, .cpp, .h, .js, .ts, .rb, .go, .php)
- **Test Code** (paths containing `test`, `spec`, or `mock`)
- **README** (files named README)
- **LICENSE** (files named LICENSE or COPYING)
- **Other** (all remaining files)

```python
# Helper function to categorize files
def categorize_file(filepath):
    if filepath is None:
        return 'Other'
    if 'test' in filepath.lower() or 'spec' in filepath.lower() or 'mock' in filepath.lower():
        return 'Test Code'
    if filepath.endswith(('.py', '.java', '.c', '.cpp', '.h', '.js', '.ts', ".rb", ".go", ".php")):
        return 'Source Code'
    if 'README' in filepath.upper():
        return 'README'
    if 'LICENSE' in filepath.upper() or 'COPYING' in filepath.upper():
        return 'LICENSE'
    return 'Other'

df['FileType'] = df['new_file_path'].apply(categorize_file)

file_types = df['FileType'].unique()
print(f"Found file types: {file_types}")
```

```
Found file types: ['README' 'Other' 'Source Code' 'Test Code' 'LICENSE']
```

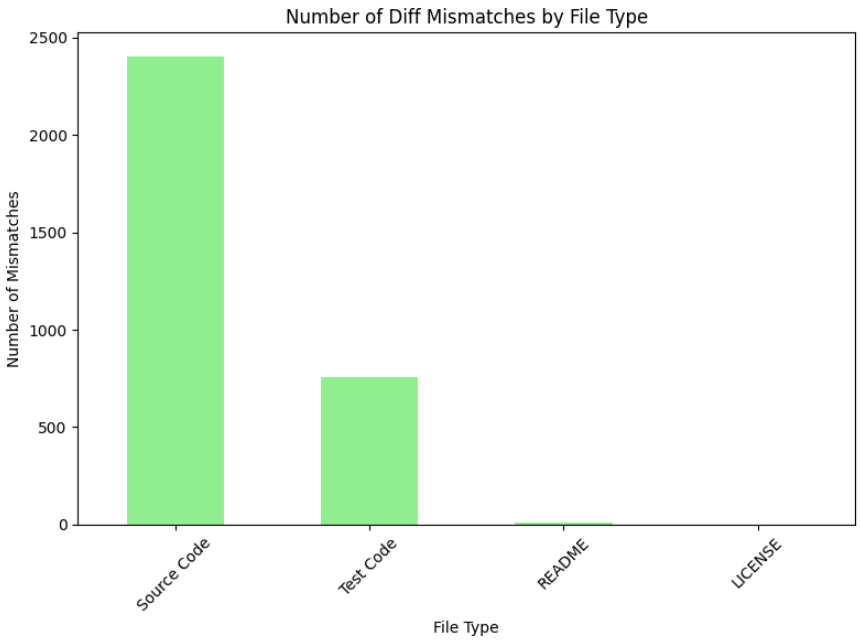
Code Snippet for File Categorization

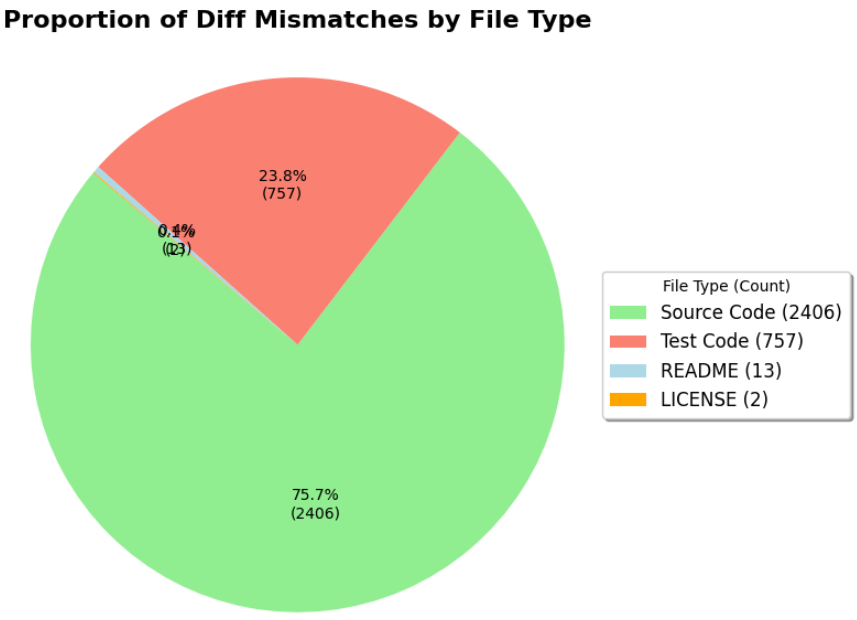
From rows where `Discrepancy == 'Yes'`, I counted mismatch counts per category.

```python
# --- Calculate statistics as required ---
mismatches_df = df[df['Discrepancy'] == 'Yes'].copy()       # Filter for rows where there is a discrepancy
print(f"Total mismatches found: {len(mismatches_df)}")

mismatch_counts = mismatches_df['FileType'].value_counts().reindex(["Source Code", "Test Code", "README", "LICENSE", "Other"]).fillna(0)

print("\nFinal Dataset Statistics:")
print(mismatch_counts)
```

```
Total mismatches found: 3998

Final Dataset Statistics:
FileType
Source Code    2406
Test Code       757
README           13
LICENSE           2
Other           820
Name: count, dtype: int64
```

