ASSIGNMENT 1

Software Tools and Techniques for CSE

Revathi Katta
23110159

Table of Contents

LABORATORY SESSION 1: Introduction to Version Controlling, Git Workflows, and Actions

INTRODUCTION

The first lab session aimed to understand the function of version control systems (VCS), Git, and GitHub, in software development, and gain experience with them. Additionally, GitHub Actions and the Pylint workflow for automatic code quality checks were implemented in the lab. During this session, I learned the key concepts of Git, such as repository, commit, push, pull, branch, and merge, along with linking local repositories to remote ones. I also explored setting up a continuous integration workflow to ensure Python code adheres to defined style guidelines.

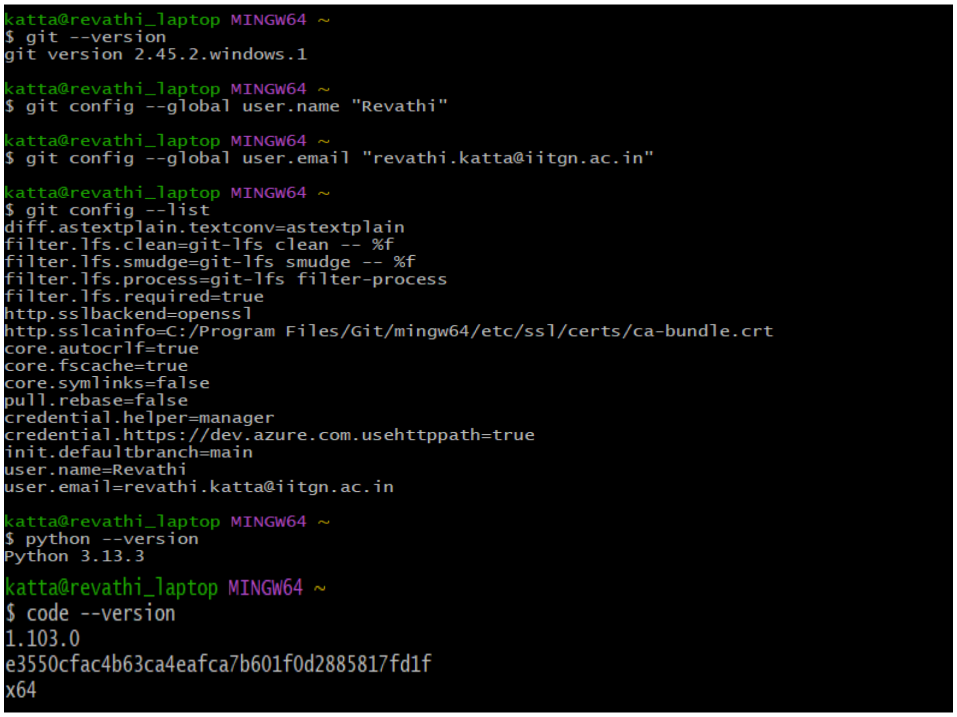
SETUP AND TOOLS

Before starting the lab tasks, I made sure that my PC had all of the necessary tools installed and configured appropriately. The required tools were checked using the commands indicated in the screenshots.

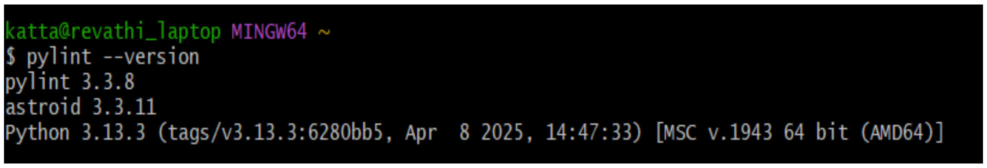
System and Tools:

- **Operating System:** Windows 11
- **Command Line Tool:** Git Bash
- **Code Editor:** Visual Studio Code (v1.103.0)
- **Version Control:** Git (v2.45.2)
- **Programming Language:** Python (v3.13.3)
- **Static Analysis Tool:** pylint (v3.2.6), for Python linting in GitHub Actions
- **Remote Hosting Platform:** GitHub

Verification Commands Executed: The screenshots below show the exact commands and their outputs from my local environment.



```
katta@revathi_laptop MINGW64 ~
$ git --version
git version 2.45.2.windows.1

katta@revathi_laptop MINGW64 ~
$ git config --global user.name "Revathi"

katta@revathi_laptop MINGW64 ~
$ git config --global user.email "revathi.katta@iitgn.ac.in"

katta@revathi_laptop MINGW64 ~
$ git config --list
diff.astextplain.textconv=astextplain
filter.lfs.clean=git-lfs clean -- %f
filter.lfs.smudge=git-lfs smudge -- %f
filter.lfs.process=git-lfs filter-process
filter.lfs.required=true
http.sslbackend=openssl
http.sslcainfo=C:/Program Files/Git/mingw64/etc/ssl/certs/ca-bundle.crt
core.autocrlf=true
core.fscache=true
core.symlinks=false
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=main
user.name=Revathi
user.email=revathi.katta@iitgn.ac.in

katta@revathi_laptop MINGW64 ~
$ python --version
Python 3.13.3

katta@revathi_laptop MINGW64 ~
$ code --version
1.103.0
e3550cfac4b63ca4eafca7b601f0d2885817fd1f
x64
```



```
katta@revathi_laptop MINGW64 ~
$ pylint --version
pylint 3.3.8
astroid 3.3.11
Python 3.13.3 (tags/v3.13.3:6280bb5, Apr  8 2025, 14:47:33) [MSC v.1943 64 bit (AMD64)]
```

METHODOLOGY AND EXECUTION

(a) Understanding Version Control: I began with why version control systems (VCS) are essential in software development. Key points included:

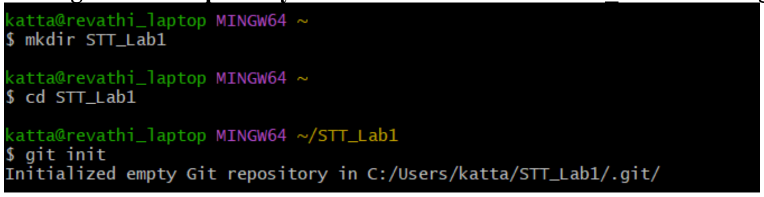
- **Change Tracking:** Records code changes to aid debugging and rollback.
- **Collaboration:** Allows multiple developers to work on the same project without overwriting each other's work.
- **Branching and Merging:** Enables trying out new features and fixes without affecting the main codebase.
- **Remote Access:** GitHub makes code available online and keeps it backed up.

Also reviewed popular VCS tools such as **Git**, **Subversion (SVN)**, and **Mercurial**, but Git came out on top because of its distributed nature and popularity. Core concepts discussed:

- **Repository:** Storage space for a project and its version history.
- **Commit:** A recorded snapshot of changes.
- **Branch:** An independent line of development.
- **Merge:** Combining changes from different branches.
- **Remote:** A version of the repository hosted online, such as on GitHub.

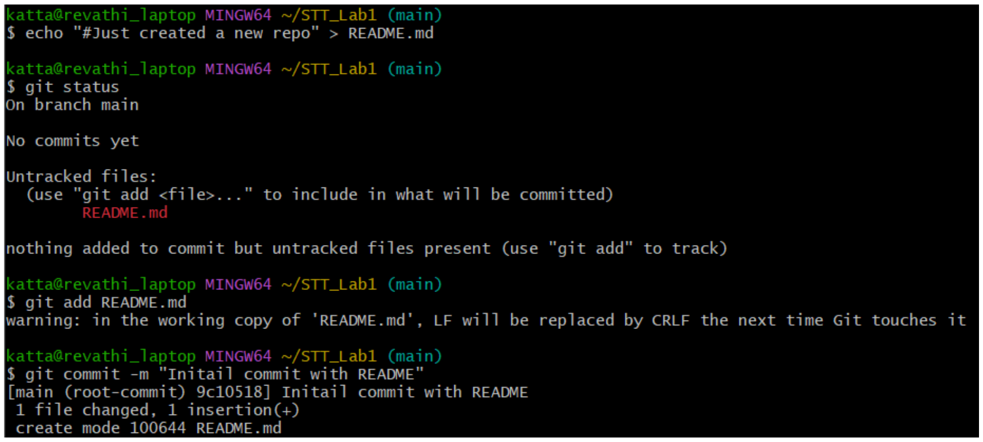
(b) Git Basics

- **Setting up Git:** I set my username and email using git config commands and verified it with git config --list. (Output is shown in the Environment Setup section.)
- **Initializing a Local Repository:** I created a folder named STT_Lab1 and ran git init inside it.



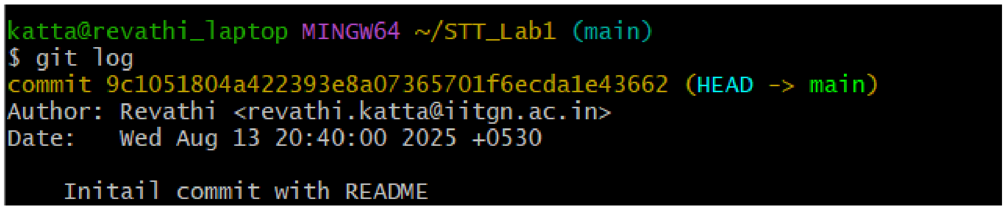
```
katta@revathi_laptop MINGW64 ~
$ mkdir STT_Lab1

katta@revathi_laptop MINGW64 ~
$ cd STT_Lab1

katta@revathi_laptop MINGW64 ~/STT_Lab1
$ git init
Initialized empty Git repository in C:/Users/katta/STT_Lab1/.git/
```

- **Adding and Committing Files:** I created a README.md file, staged it using git add README.md, and committed it with the message "Initial commit with README".



```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ echo "#Just created a new repo" > README.md

katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git status
On branch main

No commits yet

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

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

katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git add README.md
warning: in the working copy of 'README.md', LF will be replaced by CRLF the next time Git touches it

katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git commit -m "Initail commit with README"
[main (root-commit) 9c10518] Initail commit with README
 1 file changed, 1 insertion(+)
 create mode 100644 README.md
```

- **Viewing the Commit History:** Finally, I ran git log to check the commit details.

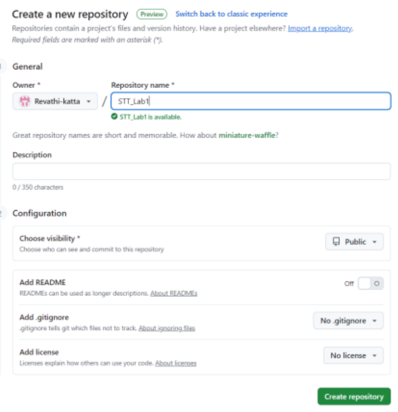


```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git log
commit 9c1051804a422393e8a07365701f6ecda1e43662 (HEAD -> main)
Author: Revathi <revathi.katta@iitgn.ac.in>
Date:   Wed Aug 13 20:40:00 2025 +0530

    Initail commit with README
```

(c) Working with Remote Repositories

- **Connecting to GitHub:**

1. Created a new empty repository on GitHub named STT_Lab1.



Create a new repository Preview Switch back to classic experience

Repositories contain a project's files and version history. Have a project elsewhere? Import a repository.

Required fields are marked with an asterisk (*).

1 General

Owner * Revathi-katta / Repository name * STT_Lab1

STT_Lab1 is available.

Great repository names are short and memorable. How about miniature-waffle?

Description

0 / 350 characters

2 Configuration

Choose visibility * — Choose who can see and commit to this repository — Public

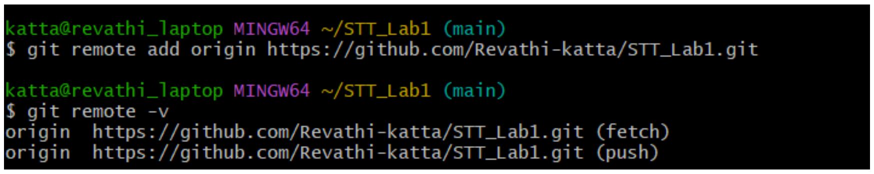
Add README — READMEs can be used as longer descriptions. About READMEs — Off

Add .gitignore — .gitignore tells git which files not to track. About ignoring files — No .gitignore

Add license — Licenses explain how others can use your code. About licenses — No license

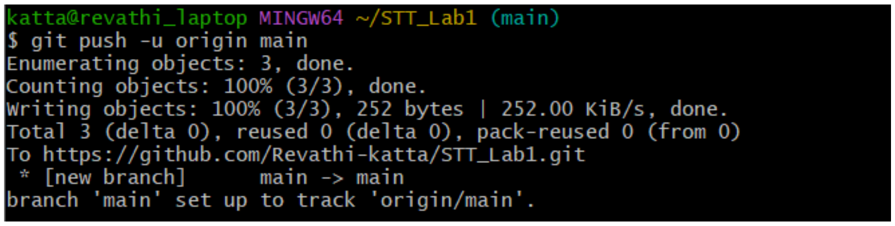
Create repository

2. Linked the local repository to the remote GitHub repository and verified the remote link:



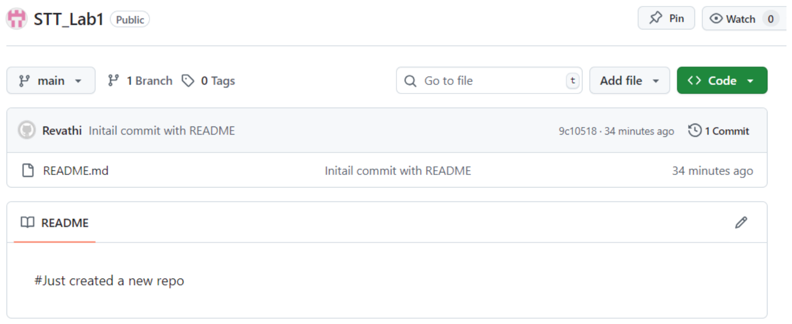
```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git remote add origin https://github.com/Revathi-katta/STT_Lab1.git

katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git remote -v
origin  https://github.com/Revathi-katta/STT_Lab1.git (fetch)
origin  https://github.com/Revathi-katta/STT_Lab1.git (push)
```

- **Pushing Changes to GitHub:** Pushed the committed changes from the local repository to the main branch of the remote repository, the -u flag set origin/main as the default upstream for future pushes.



```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), 252 bytes | 252.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/Revathi-katta/STT_Lab1.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

- We can see that the changes we made locally are pushed to the online GitHub repo:



STT_Lab1 Public

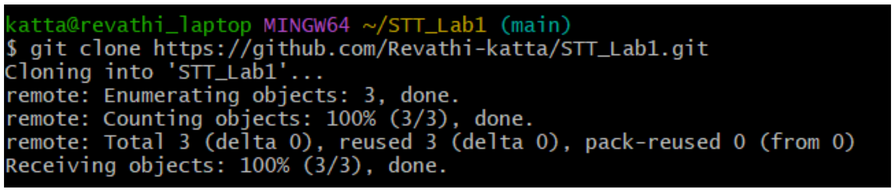
Revathi Initail commit with README 9c10518 · 34 minutes ago 1 Commit

README.md Initail commit with README 34 minutes ago

README

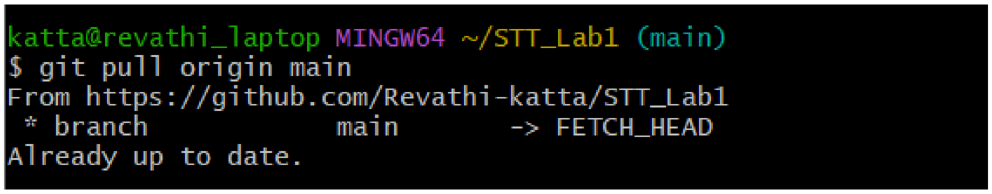
#Just created a new repo

- **Cloning a Repository:** Cloned an existing repository from GitHub to the local machine using ‘git clone’.



```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git clone https://github.com/Revathi-katta/STT_Lab1.git
Cloning into 'STT_Lab1'...
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Total 3 (delta 0), reused 3 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (3/3), done.
```

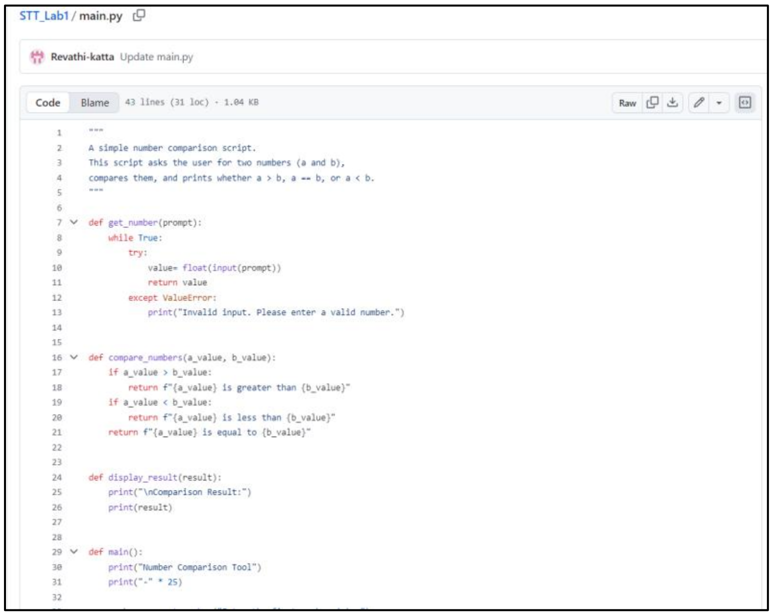
- **Pulling Changes from Remote**: Fetched and merged the latest updates from the remote repository. This ensured the local copy stayed in sync with changes made on GitHub.



```
katta@revathi_laptop MINGW64 ~/STT_Lab1 (main)
$ git pull origin main
From https://github.com/Revathi-katta/STT_Lab1
 * branch            main       -> FETCH_HEAD
Already up to date.
```

(d) Setting up a pylint workflow via GitHub Actions

- **Adding Python Code to the Repository**: I started by creating a Python script, main.py, with more than 30 lines of code. Instead of just writing random lines to hit the requirement, I made sure the code had functions, loops, and conditional logic so I could test multiple style checks from pylint. Once the script was ready, I committed it to my GitHub repository.



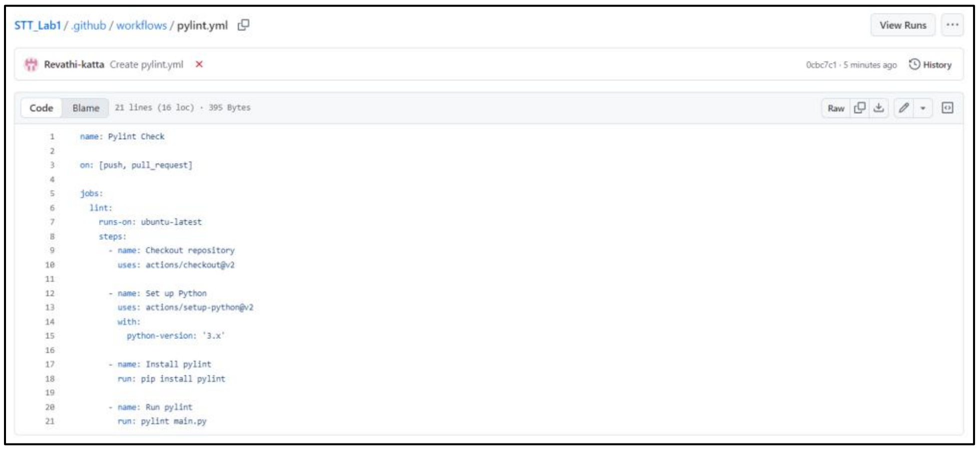
STT_Lab1 / main.py

Revathi-katta Update main.py

43 lines (31 loc) · 1.04 KB

```python
"""
A simple number comparison script.
This script asks the user for two numbers (a and b),
compares them, and prints whether a > b, a == b, or a < b.
"""


def get_number(prompt):
    while True:
        try:
            value= float(input(prompt))
            return value
        except ValueError:
            print("Invalid input. Please enter a valid number.")


def compare_numbers(a_value, b_value):
    if a_value > b_value:
        return f"{a_value} is greater than {b_value}"
    if a_value < b_value:
        return f"{a_value} is less than {b_value}"
    return f"{a_value} is equal to {b_value}"


def display_result(result):
    print("\nComparison Result:")
    print(result)


def main():
    print("Number Comparison Tool")
    print("-" * 25)

```

- **Creating the Pylint GitHub Actions Workflow:** I made a .github/workflows/ directory and added a YAML file (pylint.yml) that installed Python and ran pylint on my script.



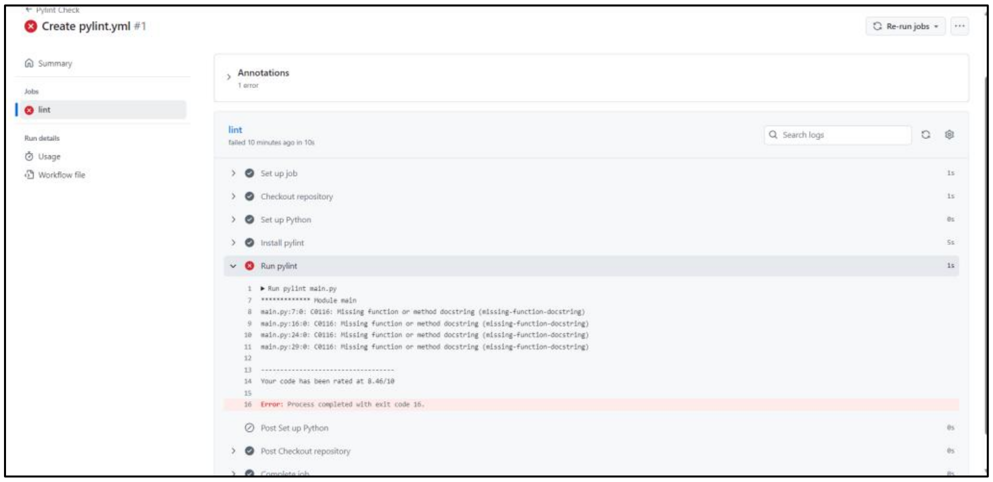
STT_Lab1 / .github / workflows / pylint.yml

Revathi-katta Create pylint.yml 0cbc7c1 · 5 minutes ago

21 lines (16 loc) · 395 Bytes