Mismatch Counts

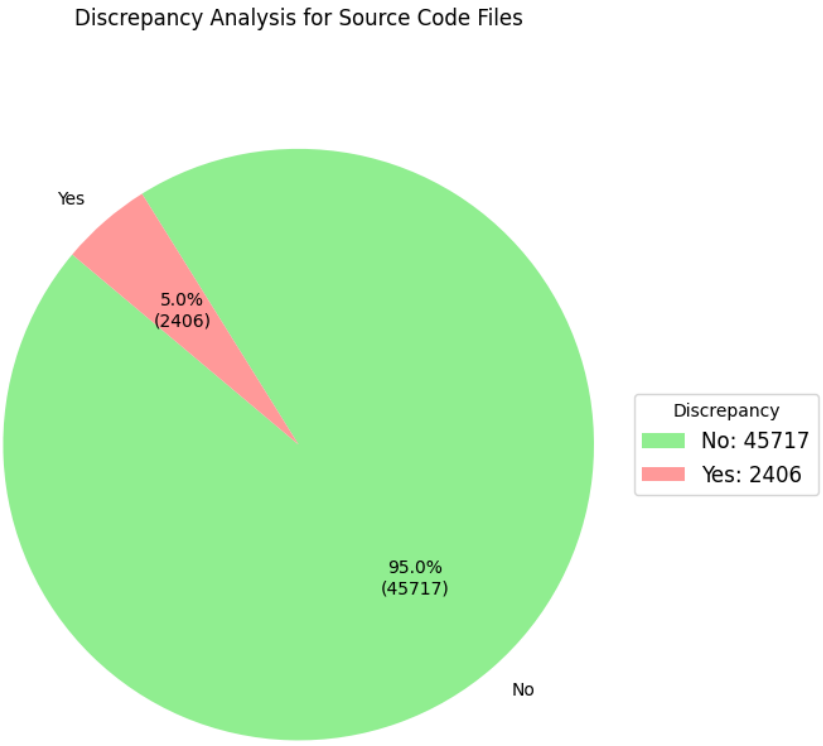




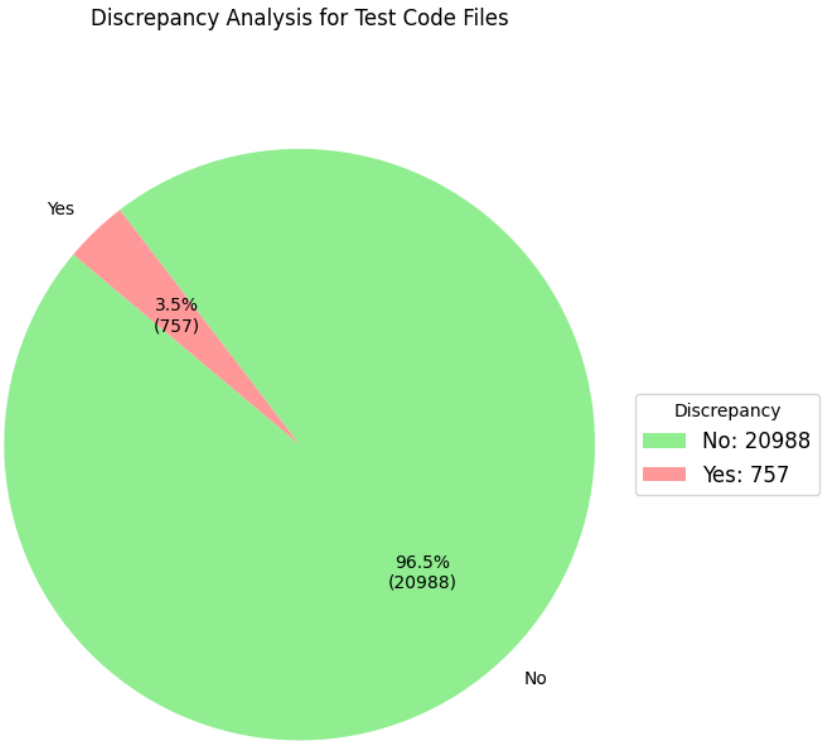
Finally, I plotted discrepancy distribution pie charts for each file type (Source Code, Test Code, README, and LICENSE) to visually compare how often the two diff algorithms disagreed across different categories.

```python
# Loop through each unique file type and create a pie chart for each
for f_type in ["Source Code", "Test Code", "README", "LICENSE"]:
    subset_df = df[df['FileType'] == f_type]
    discrepancy_counts = subset_df['Discrepancy'].value_counts()

    if not discrepancy_counts.empty:
        plt.figure(figsize=(7, 7))
        colors = ['lightgreen', '#ff9999', '#99ff99', '#ffcc99']  # Custom color palette
        wedges, texts, autotexts = plt.pie(
            discrepancy_counts,
            labels=discrepancy_counts.index,
            autopct=lambda pct: f"{pct:.1f}%\n({int(round(pct/100.*sum(discrepancy_counts)))})",  # Show percent and count
            startangle=140,
            colors=colors[:len(discrepancy_counts)],
        )

        plt.title(f'Discrepancy Analysis for {f_type} Files')
        plt.ylabel('')  # Hide y-axis label
        plt.axis('equal')  # Equal aspect ratio ensures pie is drawn as a circle
        plt.legend(wedges, [f"{label}: {count}" for label, count in discrepancy_counts.items()],
                   title="Discrepancy", loc="center left", bbox_to_anchor=(1, 0.5), fontsize=12)
        plt.tight_layout()
        plt.savefig(f"{OUTPUT_FOLDER}/pie_chart_{f_type.replace(' ', '_')}.png", bbox_inches="tight")
        plt.show()
    else:
        print(f"\nSkipping {f_type}: No discrepancy data found.")
```

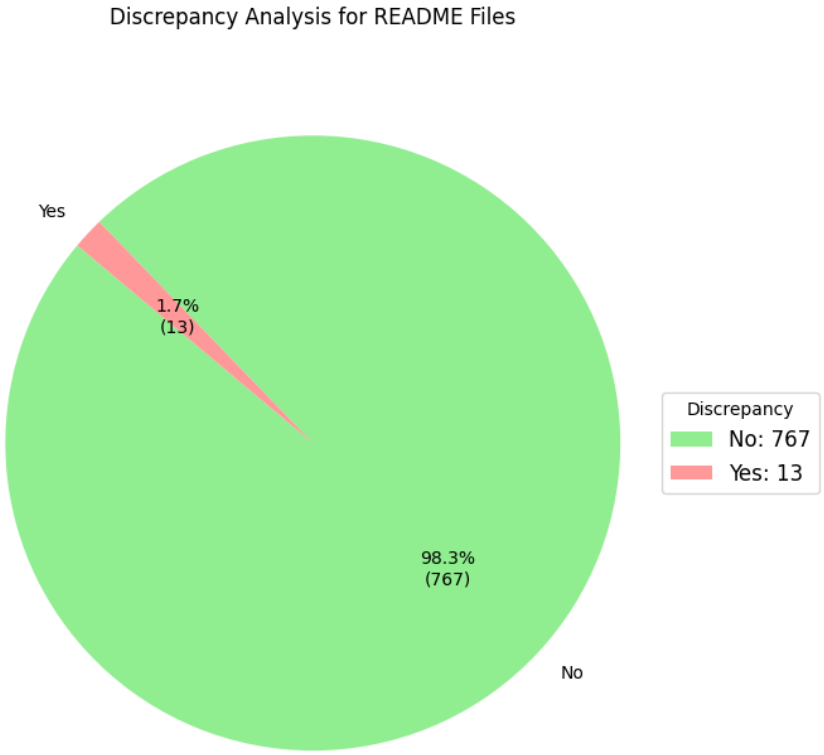
Code Snippet for Plotting File-wise Pie Charts