```yaml
name: Pylint Check

on: [push, pull_request]

jobs:
  lint:
    runs-on: ubuntu-latest
    steps:
      - name: Checkout repository
        uses: actions/checkout@v2

      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: '3.x'

      - name: Install pylint
        run: pip install pylint

      - name: Run pylint
        run: pylint main.py
```

- **Running and Debugging the Workflow:** After committing, when I pushed this file, GitHub Actions ran automatically. The first run failed because PyLint flagged issues like
  - Variable names not following snake_case.
  - Lines longer than the recommended limit.
  - Unused imports.
  - Missing docstrings in functions.



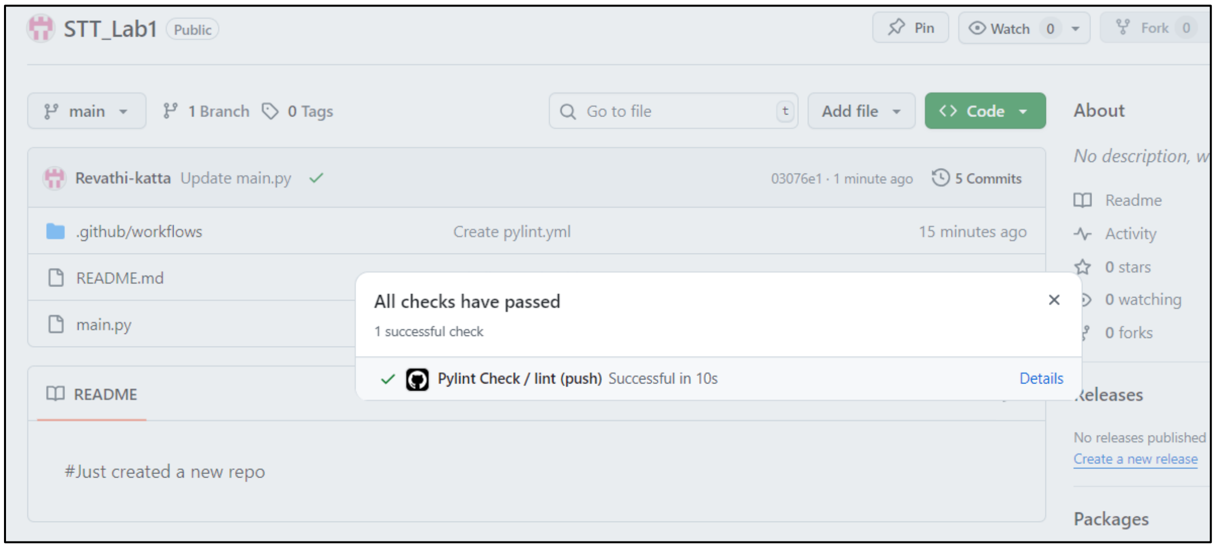
```
▶ Run pylint main.py
************* Module main
main.py:7:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:16:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:24:0: C0116: Missing function or method docstring (missing-function-docstring)
main.py:29:0: C0116: Missing function or method docstring (missing-function-docstring)

-----------------------------------
Your code has been rated at 8.46/10

Error: Process completed with exit code 16.
```

I opened main.py and fixed all the reported issues:

- Renamed variables to follow snake_case.
- Broke down long lines into shorter ones.
- Removed unused imports.
- Added docstrings to explain each function.

After making these changes, I committed the updated file.

- **Final Verification**: The GitHub Actions workflow ran again when I pushed the fixes. This time, pylint passed with a **green tick** in the Actions tab, confirming that the code met all style and linting rules. This step validated that my code was functional and followed proper Python coding standards.



Revathi-katta Update main.py ✓ 03076e1 · 1 minute ago 5 Commits

.github/workflows Create pylint.yml 15 minutes ago

README.md

main.py

All checks have passed

1 successful check

✓ Pylint Check / lint (push) Successful in 10s Details

#Just created a new repo

RESULTS AND ANALYSIS

At the end of this lab, I had:

- Configured Git and confirmed it was working.
- Created, committed, and tracked files in a local repository.
- Linked my local repo to GitHub and pushed changes.
- Practiced cloning and pulling repositories.
- Added a Python script with >30 lines.
- Set up and debugged a Pylint workflow until it passed successfully.

Key observations:

- The Git workflow became clearer only after I actually tried pushing and pulling. Reading about it was not enough.
- GitHub Actions makes linting consistent for everyone, unlike running pylint individually on different machines.
- Automation helps catch minor issues early, saving time in larger projects.

Comparison:
I found Git more flexible than SVN since I could work offline and commit locally. Also, GitHub Actions gave me a better experience than just running pylint locally, because the checks are always applied to the latest pushed code.

DISCUSSION AND CONCLUSION

Challenges:

- I was initially confused about the difference between local commits and pushing to the remote. Doing it step by step made it clear.
- My first YAML file for the workflow failed because of indentation issues, so I had to debug it carefully.
- Fixing pylint errors taught me the importance of following coding standards, even if they initially felt strict.

Reflections:
This lab was helpful because it showed the practical side of Git, not just theory. I also realized how automation can make development smoother. Even though my Python script was small, the same process could scale up for bigger projects with more contributors.

Summary:
By the end, I was comfortable with the basics of Git and GitHub. I could create repositories, track changes, and connect with remotes. On top of that, I managed to integrate a simple CI workflow using GitHub Actions and pylint. This gave me a good foundation for version control and continuous integration practices.

REFERENCES

[1] https://education.github.com/git-cheat-sheet-education.pdf

[2] CS 202 STT Lecture 1 Slides

LABORATORY SESSION 2: Commit Message Rectification for Bug-Fixing Commits in the Wild

INTRODUCTION

This lab aimed to understand how developers describe bug-fixing commits in real-world projects, and how these descriptions can sometimes be misaligned or imprecise. We explored techniques to mine GitHub repositories, identify bug-fixing commits, and rectify their commit messages using a pre-trained LLM and a rectifier mechanism.

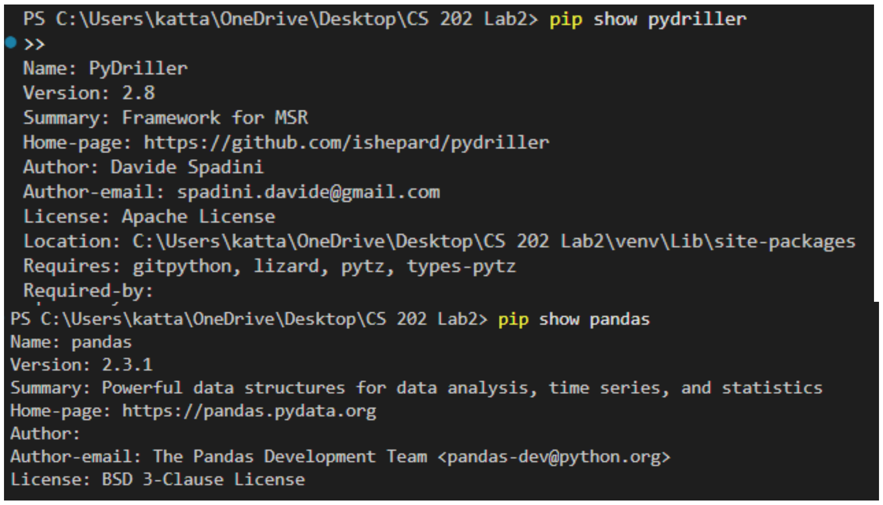
Commit messages act as the first level of documentation for a change, but developers often batch unrelated changes into one commit, making the message ambiguous. This lab helped develop a systematic framework for commit message rectification and laid the foundation for dataset preparation for downstream tasks such as program repair, vulnerability detection, and automated patch generation.

SETUP AND TOOLS

The work was carried out in **Google Colab**, with the following setup:

- **Platform**: Google Colab, Python 3.10
- **Repository Mining**: pydriller (to extract commits, metadata, diffs, and file changes)
- **Pre-trained LLM**: CommitPredictorT5 (from HuggingFace) to generate candidate commit messages
- **Dataset Management**: pandas and CSV for storing intermediate outputs
- **Environment**: SET-IITGN-VM (optional) was recommended for consistency, but I used Colab

Before starting the analysis, I verified the installation of PyDriller and the LLM model.



```
PS C:\Users\katta\OneDrive\Desktop\CS 202 Lab2> pip show pydriller
>>
Name: PyDriller
Version: 2.8
Summary: Framework for MSR
Home-page: https://github.com/ishepard/pydriller
Author: Davide Spadini
Author-email: spadini.davide@gmail.com
License: Apache License
Location: C:\Users\katta\OneDrive\Desktop\CS 202 Lab2\venv\Lib\site-packages
Requires: gitpython, lizard, pytz, types-pytz
Required-by:
PS C:\Users\katta\OneDrive\Desktop\CS 202 Lab2> pip show pandas
Name: pandas
Version: 2.3.1
Summary: Powerful data structures for data analysis, time series, and statistics
Home-page: https://pandas.pydata.org
Author:
Author-email: The Pandas Development Team <pandas-dev@python.org>
License: BSD 3-Clause License
```

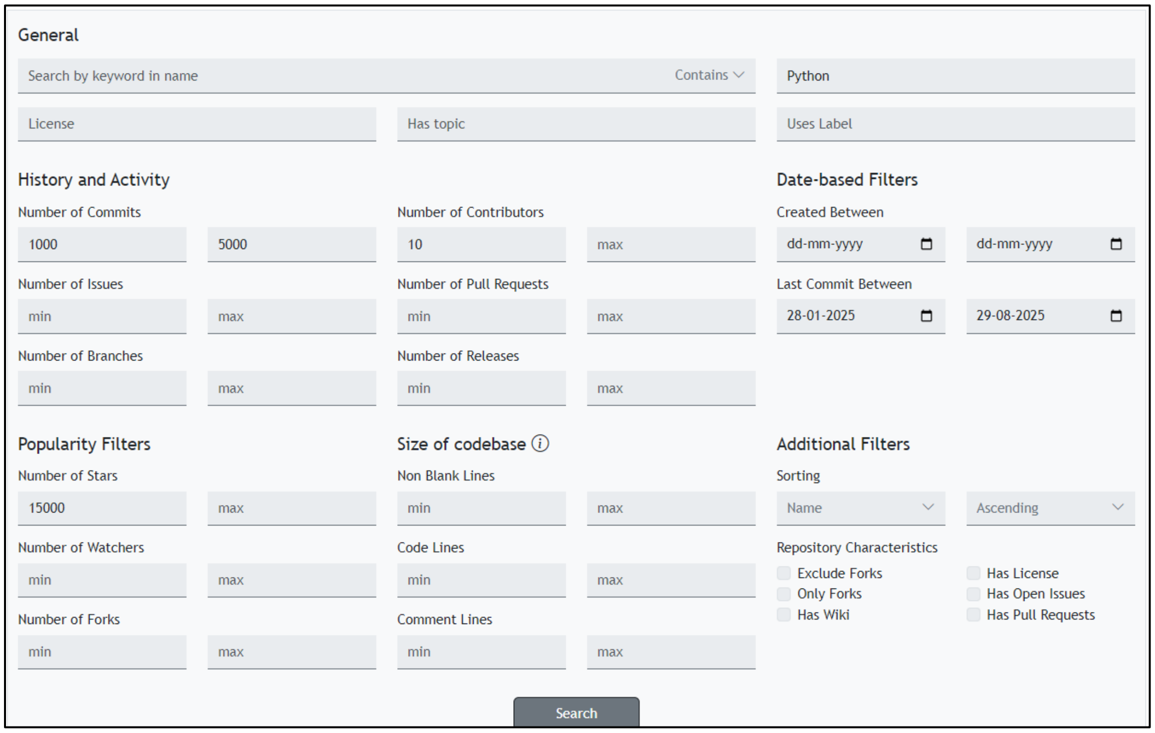
METHODOLOGY AND EXECUTION

(a) Repository Selection: I selected one medium-scale open-source project from GitHub that was identified using the **SEART GitHub Search Engine**, which provides advanced filters. This was preferable to manual browsing because it allowed systematic narrowing based on predefined inclusion criteria..

(b) Define Selection Criteria: Inspired by the **hierarchical funnel diagram** from Lecture 2, I applied a multi-stage filtering process.

- Repositories with at least 15,000 stars (popularity).
- Between 1,000–5,000 commits (sufficient history, manageable runtime).
- At least 10 contributors (to capture collaborative practices).
- Recent activity (commits from Jan–Aug 2025).
- Python as the primary language, ensuring compatibility with the lab environment.

After applying these filters, the repository agno-agi/agno was chosen. With ~3,993 commits, 257 contributors, over 32k stars, and nearly exclusive Python code, it offered scale and relevance for the study.


General

Search by keyword in name · Contains · Python

License · Has topic · Uses Label

History and Activity

Number of Commits: 1000 – 5000

Number of Contributors: 10 – max

Number of Issues: min – max

Number of Pull Requests: min – max

Number of Branches: min – max

Number of Releases: min – max

Date-based Filters

Created Between: dd-mm-yyyy – dd-mm-yyyy

Last Commit Between: 28-01-2025 – 29-08-2025

Popularity Filters

Number of Stars: 15000 – max

Number of Watchers: min – max

Number of Forks: min – max

Size of codebase

Non Blank Lines: min – max

Code Lines: min – max

Comment Lines: min – max

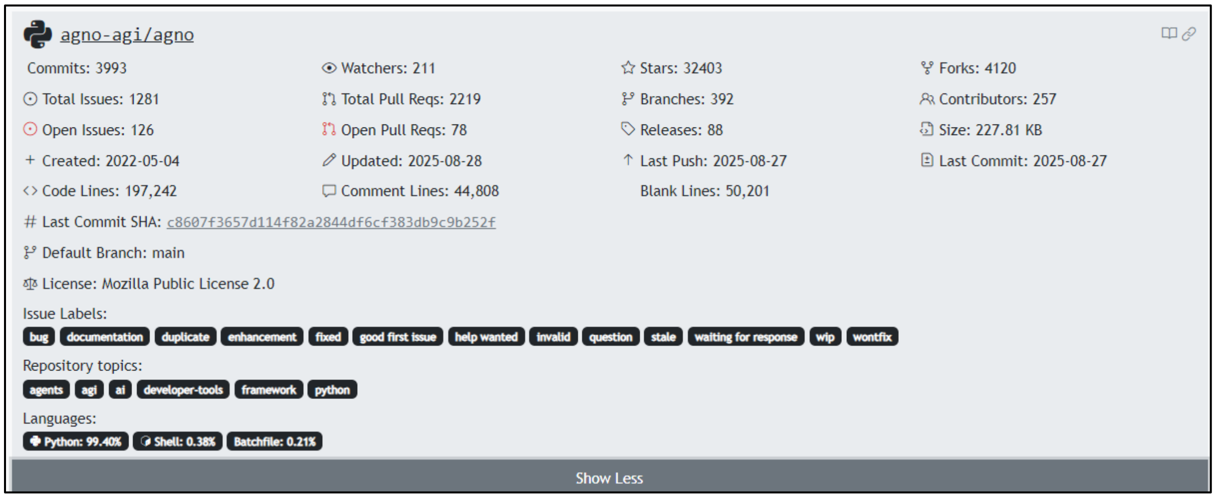
Additional Filters

Sorting: Name · Ascending

Repository Characteristics

- [ ] Exclude Forks
- [ ] Only Forks
- [ ] Has Wiki
- [ ] Has License
- [ ] Has Open Issues
- [ ] Has Pull Requests

Search


agno-agi/agno

Commits: 3993 · Watchers: 211 · Stars: 32403 · Forks: 4120

Total Issues: 1281 · Total Pull Reqs: 2219 · Branches: 392 · Contributors: 257

Open Issues: 126 · Open Pull Reqs: 78 · Releases: 88 · Size: 227.81 KB

Created: 2022-05-04 · Updated: 2025-08-28 · Last Push: 2025-08-27 · Last Commit: 2025-08-27

Code Lines: 197,242 · Comment Lines: 44,808 · Blank Lines: 50,201

Last Commit SHA: c8607f3657d114f82a2844df6cf383db9c9b252f

Default Branch: main

License: Mozilla Public License 2.0

Issue Labels:
bug · documentation · duplicate · enhancement · fixed · good first issue · help wanted · invalid · question · stale · waiting for response · wip · wontfix

Repository topics:
agents · agi · ai · developer-tools · framework · python

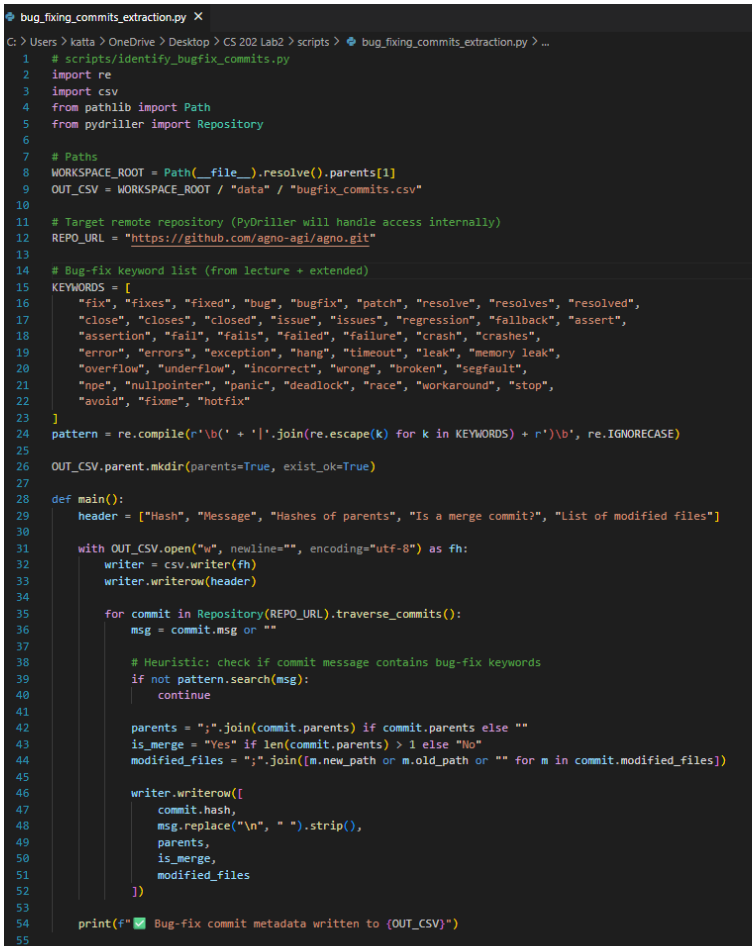
Languages:
Python: 99.40% · Shell: 0.38% · Batchfile: 0.21%

Show Less

- **Commits:** ~3,993
- **Contributors:** 257
- **Stars:** 32,403
- **Forks:** 4,120
- **Created:** May 2022
- **Last Updated:** Aug 2025
- **Languages:** ~99% Python (ideal match for this lab's requirements)
- **License:** Mozilla Public License 2.0

(c) Bug-Fixing Commit Identification

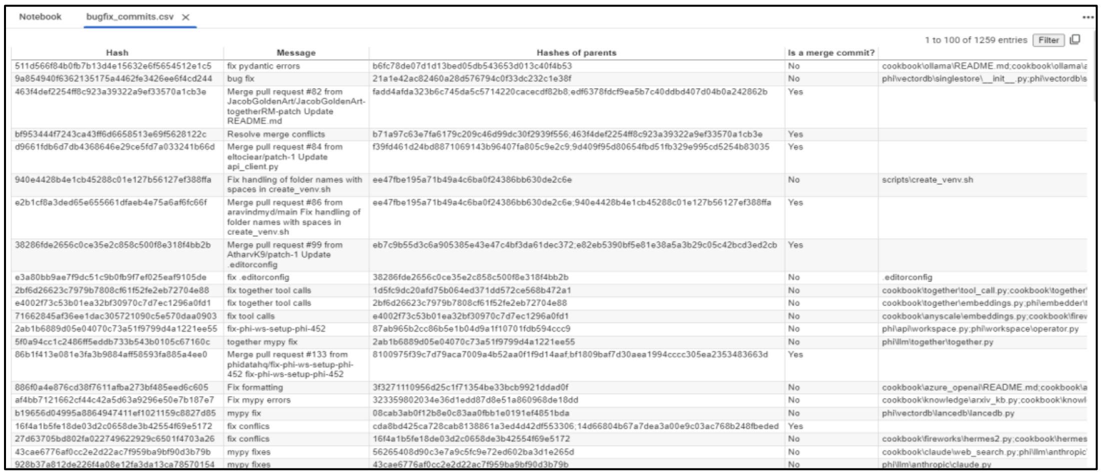
To detect bug-fix commits, I implemented a **keyword-driven search** following the strategy from Lecture 2. A curated list of bug-related tokens (e.g., "fix", "resolve", "patch", "hotfix", "segfault") was compiled into a case-insensitive regular expression. Each commit message in the repository was tested against this expression. Matching commits were recorded with metadata such as hash, parent hashes, merge status, and modified files. The output was stored in bugfix_commits.csv, yielding **1,259 bug-fixing commits**. I coded the above-described implementation as follows:


bug_fixing_commits_extraction.py
C: > Users > katta > OneDrive > Desktop > CS 202 Lab2 > scripts > bug_fixing_commits_extraction.py > ...