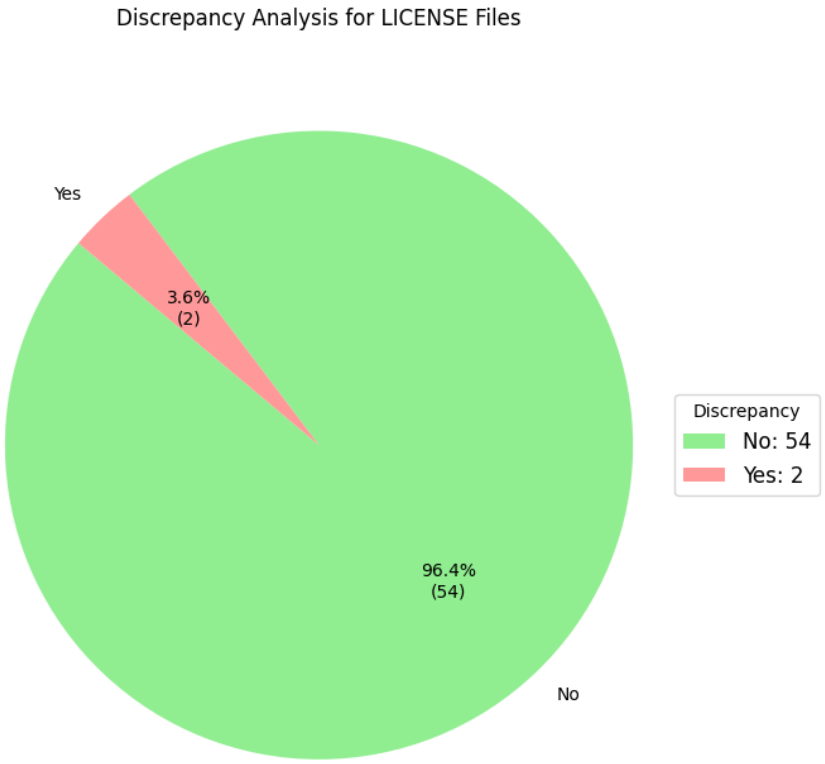
Source Code Pie Chart



Test Code Pie Chart



README Pie Chart



LICENSE Pie Chart

4.3 How to decide which performs better? (Myers vs Histogram)

If I had to automatically determine which algorithm performed better, I would not rely only on the raw mismatch count as we can't figure out which algorithm performed better with just that metric. Instead, I would try to design a small evaluation framework around the usefulness of the produced diff. The basic idea is that a "better" diff should be more consistent, easier to read, and should align with the way developers usually expect changes to be shown.

My approach would be:

1. **Define quality metrics:** For each diff, measure aspects such as number of changed lines reported, length of the diff, and whether moved blocks are detected cleanly. For example, Histogram shows block movements more clearly, while Myers sometimes fragments them into multiple small changes.
2. **Cross-check with file type:** For source code files, I would give preference to the algorithm that minimizes noise (e.g., fewer redundant changes). For text-heavy files like README or LICENSE, correctness is simpler, so the shorter diff is often better.
3. **Ground truth validation:** On a smaller subset, one could ask developers to label which diff looks clearer or more accurate. These labels could then serve as training data for an automatic scoring function.
4. **Automated scoring:** Using the above, each diff can be assigned a score (e.g., based on readability, compactness, and block preservation). Summing these scores across thousands of files would give an aggregate measure of which algorithm is generally more effective.

4.4 Results and Analysis

- Total number of modified file diffs analyzed: **95506**
- Number of mismatches found: **3998**
- **Overall discrepancy rate:** Around **4.2%** of all modified file diffs showed a mismatch between Myers and Histogram.

File Type	Mismatches
Source Code	**2406**
Test Code	**757**
README	**12**
LICENSE	**2**
Other	**820**

- **Source code files** had the most mismatches, since Histogram groups reordered/moved blocks differently than Myers.
- **Test files** also showed mismatches, often due to block movements or repeated patterns.
- **README and LICENSE files** had very few mismatches, especially after ignoring whitespace/blank lines.

4.5 Discussion and Conclusion

4.5.1 Challenges

Working with the full commit histories turned out to be quite time-consuming, especially for large projects like scikit-learn. Some commits were huge and took a long time to process, and I also ran into a few issues while saving the results into CSV files. To get around the encoding errors, I had to use surrogate escapes, which made the process more stable. On the positive side, adding the flags to ignore whitespace and blank lines really helped – it reduced a lot of unnecessary noise in the diffs and made the comparisons cleaner.

4.5.2 What I learned

Through this lab, I got a hands-on understanding of how git diffs actually work and why they matter in practice. Until now, I had only used `git diff` without thinking much about the algorithm behind it. I learned that Git mainly uses two algorithms – Myers and Histogram – and that they can produce different outputs for the same commit.

Working through real repositories helped me see these differences clearly. Myers is the default and works well in most cases, but Histogram often produces cleaner results when code blocks are moved or reordered. This gave me a much clearer picture of how diff algorithms affect the way changes are displayed, and why reviewers or tools might prefer one over the other in certain contexts.

4.6 References

[1] `https://git-scm.com/docs/git-diff`

[2] `https://github.com/ishepard/pydriller`

[3] SEART GitHub Search Engine: `https://seart-ghs.si.usi.ch/`

[4] Lab Document: Google Doc