```python
# scripts/identify_bugfix_commits.py
import re
import csv
from pathlib import Path
from pydriller import Repository

# Paths
WORKSPACE_ROOT = Path(__file__).resolve().parents[1]
OUT_CSV = WORKSPACE_ROOT / "data" / "bugfix_commits.csv"

# Target remote repository (PyDriller will handle access internally)
REPO_URL = "https://github.com/agno-agi/agno.git"

# Bug-fix keyword list (from lecture + extended)
KEYWORDS = [
    "fix", "fixes", "fixed", "bug", "bugfix", "patch", "resolve", "resolves", "resolved",
    "close", "closes", "closed", "issue", "issues", "regression", "fallback", "assert",
    "assertion", "fail", "fails", "failed", "failure", "crash", "crashes",
    "error", "errors", "exception", "hang", "timeout", "leak", "memory leak",
    "overflow", "underflow", "incorrect", "wrong", "broken", "segfault",
    "npe", "nullpointer", "panic", "deadlock", "race", "workaround", "stop",
    "avoid", "fixme", "hotfix"
]
pattern = re.compile(r'\b(' + '|'.join(re.escape(k) for k in KEYWORDS) + r')\b', re.IGNORECASE)

OUT_CSV.parent.mkdir(parents=True, exist_ok=True)

def main():
    header = ["Hash", "Message", "Hashes of parents", "Is a merge commit?", "List of modified files"]

    with OUT_CSV.open("w", newline="", encoding="utf-8") as fh:
        writer = csv.writer(fh)
        writer.writerow(header)

        for commit in Repository(REPO_URL).traverse_commits():
            msg = commit.msg or ""

            # Heuristic: check if commit message contains bug-fix keywords
            if not pattern.search(msg):
                continue

            parents = ";".join(commit.parents) if commit.parents else ""
            is_merge = "Yes" if len(commit.parents) > 1 else "No"
            modified_files = ";".join([m.new_path or m.old_path or "" for m in commit.modified_files])

            writer.writerow([
                commit.hash,
                msg.replace("\n", " ").strip(),
                parents,
                is_merge,
                modified_files
            ])

    print(f"✅ Bug-fix commit metadata written to {OUT_CSV}")
```

Output:

The script successfully generated the bugfix_commits.csv file containing the metadata of detected bug-fixing commits.


Notebook
bugfix_commits.csv
1 to 100 of 1259 entries
Filter

Hash	Message	Hashes of parents	Is a merge commit?	
511d566f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	b6fc78de07d1d13bed05db543653d013c40f4b53	No	cookbook\ollama\README.md;cookbook\ollama\a
9a854940f6362135175a4462fe3426ee6f4cd244	bug fix	21a1e42ac82460a28d576794c0f33dc232c1e38f	No	phi\vectordb\singlestore__init__.py;phi\vectordb\s
463f4def2254ff8c923a39322a9ef33570a1cb3e	Merge pull request #82 from JacobGoldenArt/JacobGoldenArt-togetherRM-patch Update README.md	fadd4afda323b6c745da5c5714220cacecdf82b8;edf6378fdcf9ea5b7c40ddbd407d04b0a242862b	Yes	
bf953444f7243ca43ff6d6658513e69f5628122c	Resolve merge conflicts	b71a97c63e7fa6179c209c46d99dc30f2939f556;463f4def2254ff8c923a39322a9ef33570a1cb3e	Yes	
d9661fdb6d7db4368646e29ce5fd7a033241b66d	Merge pull request #84 from eltociear/patch-1 Update api_client.py	f39fd461d24bd8871069143b96407fa805c9e2c9;9d409f95d80654fbd51fb329e995cd5254b83035	Yes	
940e4428b4e1cb45288c01e127b56127ef388ffa	Fix handling of folder names with spaces in create_venv.sh	ee47fbe195a71b49a4c6ba0f24386bb630de2c6e	No	scripts\create_venv.sh
e2b1cf8a3ded65e655661dfaeb4e75a6af6fc66f	Merge pull request #86 from aravindmyd/main Fix handling of folder names with spaces in create_venv.sh	ee47fbe195a71b49a4c6ba0f24386bb630de2c6e;940e4428b4e1cb45288c01e127b56127ef388ffa	Yes	
38286fde2656c0ce35e2c858c500f8e318f4bb2b	Merge pull request #99 from AtharvK9/patch-1 Update .editorconfig	eb7c9b55d3c6a905385e43e47c4bf3da61dec372;e82eb5390bf5e81e38a5a3b29c05c42bcd3ed2cb	Yes	
e3a80bb9ae7f9dc51c9b0fb9f7ef025eaf9105de	fix .editorconfig	38286fde2656c0ce35e2c858c500f8e318f4bb2b	No	.editorconfig
2bf6d26623c7979b7808cf61f52fe2eb72704e88	fix together tool calls	1d5fc9dc20afd75b064ed371dd572ce568b472a1	No	cookbook\together\tool_call.py;cookbook\together'
e4002f73c53b01ea32bf30970c7d7ec1296a0fd1	fix together tool calls	2bf6d26623c7979b7808cf61f52fe2eb72704e88	No	cookbook\together\embeddings.py;phi\embedder\t
71662845af36ee1dac305721090c5e570daa0903	fix tool calls	e4002f73c53b01ea32bf30970c7d7ec1296a0fd1	No	cookbook\anyscale\embeddings.py;cookbook\firev
2ab1b6889d05e04070c73a51f9799d4a1221ee55	fix-phi-ws-setup-phi-452	87ab965b2cc86b5e1b04d9a1f10701fdb594ccc9	No	phi\api\workspace.py;phi\workspace\operator.py
5f0a94cc1c2486ff5eddb733b543b0105c67160c	together mypy fix	2ab1b6889d05e04070c73a51f9799d4a1221ee55	No	phi\llm\together\together.py
86b1f413e081e3fa3b9884aff58593fa885a4ee0	Merge pull request #133 from phidatahq/fix-phi-ws-setup-phi-452 fix-phi-ws-setup-phi-452	8100975f39c7d79aca7009a4b52aa0f1f9d14aaf;bf1809baf7d30aea1994cccc305ea2353483663d	Yes	
886f0a4e876cd38f7611afba273bf485eed6c605	Fix formatting	3f3271110956d25c1f71354be33bcb9921ddad0f	No	cookbook\azure_openai\README.md;cookbook\a
af4bb7121662cf44c42a5d63a9296e50e7b187e7	Fix mypy errors	323359802034e36d1edd87d8e51a860968de18dd	No	cookbook\knowledge\arxiv_kb.py;cookbook\knowl
b19656d04995a8864947411ef1021159c8827d85	mypy fix	08cab3ab0f12b8e0c83aa0fbb1e0191ef4851bda	No	phi\vectordb\lancedb\lancedb.py
16f4a1b5fe18de03d2c0658de3b42554f69e5172	fix conflics	cda8bd425ca728cab8138861a3ed4d42df553306;14d66804b67a7dea3a00e9c03ac768b248fbeded	Yes	
27d63705bd802fa022749622929c6501f4703a26	fix conflics	16f4a1b5fe18de03d2c0658de3b42554f69e5172	No	cookbook\fireworks\hermes2.py;cookbook\hermes
43cae6776af0cc2e2d22ac7f959ba9bf90d3b79b	mypy fixes	56265408d90c3e7a9c5fc9e72ed602ba3d1e265d	No	cookbook\claude\web_search.py;phi\llm\anthropic'
928b37a812de226f4a08e12fa3da13ca78570154	mypy fixes	43cae6776af0cc2e2d22ac7f959ba9bf90d3b79b	No	phi\llm\anthropic\claude.py

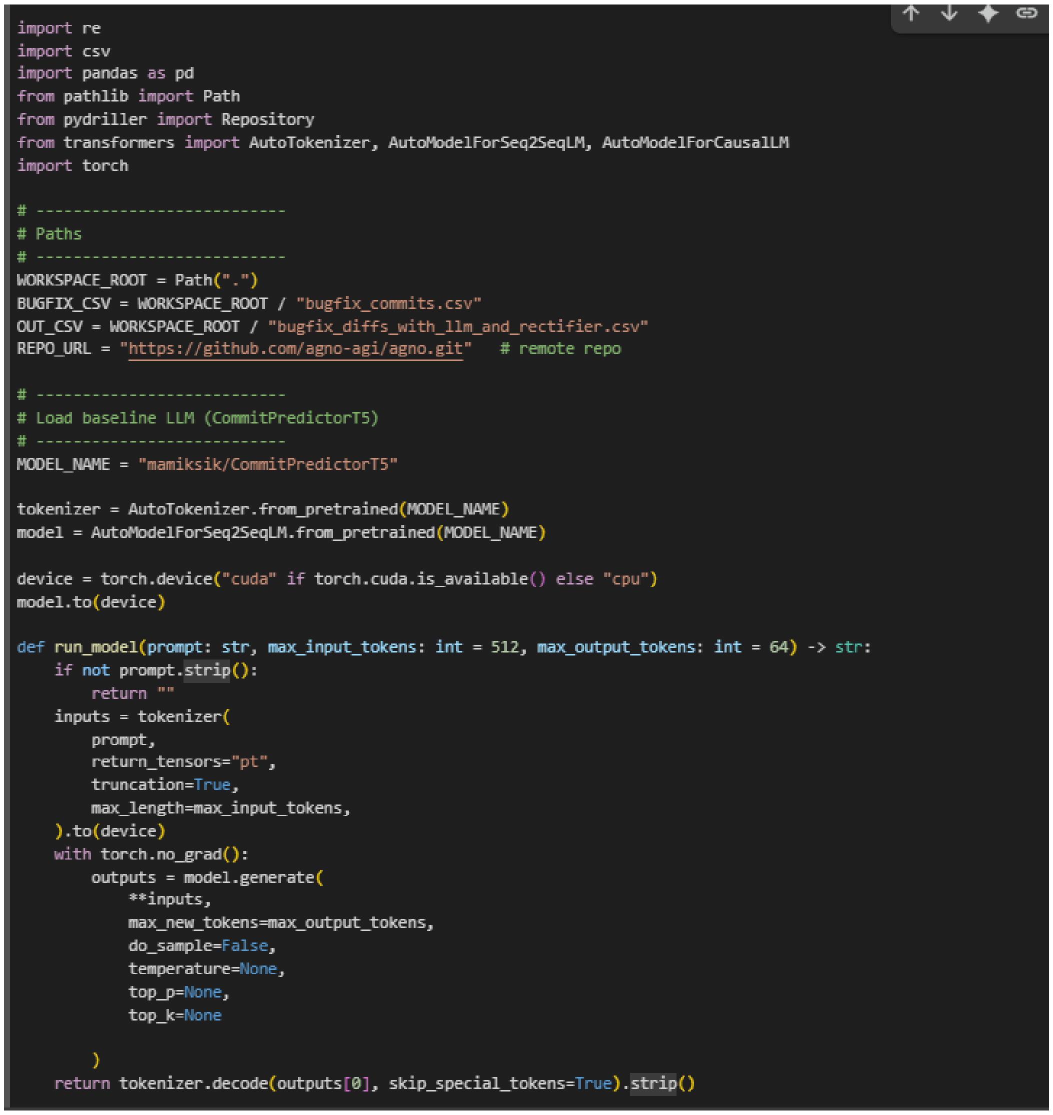
(d) Diff Extraction and Analyses

The next task was to extract **per-file code changes** for each bug-fix commit and analyze them using an LLM. This required filtering commits, preprocessing diffs, and generating candidate commit messages.

Steps Followed:

- **Target Commits:**
  From the prepared list of bug-fix commits (bugfix_commits.csv), I restricted the dataset to a subset of **1000 commits**. This was done to maintain runtime feasibility in Google Colab while ensuring enough variety. To further manage GPU resources, I processed the commits in **batches of 100** 3 times.

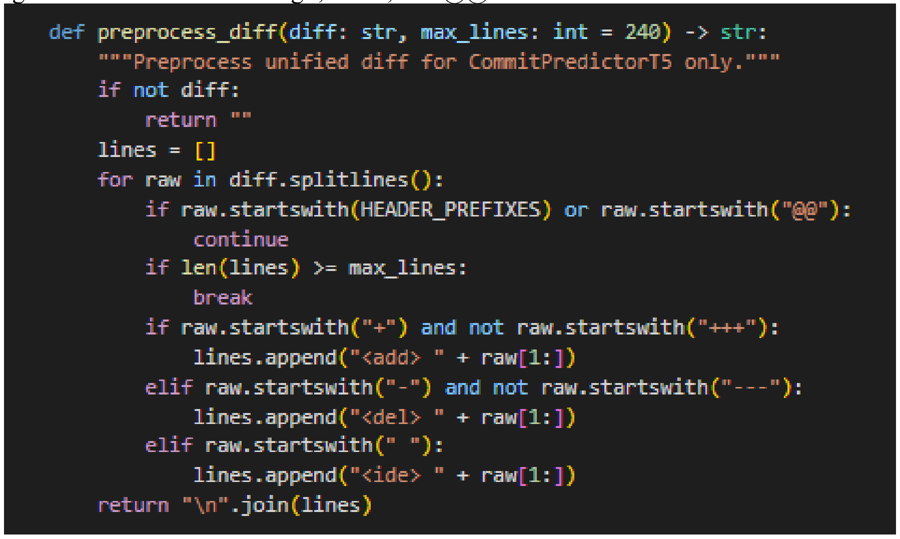
- **Baseline LLM (CommitPredictorT5)**
  I used the Hugging Face model **mamiksik/CommitPredictorT5** to generate an initial subject line for each file-level diff. The model produced a concise commit description, typically indicating the **fix type**.
    - A helper function run_model() tokenized input diffs, ran inference, and decoded the results.
    - Each output was post-processed with enforce_git_style() to conform to Git standards (imperative mood, ≤50 characters, no trailing punctuation).



```python
import re
import csv
import pandas as pd
from pathlib import Path
from pydriller import Repository
from transformers import AutoTokenizer, AutoModelForSeq2SeqLM, AutoModelForCausalLM
import torch

# --------------------------
# Paths
# --------------------------
WORKSPACE_ROOT = Path(".")
BUGFIX_CSV = WORKSPACE_ROOT / "bugfix_commits.csv"
OUT_CSV = WORKSPACE_ROOT / "bugfix_diffs_with_llm_and_rectifier.csv"
REPO_URL = "https://github.com/agno-agi/agno.git"   # remote repo

# --------------------------
# Load baseline LLM (CommitPredictorT5)
# --------------------------
MODEL_NAME = "mamiksik/CommitPredictorT5"

tokenizer = AutoTokenizer.from_pretrained(MODEL_NAME)
model = AutoModelForSeq2SeqLM.from_pretrained(MODEL_NAME)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
model.to(device)

def run_model(prompt: str, max_input_tokens: int = 512, max_output_tokens: int = 64) -> str:
    if not prompt.strip():
        return ""
    inputs = tokenizer(
        prompt,
        return_tensors="pt",
        truncation=True,
        max_length=max_input_tokens,
    ).to(device)
    with torch.no_grad():
        outputs = model.generate(
            **inputs,
            max_new_tokens=max_output_tokens,
            do_sample=False,
            temperature=None,
            top_p=None,
            top_k=None

        )
    return tokenizer.decode(outputs[0], skip_special_tokens=True).strip()
```

- **Diff Preprocessing**
  Commit diffs contain a lot of metadata (headers, line indices) that are not useful for LLMs. To highlight only semantic code changes, the LLM is also trained on this similar input format, so I implemented a **preprocessing function**:
- Added lines (+) → <add>
- Deleted lines (-) → <del>
- Context lines (unchanged) → <ide>

- Ignored metadata like diff --git, index, and @@.



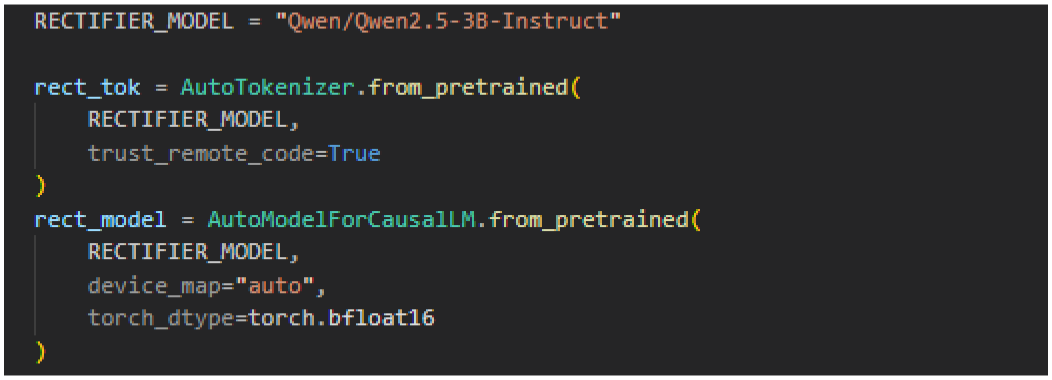
```python
def preprocess_diff(diff: str, max_lines: int = 240) -> str:
    """Preprocess unified diff for CommitPredictorT5 only."""
    if not diff:
        return ""
    lines = []
    for raw in diff.splitlines():
        if raw.startswith(HEADER_PREFIXES) or raw.startswith("@@"):
            continue
        if len(lines) >= max_lines:
            break
        if raw.startswith("+") and not raw.startswith("+++"):
            lines.append("<add> " + raw[1:])
        elif raw.startswith("-") and not raw.startswith("---"):
            lines.append("<del> " + raw[1:])
        elif raw.startswith(" "):
            lines.append("<ide> " + raw[1:])
    return "\n".join(lines)
```

(e) Rectifier Formulation

Although the baseline model produced useful summaries, the outputs were often **generic** or **misaligned** with the specific file changes. To improve precision, I implemented a **Rectifier pipeline** using a stronger LLM.

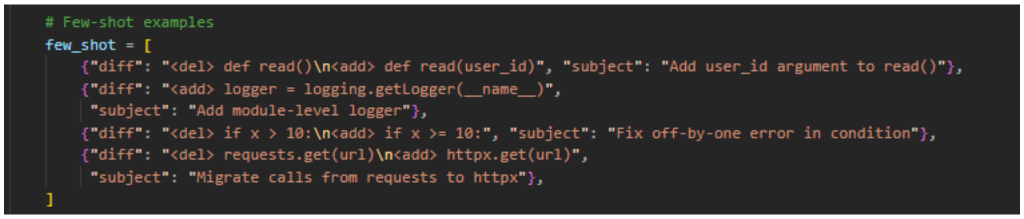
Steps Followed:

1. **Rectifier Model Selection**
   I chose **Qwen/Qwen2.5-3B-Instruct**, a larger instruction-tuned model, for rectification. It was loaded with memory-efficient settings (device_map="auto", torch_dtype=torch.bfloat16) to reduce GPU usage.



```python
RECTIFIER_MODEL = "Qwen/Qwen2.5-3B-Instruct"

rect_tok = AutoTokenizer.from_pretrained(
    RECTIFIER_MODEL,
    trust_remote_code=True
)
rect_model = AutoModelForCausalLM.from_pretrained(
    RECTIFIER_MODEL,
    device_map="auto",
    torch_dtype=torch.bfloat16
)
```

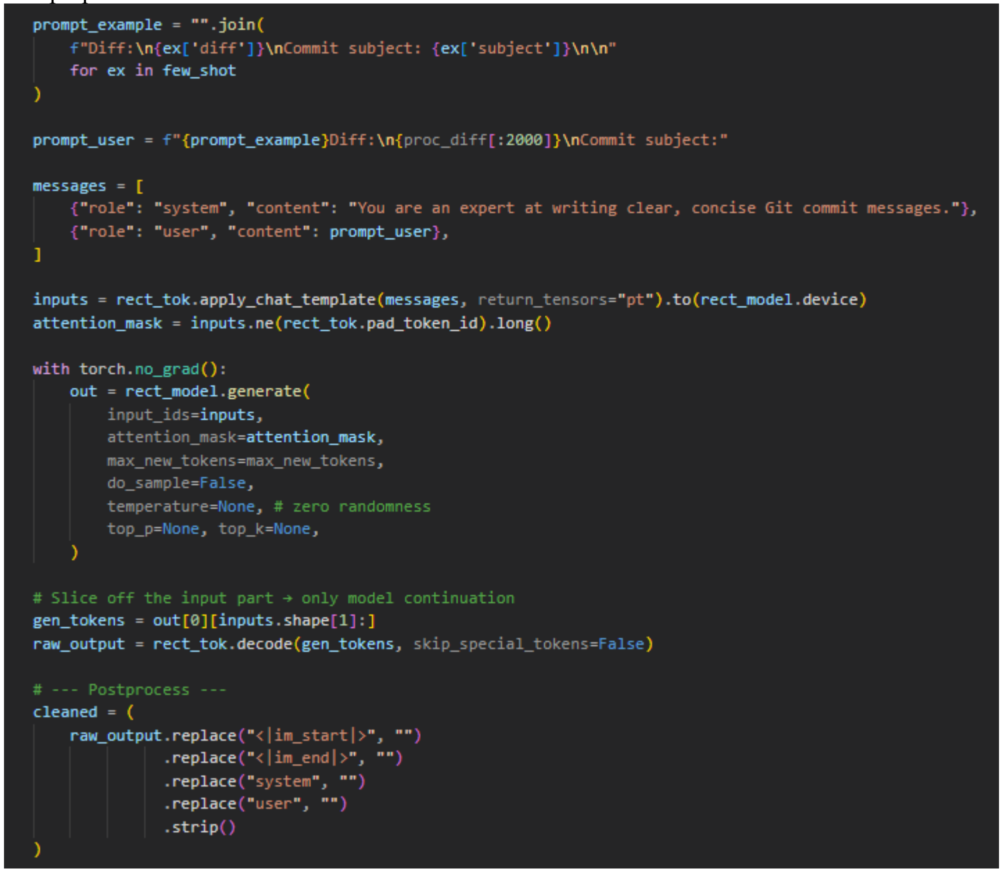
2. **Few-Shot Prompting**
   I constructed a few-shot prompts to guide the rectifier, including examples of diffs and their ideal commit subjects. These examples helped the model learn to produce **short, precise, imperative-style commit subjects**.



```python
# Few-shot examples
few_shot = [
    {"diff": "<del> def read()\n<add> def read(user_id)", "subject": "Add user_id argument to read()"},
    {"diff": "<add> logger = logging.getLogger(__name__)",
     "subject": "Add module-level logger"},
    {"diff": "<del> if x > 10:\n<add> if x >= 10:", "subject": "Fix off-by-one error in condition"},
    {"diff": "<del> requests.get(url)\n<add> httpx.get(url)",
     "subject": "Migrate calls from requests to httpx"},
]
```

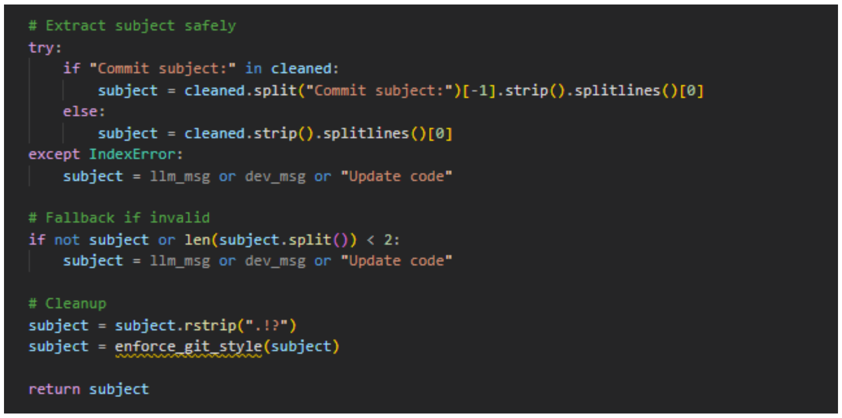
3. **Rectification Process**
   For each modified file, the model generated a refined subject line, taking the following as input:
   - The developer's original commit message. The model then generated a **refined subject line**.
   - The baseline LLM output.

- The preprocessed file-level diff.



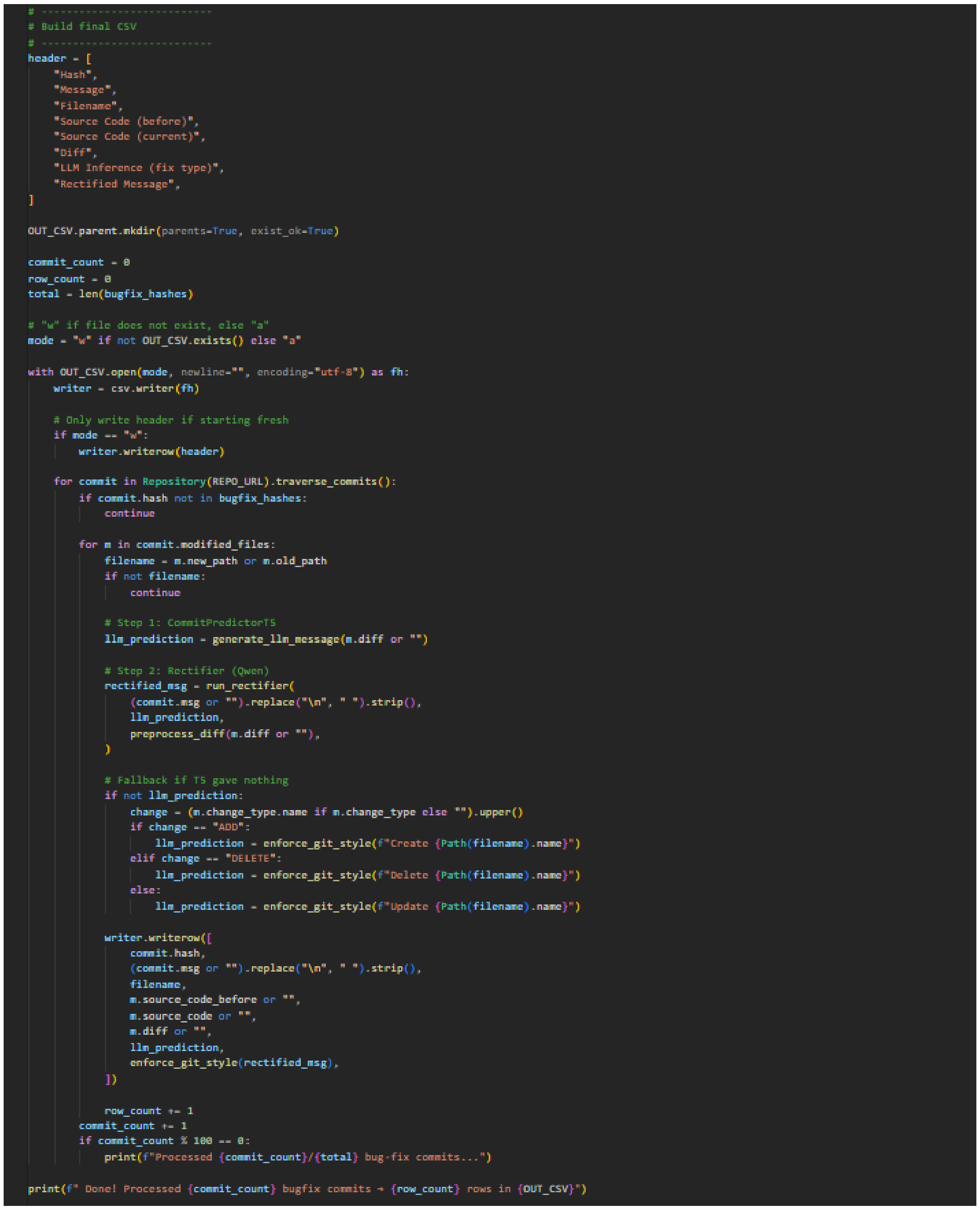
```python
prompt_example = "".join(
    f"Diff:\n{ex['diff']}\nCommit subject: {ex['subject']}\n\n"
    for ex in few_shot
)

prompt_user = f"{prompt_example}Diff:\n{proc_diff[:2000]}\nCommit subject:"

messages = [
    {"role": "system", "content": "You are an expert at writing clear, concise Git commit messages."},
    {"role": "user", "content": prompt_user},
]

inputs = rect_tok.apply_chat_template(messages, return_tensors="pt").to(rect_model.device)
attention_mask = inputs.ne(rect_tok.pad_token_id).long()

with torch.no_grad():
    out = rect_model.generate(
        input_ids=inputs,
        attention_mask=attention_mask,
        max_new_tokens=max_new_tokens,
        do_sample=False,
        temperature=None, # zero randomness
        top_p=None, top_k=None,
    )

# Slice off the input part → only model continuation
gen_tokens = out[0][inputs.shape[1]:]
raw_output = rect_tok.decode(gen_tokens, skip_special_tokens=False)

# --- Postprocess ---
cleaned = (
    raw_output.replace("<|im_start|>", "")
              .replace("<|im_end|>", "")
              .replace("system", "")
              .replace("user", "")
              .strip()
)
```

4. **Normalization and Fallback**
   - The rectified subject was cleaned with enforce_git_style() to ensure consistency.
   - If the rectifier failed (due to OOM or invalid output), the pipeline **defaulted to the baseline LLM message** or, if empty, the developer's message.



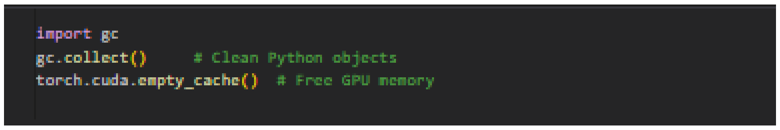
```python
# Extract subject safely
try:
    if "Commit subject:" in cleaned:
        subject = cleaned.split("Commit subject:")[-1].strip().splitlines()[0]
    else:
        subject = cleaned.strip().splitlines()[0]
except IndexError:
    subject = llm_msg or dev_msg or "Update code"

# Fallback if invalid
if not subject or len(subject.split()) < 2:
    subject = llm_msg or dev_msg or "Update code"

# Cleanup
subject = subject.rstrip(".!?")
subject = enforce_git_style(subject)

return subject
```

5. **CSV Storage**
   I appended both the LLM and the rectifier subjects to bugfix_diffs_with_llm_and_rectifier.csv.



```python
# --------------------------
# Build final CSV
# --------------------------
header = [
    "Hash",
    "Message",
    "Filename",
    "Source Code (before)",
    "Source Code (current)",
    "Diff",
    "LLM Inference (fix type)",
    "Rectified Message",
]

OUT_CSV.parent.mkdir(parents=True, exist_ok=True)

commit_count = 0
row_count = 0
total = len(bugfix_hashes)

# "w" if file does not exist, else "a"
mode = "w" if not OUT_CSV.exists() else "a"

with OUT_CSV.open(mode, newline="", encoding="utf-8") as fh:
    writer = csv.writer(fh)

    # Only write header if starting fresh
    if mode == "w":
        writer.writerow(header)

    for commit in Repository(REPO_URL).traverse_commits():
        if commit.hash not in bugfix_hashes:
            continue

        for m in commit.modified_files:
            filename = m.new_path or m.old_path
            if not filename:
                continue

            # Step 1: CommitPredictorT5
            llm_prediction = generate_llm_message(m.diff or "")

            # Step 2: Rectifier (Qwen)
            rectified_msg = run_rectifier(
                (commit.msg or "").replace("\n", " ").strip(),
                llm_prediction,
                preprocess_diff(m.diff or ""),
            )

            # Fallback if T5 gave nothing
            if not llm_prediction:
                change = (m.change_type.name if m.change_type else "").upper()
                if change == "ADD":
                    llm_prediction = enforce_git_style(f"Create {Path(filename).name}")
                elif change == "DELETE":
                    llm_prediction = enforce_git_style(f"Delete {Path(filename).name}")
                else:
                    llm_prediction = enforce_git_style(f"Update {Path(filename).name}")

            writer.writerow([
                commit.hash,
                (commit.msg or "").replace("\n", " ").strip(),
                filename,
                m.source_code_before or "",
                m.source_code or "",
                m.diff or "",
                llm_prediction,
                enforce_git_style(rectified_msg),
            ])

            row_count += 1
        commit_count += 1
        if commit_count % 100 == 0:
            print(f"Processed {commit_count}/{total} bug-fix commits...")

print(f" Done! Processed {commit_count} bugfix commits → {row_count} rows in {OUT_CSV}")
```

6. **Error Handling in Colab**
   Running both models on hundreds of commits consumed significant GPU memory. To avoid crashes:
   - I freed memory using gc.collect() and torch.cuda.empty_cache() between batches.



```python
import gc
gc.collect()      # Clean Python objects
torch.cuda.empty_cache()  # Free GPU memory
```

   - I also used batch processing (100 commits simultaneously) and later merged outputs into the master CSV.

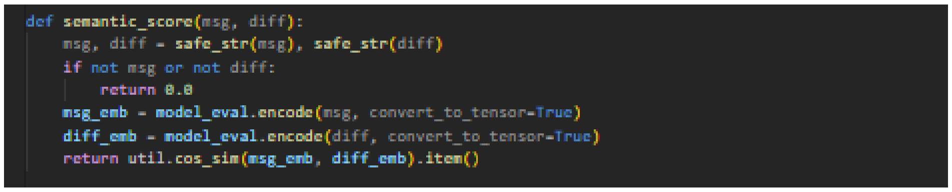
(f) Evaluation:

After generating the developer and LLM and rectifying the commit messages, I evaluated their quality with respect to the diffs using three research questions (RQ1–RQ3). To make the evaluation objective, I designed a **hybrid scoring framework** combining semantic similarity, keyword alignment, and readability.

1. Scoring Framework

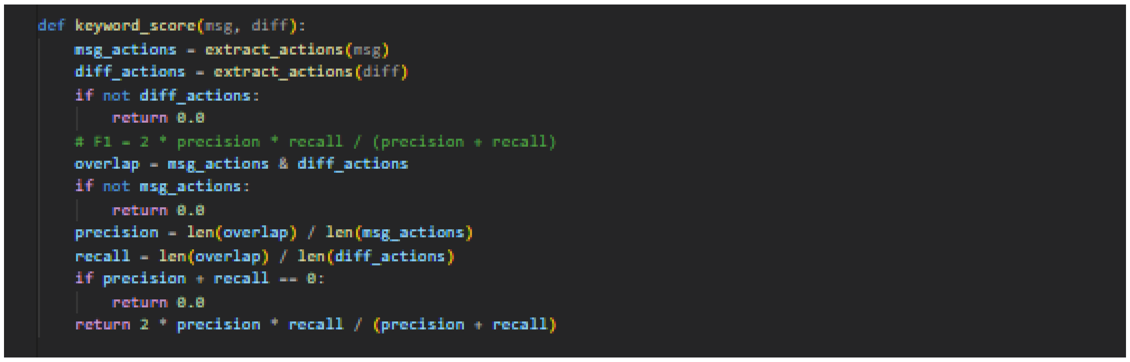
I created a composite score with three components:

- **Semantic Score**
  - Computed cosine similarity between embeddings of the commit message and the diff using *SentenceTransformer (all-MiniLM-L6-v2)*.
  - Captures whether the message semantically aligns with the actual code changes.



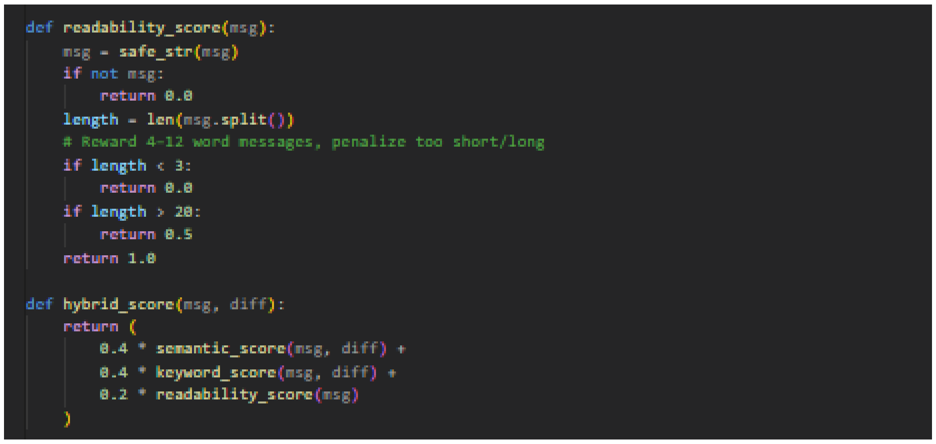
```python
def semantic_score(msg, diff):
    msg, diff = safe_str(msg), safe_str(diff)
    if not msg or not diff:
        return 0.0
    msg_emb = model_eval.encode(msg, convert_to_tensor=True)
    diff_emb = model_eval.encode(diff, convert_to_tensor=True)
    return util.cos_sim(msg_emb, diff_emb).item()
```

- **Keyword Score**
  - Extracted action keywords (*fix, add, remove, update, test, refactor,* etc.) from both the message and diff.
  - Used F1-style precision/recall overlap: Keyword Score = (2 × Precision × Recall) ÷ (Precision + Recall).



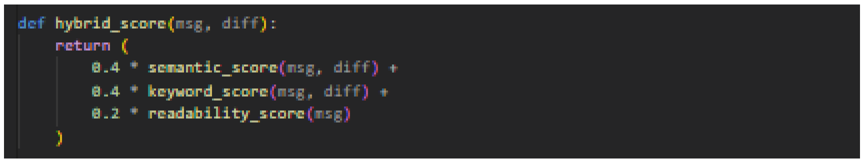
```python
def keyword_score(msg, diff):
    msg_actions = extract_actions(msg)
    diff_actions = extract_actions(diff)
    if not diff_actions:
        return 0.0
    # F1 = 2 * precision * recall / (precision + recall)
    overlap = msg_actions & diff_actions
    if not msg_actions:
        return 0.0
    precision = len(overlap) / len(msg_actions)
    recall = len(overlap) / len(diff_actions)
    if precision + recall == 0:
        return 0.0
    return 2 * precision * recall / (precision + recall)
```

- **Readability Score**
  - Rewarded commit concise but descriptive messages (4–12 words).
  - Too short → score = 0, too long → penalty applied.



```python
def readability_score(msg):
    msg = safe_str(msg)
    if not msg:
        return 0.0
    length = len(msg.split())
    # Reward 4-12 word messages, penalize too short/long
    if length < 3:
        return 0.0
    if length > 20:
        return 0.5
    return 1.0

def hybrid_score(msg, diff):
    return (
        0.4 * semantic_score(msg, diff) +
        0.4 * keyword_score(msg, diff) +
        0.2 * readability_score(msg)
    )
```

- **Final Hybrid Score**
  - Weighted average of the three: Hybrid Score = (0.4 × Semantic) + (0.4 × Keyword) + (0.2 × Readability)



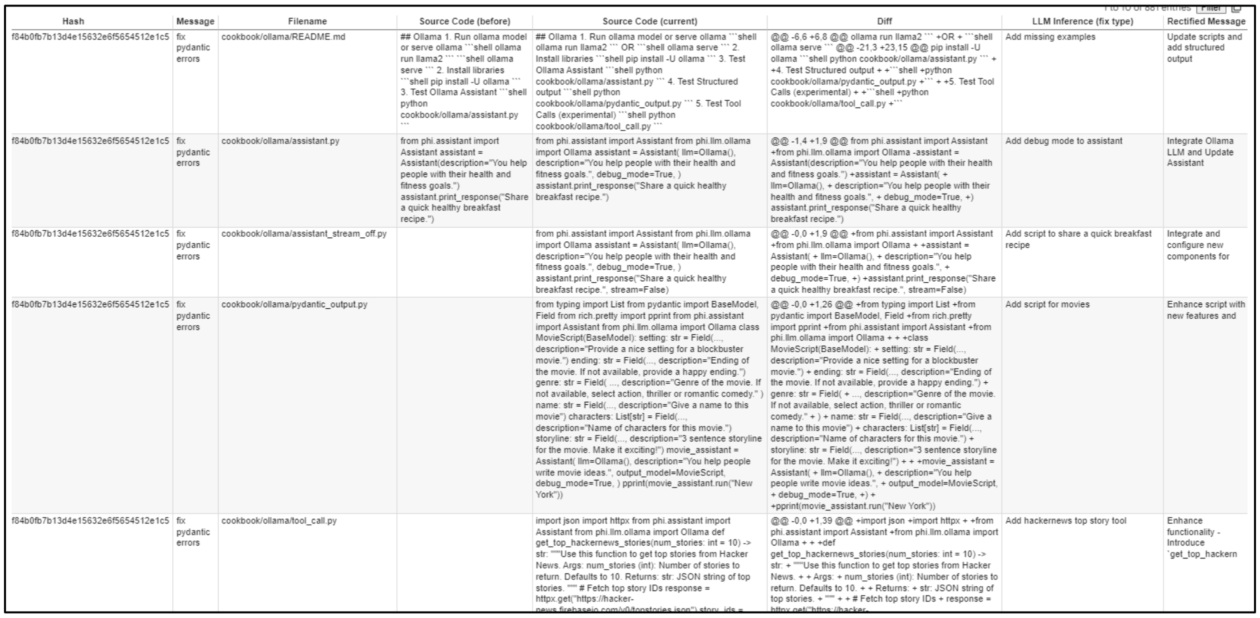
```python
def hybrid_score(msg, diff):
    return (
        0.4 * semantic_score(msg, diff) +
        0.4 * keyword_score(msg, diff) +
        0.2 * readability_score(msg)
    )
```

2. Outputs

- Printed mean scores for **RQ1, RQ2, and RQ3**.
- Plotted two bar charts:
  - o **Average Hybrid Scores** (Developer vs LLM vs Rectifier).
  - o **Count of Messages Above Threshold (0.5)** — showing how many commit messages were "good enough."

RESULTS AND ANALYSIS

Final per-file commit messages obtained from both commitT5 llm and rectifier CSV format:



Hash	Message	Filename	Source Code (before)	Source Code (current)	Diff	LLM Inference (fix type)	Rectified Message
f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	cookbook/ollama/README.md	## Ollama 1. Run ollama model or serve ollama ```shell ollama run llama2 ``` ```shell ollama serve ``` 2. Install libraries ```shell pip install -U ollama ``` 3. Test Ollama Assistant ```shell python cookbook/ollama/assistant.py ```	## Ollama 1. Run ollama model or serve ollama ```shell ollama run llama2 ``` OR ```shell ollama serve ``` 2. Install libraries ```shell pip install -U ollama ``` 3. Test Ollama Assistant ```shell python cookbook/ollama/assistant.py ``` 4. Test Structured output ```shell python cookbook/ollama/pydantic_output.py ``` 5. Test Tool Calls (experimental) ```shell python cookbook/ollama/tool_call.py ```	@@ -6,6 +6,8 @@ ollama run llama2 ``` +OR + ```shell ollama serve ``` @@ -21,3 +23,15 @@ pip install -U ollama ```shell python cookbook/ollama/assistant.py ``` + +4. Test Structured output + +```shell +python cookbook/ollama/pydantic_output.py +``` + +5. Test Tool Calls (experimental) + +```shell +python cookbook/ollama/tool_call.py +```	Add missing examples	Update scripts and add structured output
f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	cookbook/ollama/assistant.py	from phi.assistant import Assistant assistant = Assistant(description="You help people with their health and fitness goals.") assistant.print_response("Share a quick healthy breakfast recipe.")	from phi.assistant import Assistant from phi.llm.ollama import Ollama assistant = Assistant(llm=Ollama(), description="You help people with their health and fitness goals.", debug_mode=True,) assistant.print_response("Share a quick healthy breakfast recipe.")	@@ -1,4 +1,9 @@ from phi.assistant import Assistant +from phi.llm.ollama import Ollama -assistant = Assistant(description="You help people with their health and fitness goals.") +assistant = Assistant(+ llm=Ollama(), + description="You help people with their health and fitness goals.", + debug_mode=True, +) assistant.print_response("Share a quick healthy breakfast recipe.")	Add debug mode to assistant	Integrate Ollama LLM and Update Assistant
f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	cookbook/ollama/assistant_stream_off.py		from phi.assistant import Assistant from phi.llm.ollama import Ollama assistant = Assistant(llm=Ollama(), description="You help people with their health and fitness goals.", debug_mode=True,) assistant.print_response("Share a quick healthy breakfast recipe.", stream=False)	@@ -0,0 +1,9 @@ +from phi.assistant import Assistant +from phi.llm.ollama import Ollama + +assistant = Assistant(+ llm=Ollama(), + description="You help people with their health and fitness goals.", + debug_mode=True, +) +assistant.print_response("Share a quick healthy breakfast recipe.", stream=False)	Add script to share a quick breakfast recipe	Integrate and configure new components for
f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	cookbook/ollama/pydantic_output.py		from typing import List from pydantic import BaseModel, Field from rich.pretty import pprint from phi.assistant import Assistant from phi.llm.ollama import Ollama class MovieScript(BaseModel): setting: str = Field(..., description="Provide a nice setting for a blockbuster movie.") ending: str = Field(..., description="Ending of the movie. If not available, provide a happy ending.") genre: str = Field(..., description="Genre of the movie. If not available, select action, thriller or romantic comedy.") name: str = Field(..., description="Give a name to this movie") characters: List[str] = Field(..., description="Name of characters for this movie.") storyline: str = Field(..., description="3 sentence storyline for the movie. Make it exciting!") movie_assistant = Assistant(llm=Ollama(), description="You help people write movie ideas.", output_model=MovieScript, debug_mode=True,) pprint(movie_assistant.run("New York"))	@@ -0,0 +1,26 @@ +from typing import List +from pydantic import BaseModel, Field +from rich.pretty import pprint +from phi.assistant import Assistant +from phi.llm.ollama import Ollama + + +class MovieScript(BaseModel): + setting: str = Field(..., description="Provide a nice setting for a blockbuster movie.") + ending: str = Field(..., description="Ending of the movie. If not available, provide a happy ending.") + genre: str = Field(+ ..., description="Genre of the movie. If not available, select action, thriller or romantic comedy." +) + name: str = Field(..., description="Give a name to this movie") + characters: List[str] = Field(..., description="Name of characters for this movie.") + storyline: str = Field(..., description="3 sentence storyline for the movie. Make it exciting!") + + +movie_assistant = Assistant(+ llm=Ollama(), + description="You help people write movie ideas.", + output_model=MovieScript, + debug_mode=True, +) + +pprint(movie_assistant.run("New York"))	Add script for movies	Enhance script with new features and
f84b0fb7b13d4e15632e6f5654512e1c5	fix pydantic errors	cookbook/ollama/tool_call.py		import json import httpx from phi.assistant import Assistant from phi.llm.ollama import Ollama def get_top_hackernews_stories(num_stories: int = 10) -> str: """Use this function to get top stories from Hacker News. Args: num_stories (int): Number of stories to return. Defaults to 10. Returns: str: JSON string of top stories. """ # Fetch top story IDs response = httpx.get("https://hacker-	@@ -0,0 +1,39 @@ +import json +import httpx + +from phi.assistant import Assistant +from phi.llm.ollama import Ollama + + +def get_top_hackernews_stories(num_stories: int = 10) -> str: + """Use this function to get top stories from Hacker News. + + Args: + num_stories (int): Number of stories to return. Defaults to 10. + + Returns: + str: JSON string of top stories. + """ + + # Fetch top story IDs + response =	Add hackernews top story tool	Enhance functionality - Introduce `get_top_hackern

The results obtained for the three research questions (RQ1–RQ3) are as follows:

- **RQ1 (Developer hybrid hit rate):** 0.296
- **RQ2 (LLM hybrid hit rate):** 0.356
- **RQ3 (Rectifier hybrid hit rate):** 0.377

Key Observations:

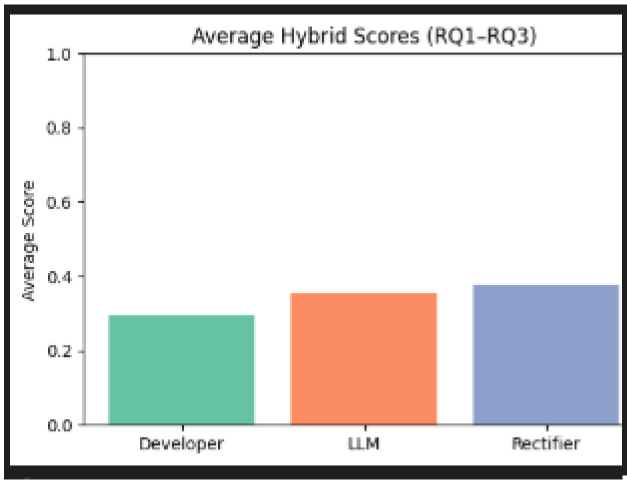
- **Developer Messages (RQ1):**
  - o The developer-written messages scored the lowest (≈0.30).
  - o This confirms that developer commit messages are often vague or overly broad, particularly because they tend to summarize multiple file changes in a single line rather than describing file-specific modifications.
- **LLM Messages (RQ2):**
  - o The baseline LLM model performed better (≈0.36).
  - o LLM-generated messages generally aligned with the diff but lacked fine-grained precision.
  - o Since the model was prompted to produce higher-level summaries, it often favored generality over detail.
- **Rectified Messages (RQ3):**
  - o The rectifier achieved the highest score (≈0.38), improving both developer and LLM baselines.
  - o The improvement over the LLM baseline, however, was modest (about 2%).

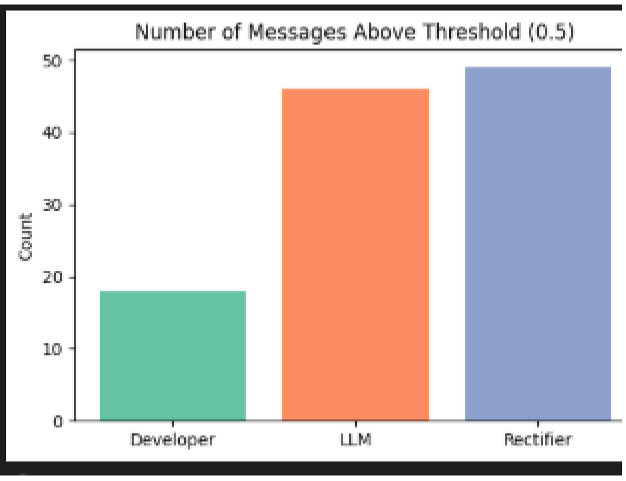
- This limited gain can be explained by the evaluation method: rectified messages, being concise and file-specific, often receive slightly lower semantic similarity scores when compared directly against the full git diff.

- **Comparative Insights:**
  - While the rectifier consistently produced short, precise messages and was easier to read, the hybrid scoring metric favored semantic overlap with the raw diff text.
  - This introduces a bias against rectified messages intentionally designed to abstract away low-level details.
  - Plots of average hybrid scores and counts above the threshold (0.5) confirmed these trends: developer messages lagged, the LLM baseline improved alignment, and the rectifier offered further but incremental gains.

Takeaways

- Developers' natural messages are often imprecise, validating the need for automated augmentation.
- LLMs provide a solid baseline, but their summaries can miss critical context at the file level without rectification.
- Rectifiers add value by generating precise, context-aware commit messages. Even though the improvement in the numeric hybrid score was modest, the **qualitative difference in readability and precision** was significant.





DISCUSSION AND CONCLUSION

Challenges Encountered:

- **LLM Pipelining Issues**
  - Integrating the rectifier LLM (Qwen) into the pipeline was the hardest part. Since Qwen is a chat-style LLM, its raw output often starts with tokens like *"system"* or unrelated formatting instead of a usable commit message.
  - To address this, I relied on extensive print statements to debug and understand the model's raw outputs, then post-processed them to extract clean commit messages.

- **GPU Limitations in Colab**
  - Running CodeBERT embeddings, CommitPredictorT5, and Qwen together was GPU-heavy.
  - Google Colab's limited GPU quota meant frequent interruptions, forcing me to shift between multiple accounts and re-run pipelines. This increased execution time and required careful checkpointing of intermediate results.

- **Variability in Rectifier Outputs**
  - Initially, the rectifier sometimes returned only a few tokens or incomplete commit messages.
  - I mitigated this by crafting a few-shot prompts with higher variation, which stabilized the output length and improved quality.

- **Balancing Evaluation and Generation**
  - The evaluation metric (hybrid score) was designed around semantic alignment with diffs, but rectified messages, being concise and abstract, sometimes received lower scores even though they were more useful in practice.

Reflections:

- Debugging LLM pipelines requires a very different mindset than running deterministic tools. Trial-and-error with prompts, output parsing, and error handling consumed most of the lab time.
- GPU resources are a real bottleneck for large-scale OSS mining. Without enough compute, progress gets throttled, and pipeline design must adapt to available resources.
- Rectifiers show clear promise but need robust prompting strategies to avoid incomplete or repetitive outputs.

Lessons Learned:

- **Print statements and post-processing are essential** when dealing with LLMs, since their outputs are non-deterministic and often contain unexpected tokens.
- **Resource management matters**, heavy pipelines require planning around limited GPU availability, making checkpoints and modular code design necessary.
- **Prompt design impacts output stability**, adding varied few-shot examples helped reduce truncated or generic commit messages.
- **Evaluation should balance semantic alignment with readability**; numerical scores alone don't capture the practical clarity of rectified messages.

Summary & Conclusion:

This lab showed how commit message rectification can bridge the gap between vague developer-written and overly generic LLM-generated messages. Despite GPU limitations and debugging challenges, the rectifier consistently produced clearer, file-specific messages compared to developers and the baseline LLM.

Quantitatively, the rectifier achieved the highest hybrid score (≈0.38), slightly better than the baseline LLM (≈0.36) and significantly above developer messages (≈0.30). Qualitatively, however, the rectifier offered the best precision and readability, confirming its value in improving commit documentation.

The main takeaway is that while LLM pipelines are robust, they require careful engineering, from prompt design to post-processing, and their effectiveness is tightly coupled with available computational resources.

REFERENCES

[3] https://pydriller.readthedocs.io/en/latest/index.html

[4] CS 202 STT Lecture 2 Slides

LABORATORY SESSION 3: Multi-Metric Bug Context Analysis and Agreement Detection in Bug-Fix Commit

INTRODUCTION

This laboratory session aimed to extend the bug-fix commit dataset created in Lab 2 by incorporating multiple software quality metrics and similarity measures. The lab focused on analyzing bug-fix commits using **structural metrics** (Maintainability Index, Cyclomatic Complexity, and Lines of Code) and comparing them with **change magnitude metrics** (semantic similarity via CodeBERT embeddings and token similarity via BLEU).

Through this session, I gained practical experience in:

- Applying *radon* to compute file-level Maintainability Index (MI), Cyclomatic Complexity (CC), and Lines of Code (LOC) before and after each bug fix.
- Using *CodeBERT* for semantic similarity and *SacreBLEU* for token similarity to measure the magnitude of changes.
- Classifying bug fixes as **Major** or **Minor** based on threshold values of similarity scores.
- Identifying cases where structural and similarity-based metrics give conflicting assessments, highlighting the complexity of evaluating code changes.

This experiment builds a bridge between **quantitative software quality measures** and **natural language/code similarity** measures, providing deeper insight into the relationship between the *size* of a code change and its *impact*.

SETUP AND TOOLS

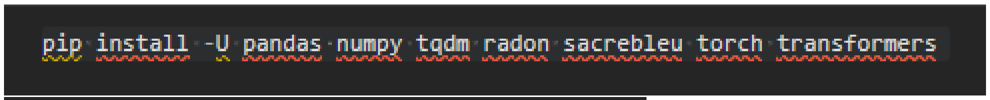
Before starting the lab tasks, I ensured my environment was configured correctly with all the required tools and dependencies. Since the work was carried out on **Google Colab**, the setup was straightforward with package installations performed via pip.

System and Tools:

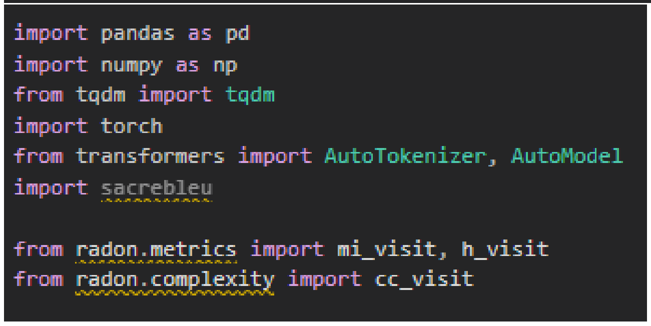
- **Platform**: Google Colab (cloud-based Jupyter environment)
- **Programming Language**: Python 3.10+
- **Data Analysis**: pandas (v2.2), numpy (v1.26), tqdm (v4.66)
- **Code Quality Metrics**: radon (v6.0) — for MI, CC, LOC extraction
- **Language Models**: CodeBERT (microsoft/codebert-base, via HuggingFace Transformers v4.43)
- **Similarity Scoring**: SacreBLEU (v2.4) — for token-based BLEU scoring
- **Deep Learning Backend**: PyTorch (v2.4) with GPU acceleration (CUDA) when available

Verification of Tools:

All tools were installed and verified within Colab:



```
pip install -U pandas numpy tqdm radon sacrebleu torch transformers
```



```
import pandas as pd
import numpy as np
from tqdm import tqdm
import torch
from transformers import AutoTokenizer, AutoModel
import sacrebleu

from radon.metrics import mi_visit, h_visit
from radon.complexity import cc_visit
```

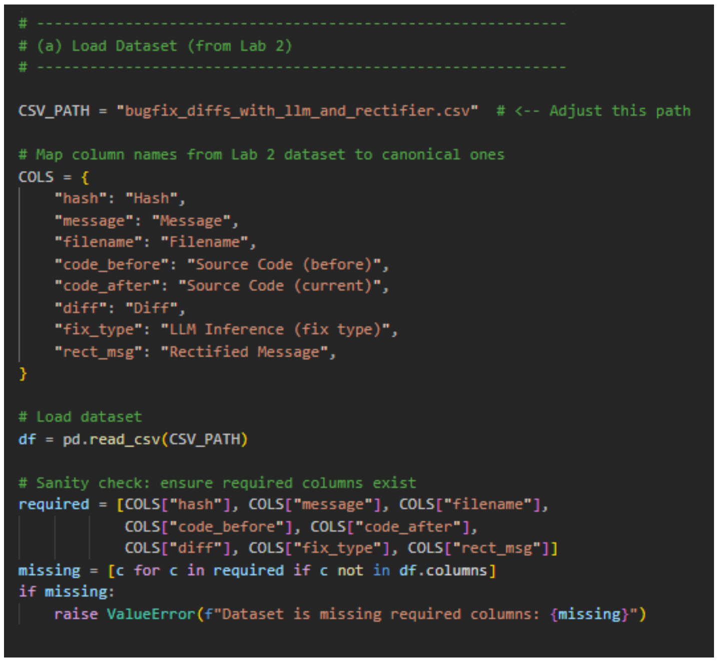
METHODOLOGY AND EXECUTION

The lab tasks were carried out in sequential steps, starting from the Lab 2 dataset and progressively adding multiple layers of analysis.

(a) Starting from the Lab 2 Dataset

I loaded the CSV produced in Lab 2 and normalized the column names. Hence, my pipeline always uses duplicate column keys to ensure consistent column names, which avoids subtle bugs later when I reference columns repeatedly.

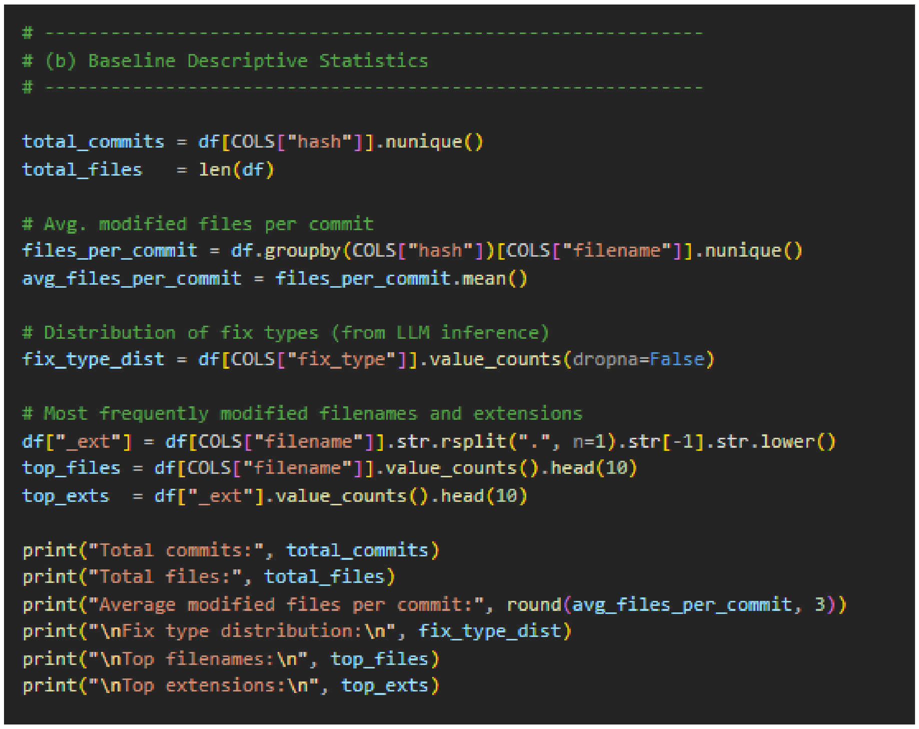
This dataset served as the baseline for computing both structural and similarity metrics.



```python
# -----------------------------------------------------------
# (a) Load Dataset (from Lab 2)
# -----------------------------------------------------------

CSV_PATH = "bugfix_diffs_with_llm_and_rectifier.csv"  # <-- Adjust this path

# Map column names from Lab 2 dataset to canonical ones
COLS = {
    "hash": "Hash",
    "message": "Message",
    "filename": "Filename",
    "code_before": "Source Code (before)",
    "code_after": "Source Code (current)",
    "diff": "Diff",
    "fix_type": "LLM Inference (fix type)",
    "rect_msg": "Rectified Message",
}

# Load dataset
df = pd.read_csv(CSV_PATH)

# Sanity check: ensure required columns exist
required = [COLS["hash"], COLS["message"], COLS["filename"],
            COLS["code_before"], COLS["code_after"],
            COLS["diff"], COLS["fix_type"], COLS["rect_msg"]]
missing = [c for c in required if c not in df.columns]
if missing:
    raise ValueError(f"Dataset is missing required columns: {missing}")
```

(b) Baseline Descriptive Statistics

I computed the basic distributions to describe the dataset in the report: totals, average files/commit, distribution of fix types, and the top file extensions.

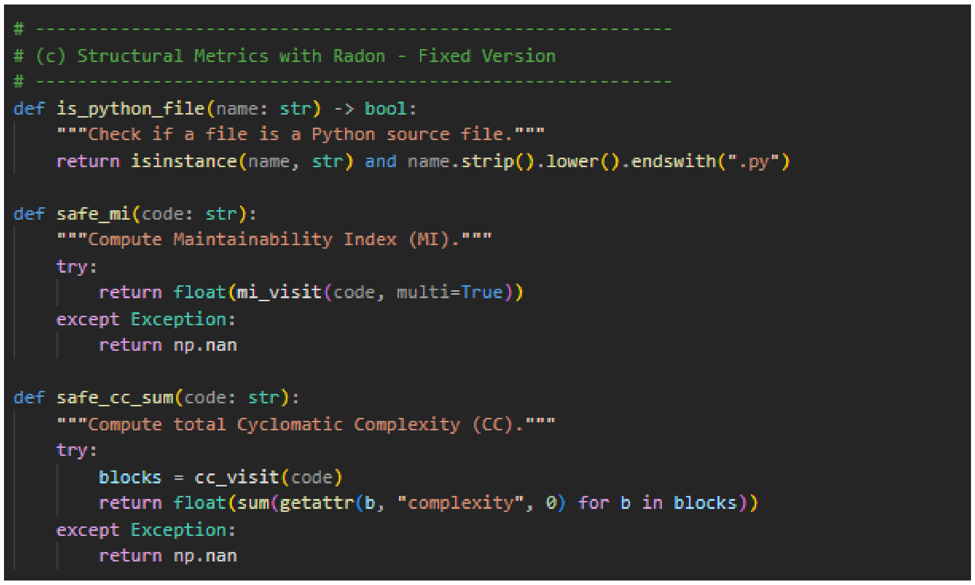


```python
# -----------------------------------------------------------
# (b) Baseline Descriptive Statistics
# -----------------------------------------------------------

total_commits = df[COLS["hash"]].nunique()
total_files   = len(df)

# Avg. modified files per commit
files_per_commit = df.groupby(COLS["hash"])[COLS["filename"]].nunique()
avg_files_per_commit = files_per_commit.mean()

# Distribution of fix types (from LLM inference)
fix_type_dist = df[COLS["fix_type"]].value_counts(dropna=False)

# Most frequently modified filenames and extensions
df["_ext"] = df[COLS["filename"]].str.rsplit(".", n=1).str[-1].str.lower()
top_files = df[COLS["filename"]].value_counts().head(10)
top_exts  = df["_ext"].value_counts().head(10)

print("Total commits:", total_commits)
print("Total files:", total_files)
print("Average modified files per commit:", round(avg_files_per_commit, 3))
print("\nFix type distribution:\n", fix_type_dist)
print("\nTop filenames:\n", top_files)
print("\nTop extensions:\n", top_exts)
```

(c) Structural Metrics with radon

I wrote small helper functions that call radon and handle errors. These wrappers ensure the pipeline is robust (Radon sometimes fails on fragments/diffs), and I keep sensible fallbacks.

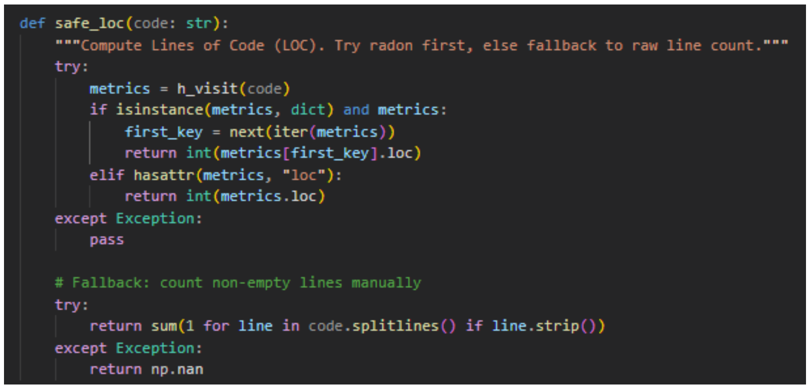
Key helper functions:



```python
# ------------------------------------------------------------
# (c) Structural Metrics with Radon - Fixed Version
# ------------------------------------------------------------
def is_python_file(name: str) -> bool:
    """Check if a file is a Python source file."""
    return isinstance(name, str) and name.strip().lower().endswith(".py")

def safe_mi(code: str):
    """Compute Maintainability Index (MI)."""
    try:
        return float(mi_visit(code, multi=True))
    except Exception:
        return np.nan

def safe_cc_sum(code: str):
    """Compute total Cyclomatic Complexity (CC)."""
    try:
        blocks = cc_visit(code)
        return float(sum(getattr(b, "complexity", 0) for b in blocks))
    except Exception:
        return np.nan
```

For each file, the buggy and fixed code versions were analyzed using **radon** to compute three structural quality indicators:

- **Maintainability Index (MI):** Higher values indicate better maintainability.
- **Cyclomatic Complexity (CC):** Measures control flow complexity; higher values suggest more difficult-to-maintain code.
- **Lines of Code (LOC):** Counts the number of lines in the file. Radon's h_visit can return structures that are sometimes empty when given fragments, so I implemented a robust safe_loc that uses radon when possible and falls back to a raw non-empty line count:

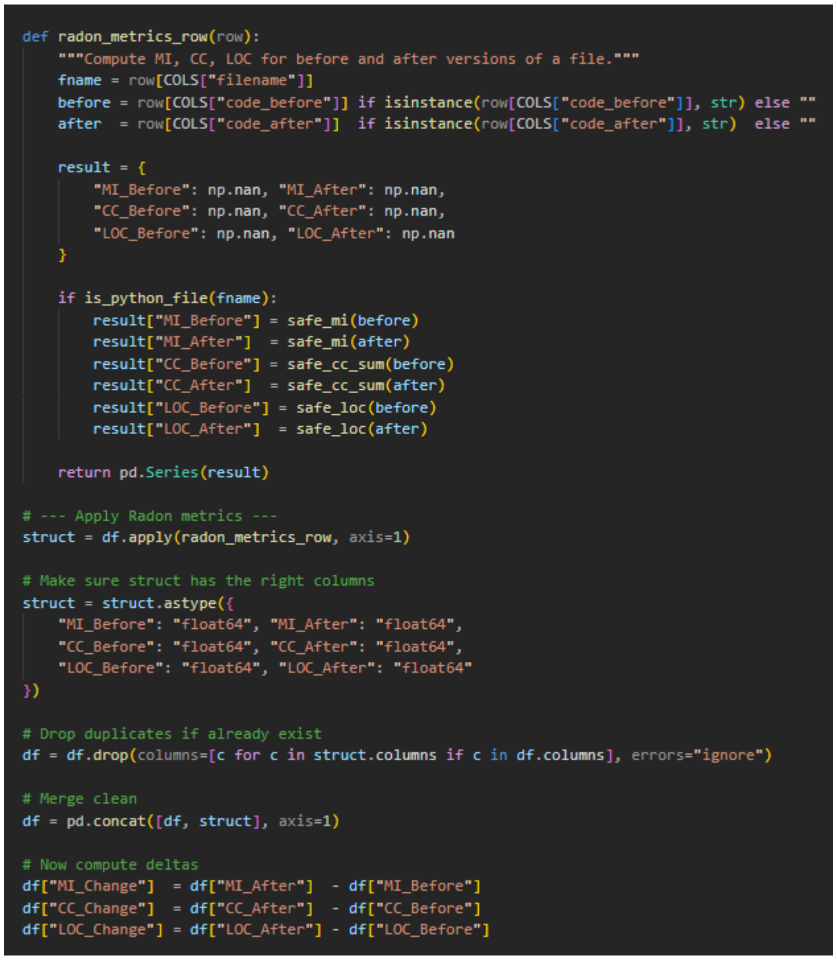


```python
def safe_loc(code: str):
    """Compute Lines of Code (LOC). Try radon first, else fallback to raw line count."""
    try:
        metrics = h_visit(code)
        if isinstance(metrics, dict) and metrics:
            first_key = next(iter(metrics))
            return int(metrics[first_key].loc)
        elif hasattr(metrics, "loc"):
            return int(metrics.loc)
    except Exception:
        pass

    # Fallback: count non-empty lines manually
    try:
        return sum(1 for line in code.splitlines() if line.strip())
    except Exception:
        return np.nan
```

For each file, I recorded:

- MI_Before, MI_After
- CC_Before, CC_After
- LOC_Before, LOC_After
- MI_Change = MI_After – MI_Before
- CC_Change = CC_After – CC_Before
- LOC_Change = LOC_After – LOC_Before

These metrics revealed whether the bug fix **improved or worsened** structural quality.



```python
def radon_metrics_row(row):
    """Compute MI, CC, LOC for before and after versions of a file."""
    fname = row[COLS["filename"]]
    before = row[COLS["code_before"]] if isinstance(row[COLS["code_before"]], str) else ""
    after  = row[COLS["code_after"]]  if isinstance(row[COLS["code_after"]], str)  else ""

    result = {
        "MI_Before": np.nan, "MI_After": np.nan,
        "CC_Before": np.nan, "CC_After": np.nan,
        "LOC_Before": np.nan, "LOC_After": np.nan
    }

    if is_python_file(fname):
        result["MI_Before"] = safe_mi(before)
        result["MI_After"]  = safe_mi(after)
        result["CC_Before"] = safe_cc_sum(before)
        result["CC_After"]  = safe_cc_sum(after)
        result["LOC_Before"] = safe_loc(before)
        result["LOC_After"]  = safe_loc(after)

    return pd.Series(result)

# --- Apply Radon metrics ---
struct = df.apply(radon_metrics_row, axis=1)

# Make sure struct has the right columns
struct = struct.astype({
    "MI_Before": "float64", "MI_After": "float64",
    "CC_Before": "float64", "CC_After": "float64",
    "LOC_Before": "float64", "LOC_After": "float64"
})

# Drop duplicates if already exist
df = df.drop(columns=[c for c in struct.columns if c in df.columns], errors="ignore")

# Merge clean
df = pd.concat([df, struct], axis=1)

# Now compute deltas
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"]
```

(d) Change Magnitude Metrics

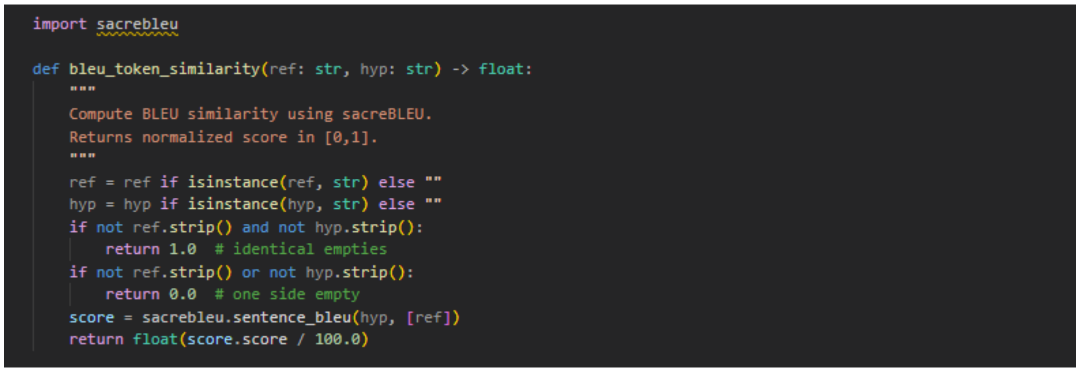
Two complementary similarity measures were applied to capture the **magnitude of change**:

- **Semantic Similarity:** Computed using **CodeBERT embeddings** and cosine similarity. This measured whether the functionality of the code remained close to the original.
- **Token Similarity:** Computed using **SacreBLEU**, which compares textual overlap between buggy and fixed code.

Both scores were normalized between **0 and 1**, where higher values meant the code was closer in meaning or tokens to its buggy version. Columns added:

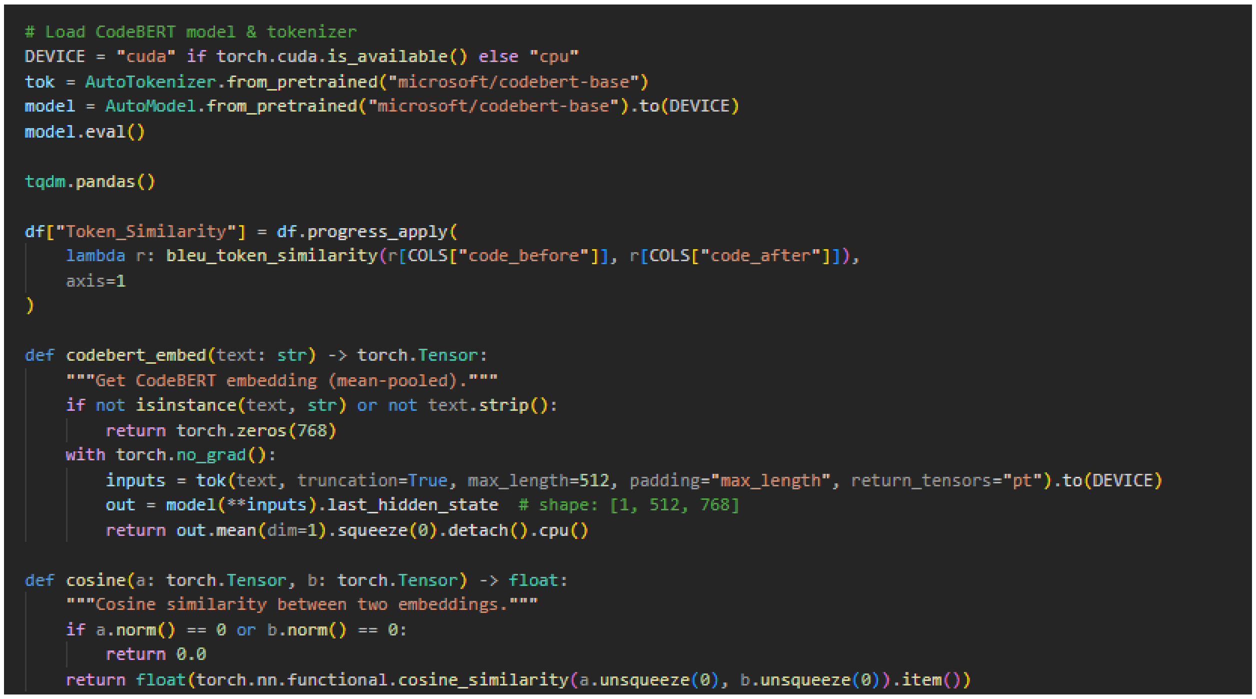
- Semantic_Similarity
- Token_Similarity

Token similarity (BLEU): I used sacreBLEU sentence_bleu normalized to [0,1]:



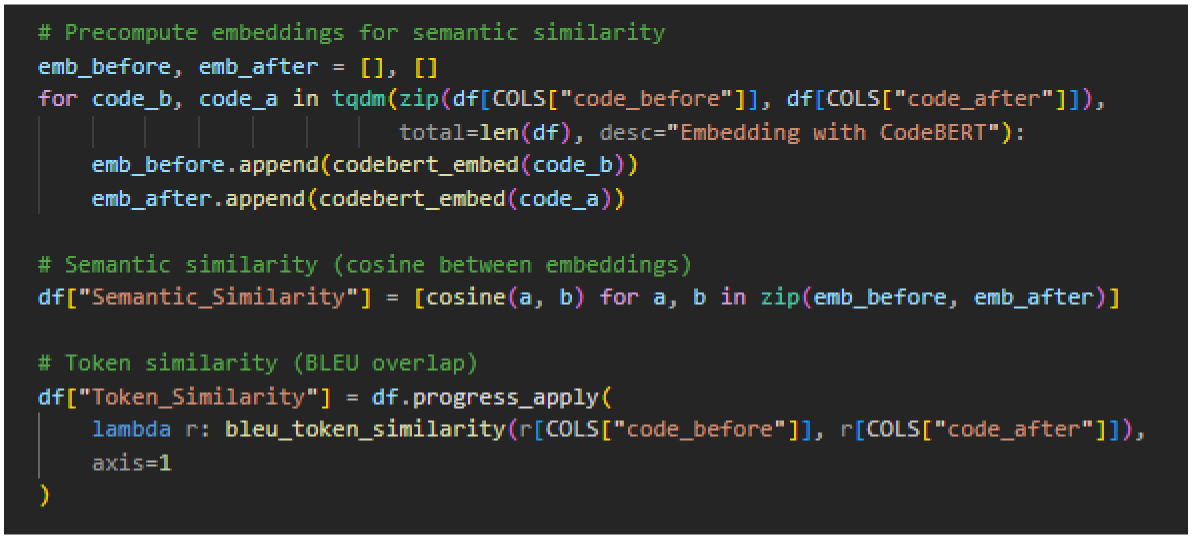
```python
import sacrebleu

def bleu_token_similarity(ref: str, hyp: str) -> float:
    """
    Compute BLEU similarity using sacreBLEU.
    Returns normalized score in [0,1].
    """
    ref = ref if isinstance(ref, str) else ""
    hyp = hyp if isinstance(hyp, str) else ""
    if not ref.strip() and not hyp.strip():
        return 1.0  # identical empties
    if not ref.strip() or not hyp.strip():
        return 0.0  # one side empty
    score = sacrebleu.sentence_bleu(hyp, [ref])
    return float(score.score / 100.0)
```

Semantic similarity (CodeBERT): Semantic similarity (CodeBERT): I loaded microsoft/codebert-base, tokenized the code, mean-pooled the last hidden state, and used cosine similarity:



```python
# Load CodeBERT model & tokenizer
DEVICE = "cuda" if torch.cuda.is_available() else "cpu"
tok = AutoTokenizer.from_pretrained("microsoft/codebert-base")
model = AutoModel.from_pretrained("microsoft/codebert-base").to(DEVICE)
model.eval()

tqdm.pandas()

df["Token_Similarity"] = df.progress_apply(
    lambda r: bleu_token_similarity(r[COLS["code_before"]], r[COLS["code_after"]]),
    axis=1
)

def codebert_embed(text: str) -> torch.Tensor:
    """Get CodeBERT embedding (mean-pooled)."""
    if not isinstance(text, str) or not text.strip():
        return torch.zeros(768)
    with torch.no_grad():
        inputs = tok(text, truncation=True, max_length=512, padding="max_length", return_tensors="pt").to(DEVICE)
        out = model(**inputs).last_hidden_state  # shape: [1, 512, 768]
        return out.mean(dim=1).squeeze(0).detach().cpu()

def cosine(a: torch.Tensor, b: torch.Tensor) -> float:
    """Cosine similarity between two embeddings."""
    if a.norm() == 0 or b.norm() == 0:
        return 0.0
    return float(torch.nn.functional.cosine_similarity(a.unsqueeze(0), b.unsqueeze(0)).item())
```

To speed up and monitor the process, I used tqdm. BLEU is a strict surface-level metric (sensitive to token reorderings and minor edits). CodeBERT embedding + cosine captures semantic preservation even when tokens change. Using both gives complementary perspectives.



```python
# Precompute embeddings for semantic similarity
emb_before, emb_after = [], []
for code_b, code_a in tqdm(zip(df[COLS["code_before"]], df[COLS["code_after"]]),
                           total=len(df), desc="Embedding with CodeBERT"):
    emb_before.append(codebert_embed(code_b))
    emb_after.append(codebert_embed(code_a))

# Semantic similarity (cosine between embeddings)
df["Semantic_Similarity"] = [cosine(a, b) for a, b in zip(emb_before, emb_after)]

# Token similarity (BLEU overlap)
df["Token_Similarity"] = df.progress_apply(
    lambda r: bleu_token_similarity(r[COLS["code_before"]], r[COLS["code_after"]]),
    axis=1
)
```

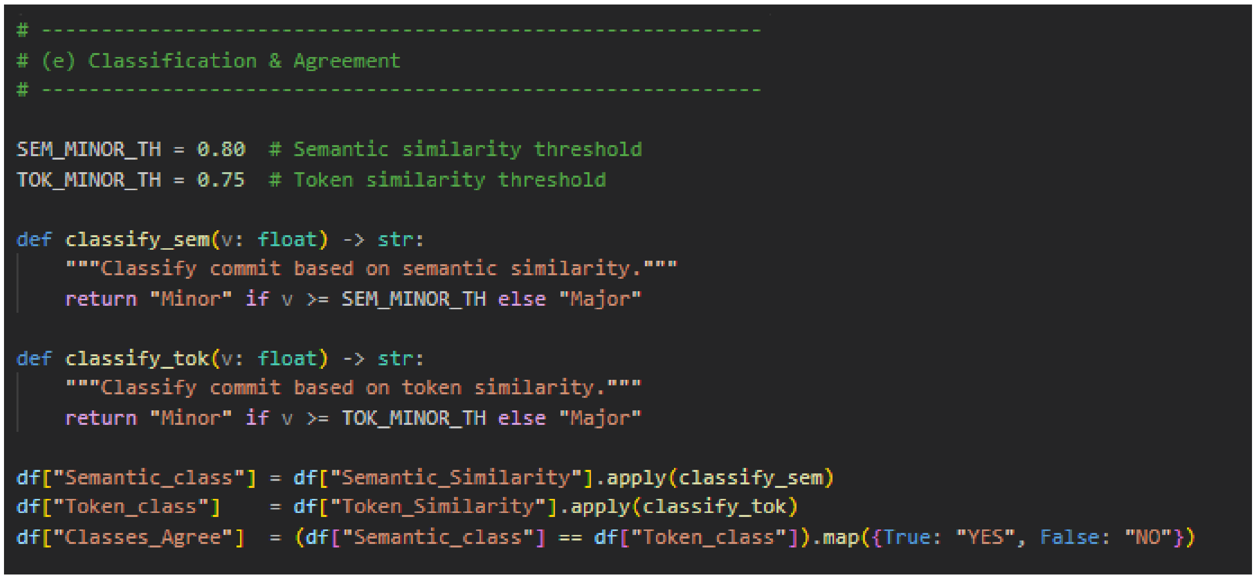
(e) Classification & Agreement

Based on predefined thresholds, bug fixes were classified as **Major** or **Minor**:

- **Semantic Similarity:**
  - ≥ 0.80 → Minor Fix
  - < 0.80 → Major Fix
- **Token Similarity:**
  - ≥ 0.75 → Minor Fix
  - < 0.75 → Major Fix

For each commit, I derived the Semantic class and the Token class. Finally, I compared both labels to identify agreement or disagreement:

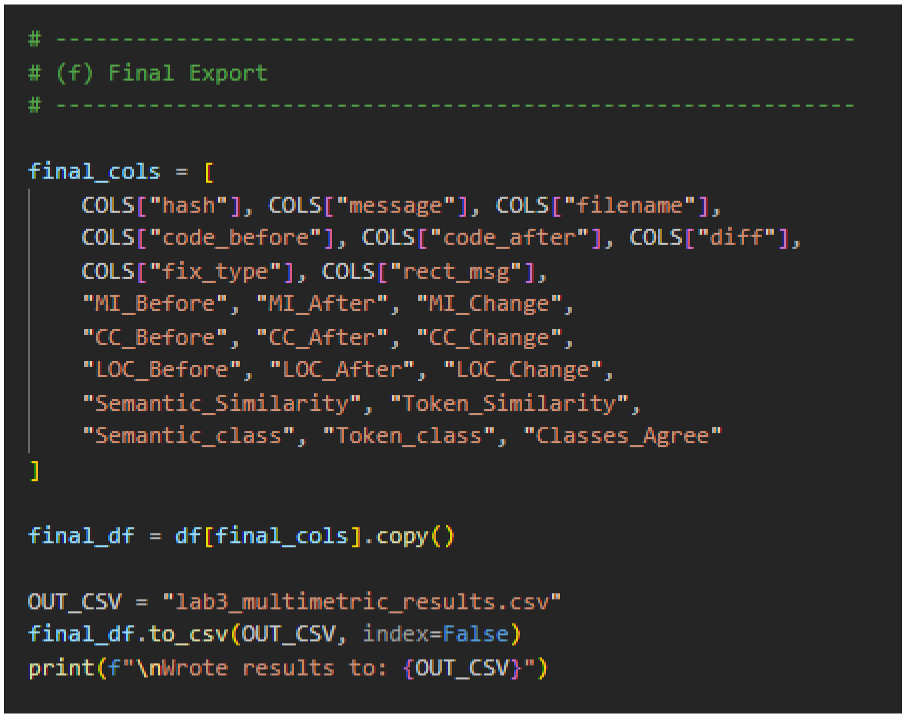
- Classes_Agree = "YES" if both classifications matched, otherwise "NO".



```
# -------------------------------------------------------------
# (e) Classification & Agreement
# -------------------------------------------------------------

SEM_MINOR_TH = 0.80  # Semantic similarity threshold
TOK_MINOR_TH = 0.75  # Token similarity threshold

def classify_sem(v: float) -> str:
    """Classify commit based on semantic similarity."""
    return "Minor" if v >= SEM_MINOR_TH else "Major"

def classify_tok(v: float) -> str:
    """Classify commit based on token similarity."""
    return "Minor" if v >= TOK_MINOR_TH else "Major"

df["Semantic_class"] = df["Semantic_Similarity"].apply(classify_sem)
df["Token_class"]    = df["Token_Similarity"].apply(classify_tok)
df["Classes_Agree"]  = (df["Semantic_class"] == df["Token_class"]).map({True: "YES", False: "NO"})
```

(f) Final Table

The dataset was extended with all new metrics and classification results, producing a final structure:



```
# -------------------------------------------------------------
# (f) Final Export
# -------------------------------------------------------------

final_cols = [
    COLS["hash"], COLS["message"], COLS["filename"],
    COLS["code_before"], COLS["code_after"], COLS["diff"],
    COLS["fix_type"], COLS["rect_msg"],
    "MI_Before", "MI_After", "MI_Change",
    "CC_Before", "CC_After", "CC_Change",
    "LOC_Before", "LOC_After", "LOC_Change",
    "Semantic_Similarity", "Token_Similarity",
    "Semantic_class", "Token_class", "Classes_Agree"
]

final_df = df[final_cols].copy()

OUT_CSV = "lab3_multimetric_results.csv"
final_df.to_csv(OUT_CSV, index=False)
print(f"\nWrote results to: {OUT_CSV}")
```

Error Handling

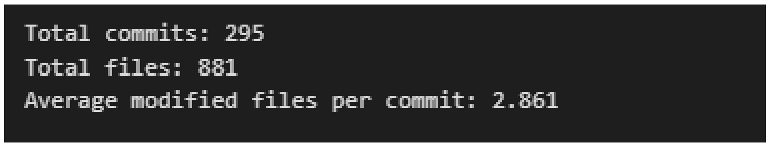
- **AttributeError:** 'DataFrame' object has no attribute 'progress_apply'
  - Did not enable tqdm integration with pandas. So I added tqdm.pandas() before using df.progress_apply().
- **ValueError:** Columns must be the same length as key (during metric concat)

- Duplicate metric columns were created when running the code multiple times in Colab. So I dropped existing metric columns before concatenation:
- df = df.drop(columns=[c for c in struct.columns if c in df.columns], errors="ignore")

- **LOC values all 0 or NaN**
  - h_visit output was inconsistent for the dataset's partial/invalid code fragments. Implemented a robust safe_loc that first tries radon's metrics and falls back to counting non-empty lines manually.

RESULTS AND ANALYSIS

1. Dataset Overview



```
Total commits: 295
Total files: 881
Average modified files per commit: 2.861
```

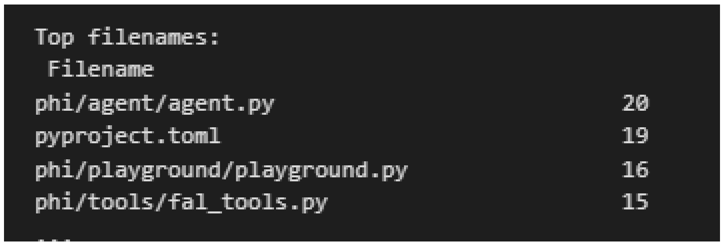
This indicates that each commit touched around three files on average, which is typical for bug-fix commits involving small changes.

2. File-Level Statistics

The most frequently modified files included:

- phi/agent/agent.py – 20 times
- pyproject. toml – 19 times
- phi/playground/playground.py – 16 times
- phi/tools/fal_tools.py – 15 times

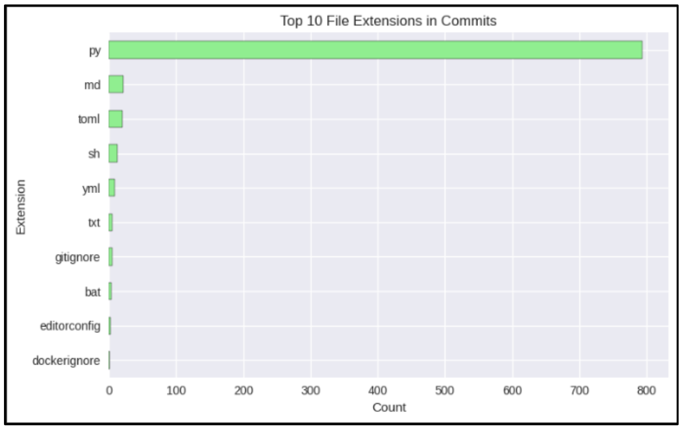
This indicates that agent.py and playground.py were hotspots for frequent changes, possibly due to complexity or evolving functionality.



```
Top filenames:
 Filename
phi/agent/agent.py                              20
pyproject.toml                                  19
phi/playground/playground.py                    16
phi/tools/fal_tools.py                          15
```

3. File Extensions:

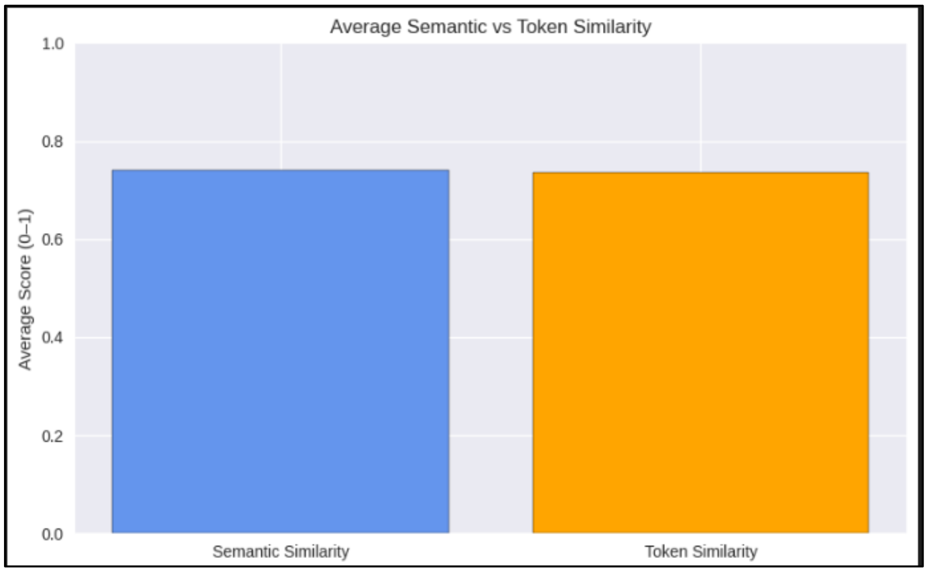
- .py files dominated (≈800 changes).
- .md files were following (20–30).
- Very few changes were seen in .dockerignore and .editorconfig (<5).



4. Similarity Measures

- Semantic Similarity (CodeBERT) ≈ 0.75
- Token Similarity (BLEU) ≈ 0.75
- The difference between the two was negligible.

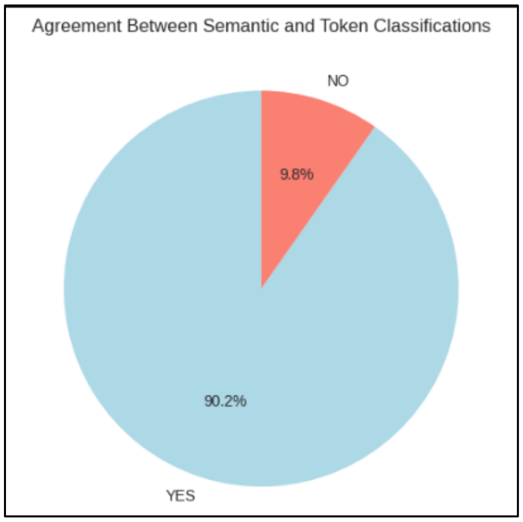
This shows that both measures show that most bug-fixes were minor, with before/after code being similar semantically and syntactically.



5. Agreement vs Disagreement

- **Agreement (YES)**: 90.2%
- **Disagreement (NO)**: 9.8%

Thus, in ~90% of the cases, semantic and token-based classifiers agreed on whether a commit was major or minor. The disagreements (~10%) occurred mainly in commits where the token-level changes were high, but semantics remained the same (e.g., renaming variables).

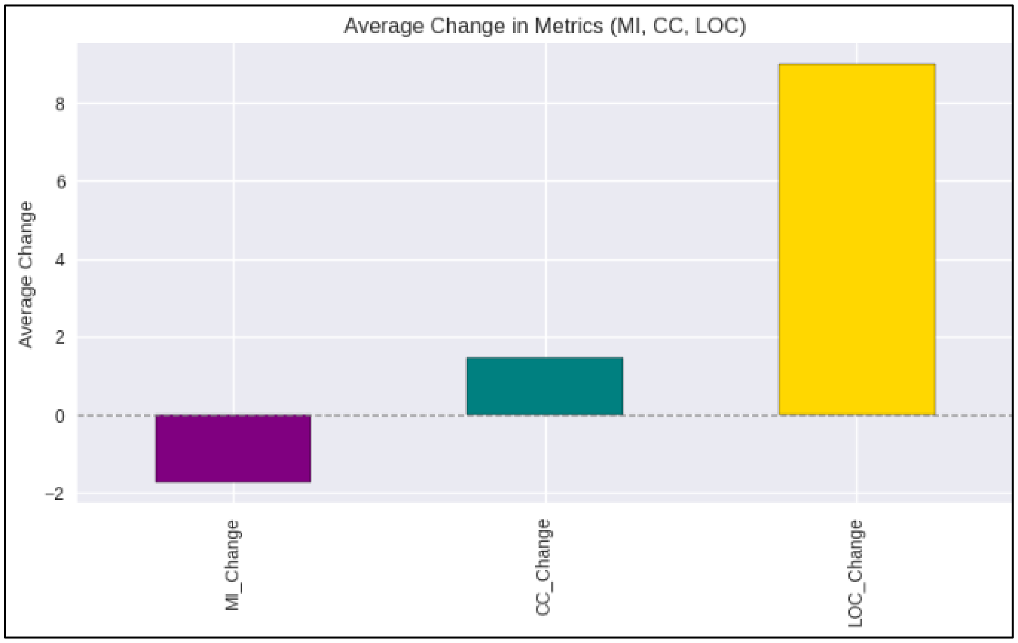


6. Structural Metric Changes

- **Maintainability Index (MI)**: Average **decrease of –1.8**

- **Cyclomatic Complexity (CC)**: Average **increase of +1.8**
- **Lines of Code (LOC)**: Average **increase of +9.5**

This shows that most bug-fixes introduced a slight increase in code complexity and size, while slightly reducing maintainability. However, the average changes are small, which aligns with the expectation that most bug-fixes are incremental.



DISCUSSION AND CONCLUSION

Challenges Encountered

- **AttributeError (progress_apply):** Fixed by enabling tqdm.pandas().
- **ValueError (column mismatch):** Resolved by dropping old metric columns before merging.
- **LOC values missing:** Fixed by enhancing safe_loc to fallback on raw non-empty line count.

Reflections

- This lab illustrated how **quantitative code quality metrics (MI, CC, LOC)** can be contrasted with **embedding-based similarity (CodeBERT)**.
- From the lecture, one key idea is that **software quality is multi-dimensional**: a bug-fix may improve correctness but worsen maintainability. My results confirmed this trade-off in practice.
- Semantic similarity (CodeBERT) is more stable and tolerant of token-level edits, while BLEU is stricter.

Key Lessons Learned

- **Most bug fixes are minor.** Both semantic and token similarity, along with small MI/CC/LOC changes, confirm this.
- **Token-level measures exaggerate changes.** BLEU marked more significant changes, even when semantics were preserved.
- **Structural metrics complement similarity.** While similarity captures *magnitude*, metrics like MI/CC/LOC reveal *quality impact*.
- **Hotspot analysis is proper.** Identifying frequently changed files highlights fragile or evolving parts of the system.

Summary & Conclusion

This lab demonstrated how multiple metrics can be integrated to study bug-fix commits. Using Radon, we quantified structural changes (MI, CC, LOC), while CodeBERT and BLEU provided semantic and token-level similarity. Results showed that:

- The dataset was dominated by minor fixes (imports, lint, small edits).
- Python files (.py) accounted for the vast majority of changes.

- Semantic and token similarities were almost identical (~0.75), with 90% agreement in classifications.
- Bug fixes slightly increased complexity and LOC, reducing maintainability by a small margin.
- Overall, the analysis confirms that bug fixes tend to be **small, localized, and incremental**, but even small changes can have measurable impacts on complexity and maintainability.

REFERENCES

[5] https://huggingface.co/microsoft/codebert-base

[6] https://radon.readthedocs.io/en/latest/

[7] CS 202 STT Lecture 3 Slides

LABORATORY SESSION 4: Exploration of different diff algorithms on Open-Source Repositories

INTRODUCTION

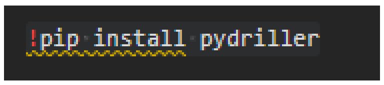
This lab aimed to examine how different diff algorithms (specifically Myers and histogram, as supported by git) produce different outputs on real-world commits. I mined three diverse open-source repositories, computed diffs for each modified file in commits using both algorithms, and analyzed where and why the outputs diverged. The investigation helps understand algorithmic impacts on code vs. non-code artifacts and supports choosing an appropriate diff strategy for downstream analysis (e.g., automated repair, patch extraction, documentation).

SETUP AND TOOLS

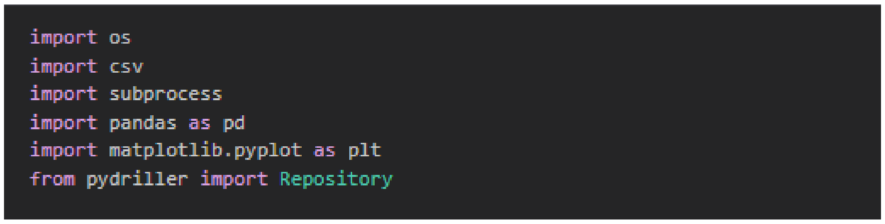
I ran the pipeline in Python (Google Colab / local machine). Key tools and why I used them:

- **Pydriller:** to traverse commits and get modified files and commit metadata; it provides a convenient Python API for repository mining.
- **git (subprocess)**: to run git diff with the --diff-algorithm flag (myers / histogram). I used subprocess because PyDriller does not expose an option to choose git's diff algorithm directly.
- **Pandas**: for dataset manipulation and CSV I/O.
- **Matplotlib**: for simple plots (mismatch counts by file category).
- **csv & os**: standard tools for writing CSV and managing local repository clones.

Packages installed:



```
!pip install pydriller
```



```
import os
import csv
import subprocess
import pandas as pd
import matplotlib.pyplot as plt
from pydriller import Repository
```

METHODOLOGY AND EXECUTION

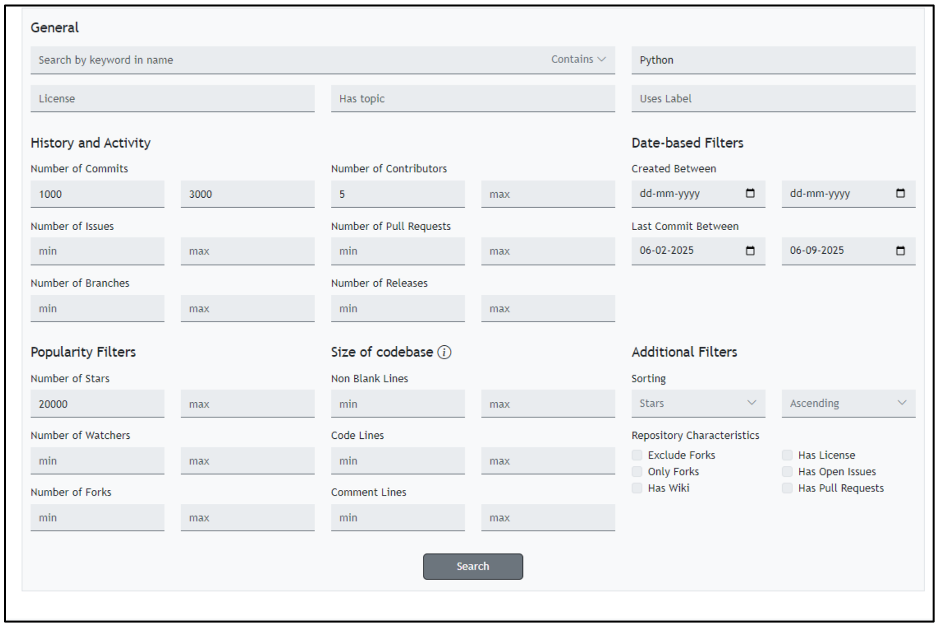
Repository Selection & Criteria

For this lab, I selected three real-world, medium-to-large-scale open-source repositories:

1. **copyparty** – https://github.com/9001/copyparty.git (repo1)
2. **deepface** – https://github.com/serengil/deepface.git (repo2)
3. **python-telegram-bot** – https://github.com/python-telegram-bot/python-telegram-bot.git (repo3)
4. I did not reuse any repositories from my previous labs. Instead, I systematically explored candidates using the **SEART GitHub Search Engine**, which allowed me to filter projects based on activity and popularity.

Selection filters I applied:

- **Commit count between 1000–5000**: ensures a sufficiently large history for analysis but avoids huge repos where diff extraction would be computationally heavy (a problem I faced in Lab 2).
- **Primary language = Python**: chosen for easier parsing and handling within *PyDriller*, since the library handles Python projects more reliably.
- **Stars ≥ 20,000**: ensures the project is well-recognized, actively maintained, and widely used, making results representative of real-world development practices.
- **Last commit within 6 months**: guarantees that the repositories are active and reflect current development practices rather than abandoned projects.


General

Search by keyword in name — Contains — Python

License | Has topic | Uses Label

History and Activity

Number of Commits: 1000 – 3000

Number of Contributors: 5 – max

Number of Issues: min – max

Number of Pull Requests: min – max

Number of Branches: min – max

Number of Releases: min – max

Date-based Filters

Created Between: dd-mm-yyyy – dd-mm-yyyy

Last Commit Between: 06-02-2025 – 06-09-2025

Popularity Filters

Number of Stars: 20000 – max

Number of Watchers: min – max

Number of Forks: min – max

Size of codebase

Non Blank Lines: min – max

Code Lines: min – max

Comment Lines: min – max

Additional Filters

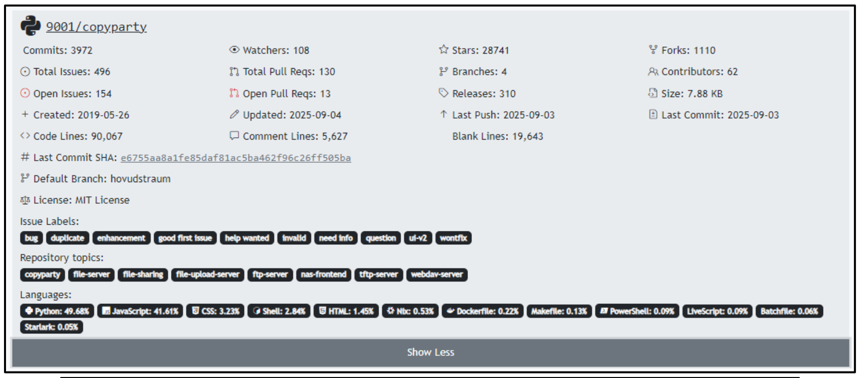
Sorting: Stars — Ascending

Repository Characteristics: Exclude Forks, Only Forks, Has Wiki, Has License, Has Open Issues, Has Pull Requests

Search

Selection rationale:

- All three repositories are **actively maintained** and have a **substantial commit history** that fits within my computational budget.
- They represent **different domains and codebases** — a file server/utility (copyparty), a machine learning library (deepface), and a bot framework (python-telegram-bot). This diversity increases the chance of observing how diff algorithms behave differently across file types and domains.
- By applying systematic criteria rather than choosing arbitrarily, I ensured that my dataset is **popular (high stars)** and **recent (active commits)**, while remaining feasible to process in Colab with limited GPU/CPU resources.


9001/copyparty

Commits: 3972	Watchers: 108	Stars: 28741	Forks: 1110
Total Issues: 496	Total Pull Reqs: 130	Branches: 4	Contributors: 62
Open Issues: 154	Open Pull Reqs: 13	Releases: 310	Size: 7.88 KB
Created: 2019-05-26	Updated: 2025-09-04	Last Push: 2025-09-03	Last Commit: 2025-09-03
Code Lines: 90,067	Comment Lines: 5,627	Blank Lines: 19,643	

Last Commit SHA: e6755aa8a1fe85daf81ac5ba462f96c26ff505ba

Default Branch: hovudstraum

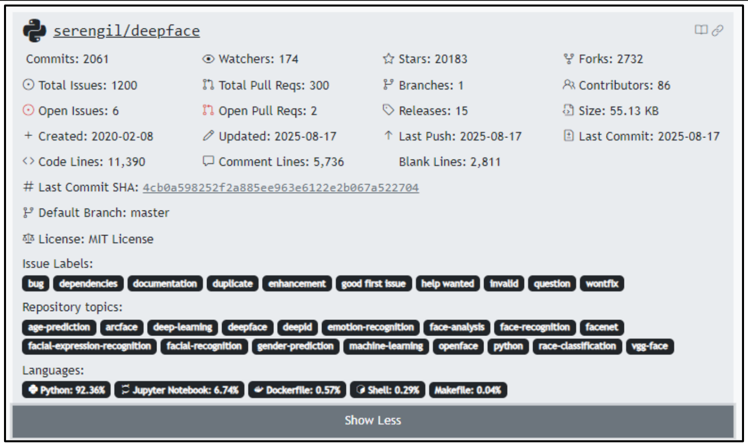
License: MIT License

Issue Labels: bug, duplicate, enhancement, good first issue, help wanted, invalid, need info, question, ui-v2, wontfix

Repository topics: copyparty, file-server, file-sharing, file-upload-server, ftp-server, nas-frontend, tftp-server, webdav-server

Languages: Python: 49.68%, JavaScript: 41.61%, CSS: 3.23%, Shell: 2.84%, HTML: 1.45%, Nix: 0.53%, Dockerfile: 0.22%, Makefile: 0.13%, PowerShell: 0.09%, LiveScript: 0.09%, Batchfile: 0.06%, Starlark: 0.05%

Show Less


serengil/deepface

Commits: 2061	Watchers: 174	Stars: 20183	Forks: 2732
Total Issues: 1200	Total Pull Reqs: 300	Branches: 1	Contributors: 86
Open Issues: 6	Open Pull Reqs: 2	Releases: 15	Size: 55.13 KB
Created: 2020-02-08	Updated: 2025-08-17	Last Push: 2025-08-17	Last Commit: 2025-08-17
Code Lines: 11,390	Comment Lines: 5,736	Blank Lines: 2,811	

Last Commit SHA: 4cb0a598252f2a885ee963e6122e2b067a522704

Default Branch: master

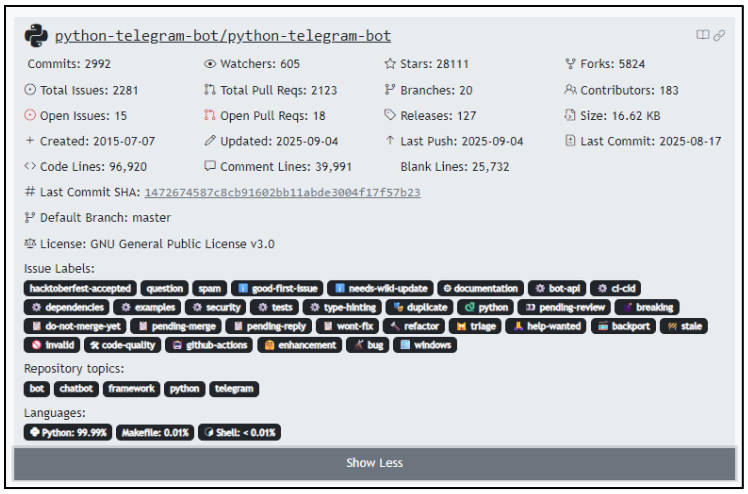
License: MIT License

Issue Labels: bug, dependencies, documentation, duplicate, enhancement, good first issue, help wanted, invalid, question, wontfix

Repository topics: age-prediction, arcface, deep-learning, deepface, deepid, emotion-recognition, face-analysis, face-recognition, facenet, facial-expression-recognition, facial-recognition, gender-prediction, machine-learning, openface, python, race-classification, vgg-face

Languages: Python: 92.36%, Jupyter Notebook: 6.74%, Dockerfile: 0.57%, Shell: 0.29%, Makefile: 0.04%

Show Less



python-telegram-bot/python-telegram-bot

Commits: 2992 | Watchers: 605 | Stars: 28111 | Forks: 5824

Total Issues: 2281 | Total Pull Reqs: 2123 | Branches: 20 | Contributors: 183

Open Issues: 15 | Open Pull Reqs: 18 | Releases: 127 | Size: 16.62 KB

Created: 2015-07-07 | Updated: 2025-09-04 | Last Push: 2025-09-04 | Last Commit: 2025-08-17

Code Lines: 96,920 | Comment Lines: 39,991 | Blank Lines: 25,732

Last Commit SHA: 1472674587c8cb91602bb11abde3004f17f57b23

Default Branch: master

License: GNU General Public License v3.0

Issue Labels:

hacktoberfest-accepted, question, spam, good-first-issue, needs-wiki-update, documentation, bot-api, ci-cid, dependencies, examples, security, tests, type-hinting, duplicate, python, pending-review, breaking, do-not-merge-yet, pending-merge, pending-reply, wont-fix, refactor, triage, help-wanted, backport, stale, invalid, code-quality, github-actions, enhancement, bug, windows

Repository topics:

bot, chatbot, framework, python, telegram

Languages:

Python: 99.99%, Makefile: 0.01%, Shell: < 0.01%

(a) Clone / prepare repository

extract_dataset(repo_url, local_path, dataset_file):

- If the repo is not present locally, clone it. This avoids repeated network operations later.
- I traverse commits with pydriller.Repository(local_path).traverse_commits(), this yields commit objects and their modified files.

Though PyDriller simplifies getting modified file paths and commit context, direct git is necessary for applying specific--diff-algorithm options.

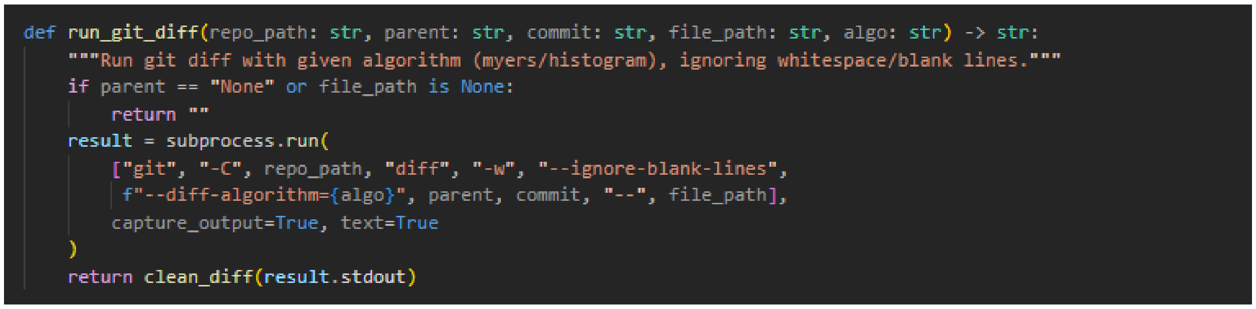
```python
def extract_dataset(repo_url: str, local_path: str, dataset_file: str):
    """Extract commits and modified files from one repo into dataset.csv"""
    # Clone locally if not exists
    if not os.path.exists(local_path):
        subprocess.run(["git", "clone", repo_url, local_path])

    with open(dataset_file, "w", newline="", encoding="utf-8") as f:
        writer = csv.writer(f)
        writer.writerow([
            "old_file_path", "new_file_path",
            "commit_SHA", "parent_commit_SHA",
            "commit_message", "diff_myers", "diff_hist"
        ])
        for commit in Repository(local_path).traverse_commits():
            parent_commits = commit.parents if commit.parents else ["None"]
            parent = parent_commits[0]

            for m in commit.modified_files:
                file_path = m.new_path or m.old_path
                diff_myers = run_git_diff(local_path, parent, commit.hash, file_path, "myers")
                diff_hist = run_git_diff(local_path, parent, commit.hash, file_path, "histogram")

                writer.writerow([
                    m.old_path, m.new_path,
                    commit.hash, parent,
                    commit.msg.strip(),
                    diff_myers, diff_hist
                ])
    print(f"dataset written to {dataset_file}")
```

(b) Compute diffs using two algorithms

run_git_diff(repo_path, parent, commit, file_path, algo):

- Uses git -C <repo> diff -w --ignore-blank-lines --diff-algorithm=<algo> <parent> <commit> -- <file>
- Options:
  - -w ignores whitespace differences.

    - --ignore-blank-lines ignores blank-line differences.
    - --diff-algorithm=myers or --diff-algorithm=histogram selects algorithm.
- -w and --ignore-blank-lines reduce spurious mismatches due to formatting only; we want substantive differences.
- Comparing Myers vs histogram shows algorithmic sensitivity (where one algorithm groups/aligns changed lines differently than another).

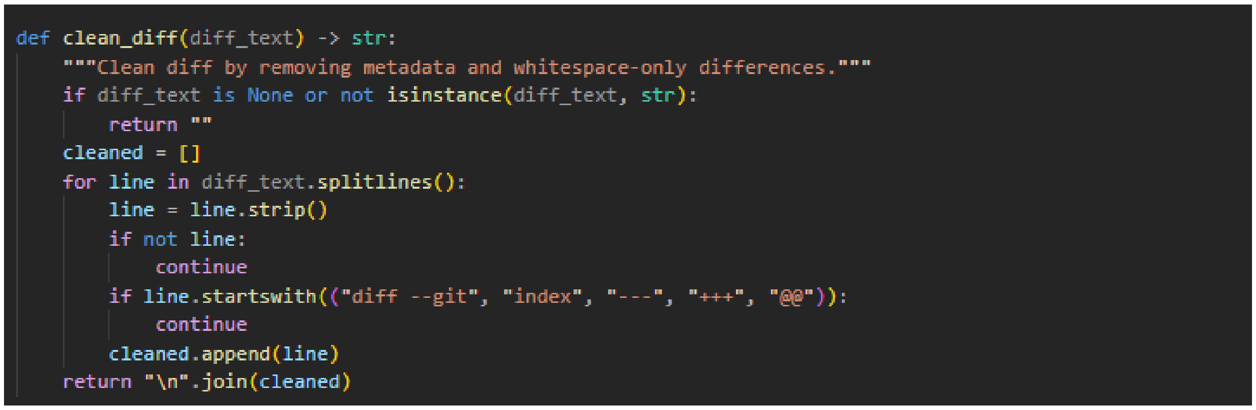


```python
def run_git_diff(repo_path: str, parent: str, commit: str, file_path: str, algo: str) -> str:
    """Run git diff with given algorithm (myers/histogram), ignoring whitespace/blank lines."""
    if parent == "None" or file_path is None:
        return ""
    result = subprocess.run(
        ["git", "-C", repo_path, "diff", "-w", "--ignore-blank-lines",
         f"--diff-algorithm={algo}", parent, commit, "--", file_path],
        capture_output=True, text=True
    )
    return clean_diff(result.stdout)
```

(c) Clean diffs for fair comparison

clean_diff(diff_text):

- Removes diff metadata lines (diff --git, index, ---, +++, @@) and strips whitespace-only lines.
- Only content lines (added/removed code context) are normalized.
- Raw git diff output contains header/hunk metadata that will always differ even if algorithms produce similar hunks. Cleaning helps focus the comparison on the changed content itself.



```python
def clean_diff(diff_text) -> str:
    """Clean diff by removing metadata and whitespace-only differences."""
    if diff_text is None or not isinstance(diff_text, str):
        return ""
    cleaned = []
    for line in diff_text.splitlines():
        line = line.strip()
        if not line:
            continue
        if line.startswith(("diff --git", "index", "---", "+++", "@@")):
            continue
        cleaned.append(line)
    return "\n".join(cleaned)
```

(d) Store dataset

In extract_dataset, I write rows to dataset_file with columns:

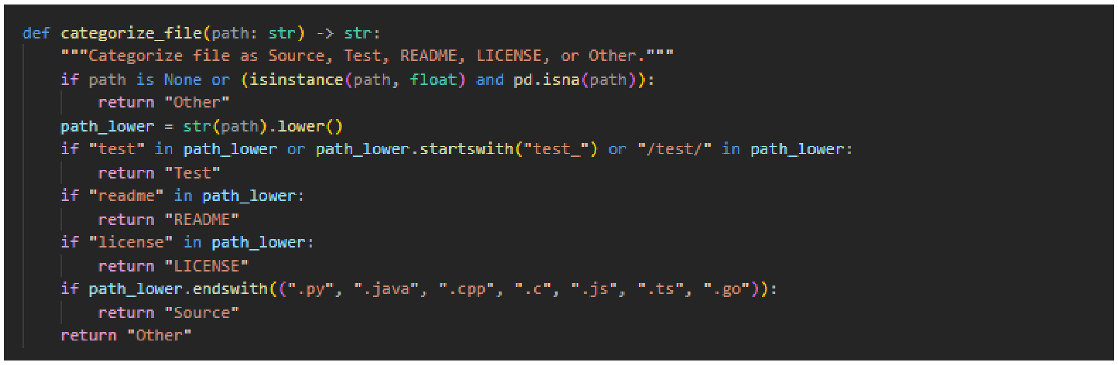
old_file_path, new_file_path, commit_SHA, parent_commit_SHA, commit_message, diff_myers, diff_hist

Each row corresponds to one modified file in a commit. Developers often change multiple files in one commit; analyzing per-file diffs gives a finer-grained view and lets us categorize by file type (code vs tests vs docs).

(e) Add discrepancy label and categorize files

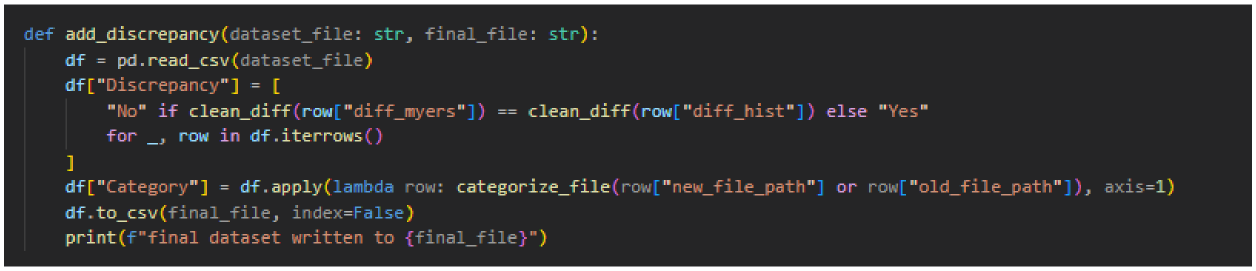
add_discrepancy(dataset_file, final_file):

- Loads the dataset_file.
- Compares cleaned diff_myers and cleaned diff_hist for equality:
    - If equal → Discrepancy = "No".
    - If different → Discrepancy = "Yes".
- Adds Category for each row using categorize_file(path):
    - Categories: Source, Test, README, LICENSE, Other (based on filename/extension and simple keyword heuristics).



```python
def categorize_file(path: str) -> str:
    """Categorize file as Source, Test, README, LICENSE, or Other."""
    if path is None or (isinstance(path, float) and pd.isna(path)):
        return "Other"
    path_lower = str(path).lower()
    if "test" in path_lower or path_lower.startswith("test_") or "/test/" in path_lower:
        return "Test"
    if "readme" in path_lower:
        return "README"
    if "license" in path_lower:
        return "LICENSE"
    if path_lower.endswith((".py", ".java", ".cpp", ".c", ".js", ".ts", ".go")):
        return "Source"
    return "Other"
```

- Writes final_file (final_dataset).
- The binary label is a compact way to count where algorithms diverge.
- Category lets us analyze whether differences primarily affect code, tests, or documentation.

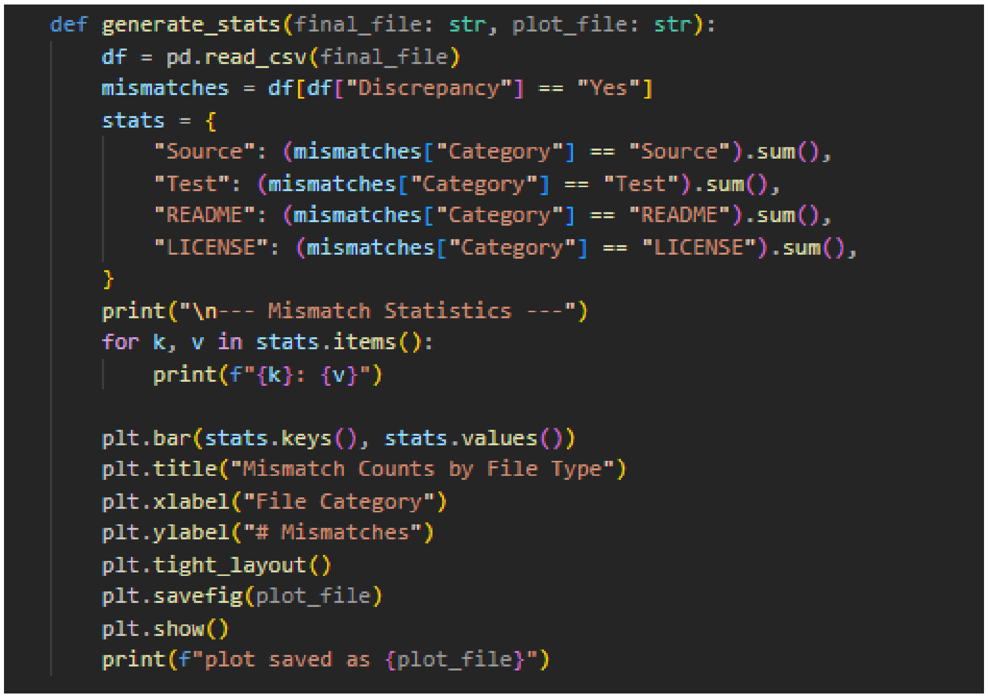


```python
def add_discrepancy(dataset_file: str, final_file: str):
    df = pd.read_csv(dataset_file)
    df["Discrepancy"] = [
        "No" if clean_diff(row["diff_myers"]) == clean_diff(row["diff_hist"]) else "Yes"
        for _, row in df.iterrows()
    ]
    df["Category"] = df.apply(lambda row: categorize_file(row["new_file_path"] or row["old_file_path"]), axis=1)
    df.to_csv(final_file, index=False)
    print(f"final dataset written to {final_file}")
```

(f) Compute statistics & plot

generate_stats(final_file, plot_file):

- Produces counts of mismatches broken down by Category (Source, Test, README, LICENSE).
- Prints statistics and creates a bar chart (mismatch_stats_<tag>.png).
- A simple bar chart makes it easy to see which file types produce the most algorithm differences and supports claims such as "diff algorithms diverge more on source files than on docs" (or vice versa).



```python
def generate_stats(final_file: str, plot_file: str):
    df = pd.read_csv(final_file)
    mismatches = df[df["Discrepancy"] == "Yes"]
    stats = {
        "Source": (mismatches["Category"] == "Source").sum(),
        "Test": (mismatches["Category"] == "Test").sum(),
        "README": (mismatches["Category"] == "README").sum(),
        "LICENSE": (mismatches["Category"] == "LICENSE").sum(),
    }
    print("\n--- Mismatch Statistics ---")
    for k, v in stats.items():
        print(f"{k}: {v}")

    plt.bar(stats.keys(), stats.values())
    plt.title("Mismatch Counts by File Type")
    plt.xlabel("File Category")
    plt.ylabel("# Mismatches")
    plt.tight_layout()
    plt.savefig(plot_file)
    plt.show()
    print(f"plot saved as {plot_file}")
```

(g) Algorithm comparison (conceptual)

algorithm_comparison(final_file) (skeleton in code) — I printed a placeholder text if no mismatches exist.

How to decide which algorithm "performs better" (if asked to automate)

A fair automatic comparison needs a clear definition of "better," depending on the use case. Here are principled approaches I would follow:

A. Ground-truth / human-labeled evaluation

- Sample a stratified random subset of mismatched files and ask human annotators to judge which diff better aligns with the intended change (better hunk boundaries, clearer moved/renamed lines).
- Compute agreement rates (algorithm A wins, algorithm B wins, tie).

B. Downstream task performance

- Use diffs to extract code patches and feed them to a downstream task (e.g., patch application, automated program repair, or test reruns). Choose the algorithm that yields patches that apply cleanly / produce fewer test regressions.

C. Structural alignment score

- Convert diffs to AST-level edit scripts (where possible) and compute how well each diff maps to actual AST changes. Prefer an algorithm with a smaller edit script or more semantically coherent hunks.

D. Consistency & stability metrics

- Check how often an algorithm produces minimal, focused hunks vs. large, sprawling hunks. For many uses, smaller, targeted hunks are preferable.

E. Runtime/Scalability

- Compare runtime and memory usage on large repos; histogram algorithm might be slower/faster depending on implementation and context.

Automated heuristic (no ground truth)

- If we must pick an algorithm automatically without ground truth, we can:
  - Favor the algorithm that produces fewer *spurious* hunks on simple, known edits (benchmarks).
  - For code files: prefer the algorithm that produces hunks corresponding to token-level changes (higher token overlap in hunks).
  - For non-code files: prefer the algorithm that better preserves documentation blocks (e.g., README changes should be contiguous).

However, the robust approach is a **human-validated sampling** + a downstream task evaluation.

Error Handling

- **Git binary not found / clone fails**
  - Missing git installation or network restrictions prevented automatic cloning. Verified git installation, ensured network access, and sometimes manually cloned the repository, then pointed local_path in the code to the cloned folder.
- **False positive mismatches due to metadata**
  - Diff outputs contained metadata lines (headers, hunk markers) that varied across algorithms, causing mismatches even when actual code changes matched. Implemented a clean_diff() function to strip headers (diff --git, index, @@ markers) and whitespace-only differences before comparison.
- **Remote-only PyDriller access failed**
  - Initially, I attempted to run *pydriller* directly on the remote GitHub URL, but this caused failures in diff extraction. Switched to **local cloning**, which ensured full commit history and allowed consistent use of git diff with different algorithms.

RESULTS AND ANALYSIS

After running *PyDriller* with **Myers** and **Histogram** diff algorithms on the three selected repositories, I compared the outputs for discrepancies. The consolidated datasets included commit metadata, file paths, commit messages, and diff outputs for both algorithms. I then tagged each file-level change with a **Discrepancy** label ("Yes" or "No") depending on whether the two diff outputs matched after cleaning.

Repository 1 – Copyparty

- **Discrepancy Counts**:
  - Yes = 272
  - No = 10,367
- **Mismatch Breakdown**:
  - Source files: 221
  - Test files: 3
  - README files: 8
  - LICENSE files: 0
- **Observation**: Most mismatches occurred in source files, with few in test or documentation. README mismatches were mainly due to formatting and text changes, where the algorithms handled line breaks differently.

Repository 2 – DeepFace

- **Discrepancy Counts**:
  - Yes = 131
  - No = 3,071
- **Mismatch Breakdown**:
  - Source files: 98
  - Test files: 17
  - README files: 10
  - LICENSE files: 0
- **Observation**: Compared to repo1, DeepFace had far fewer mismatches in absolute terms. This is likely due to its smaller size and more structured commit history. The mismatches primarily came from Python source files, suggesting algorithm sensitivity to indentation and small code-block movements.

Repository 3 – Python-Telegram-Bot

- **Discrepancy Counts**:
  - Yes = 558
  - No = 20,892
- **Mismatch Breakdown**:
  - Source files: 317
  - Test files: 215
  - README files: 13
  - LICENSE files: 1
- **Observation**: This project had the most significant mismatches overall, particularly in source and test files. The high test-file discrepancy suggests that frequent structural edits in unit tests (e.g., reordering, indentation) produce differences that Myers and Histogram handle differently.

Comparative Insights

- Across all three projects, **source files consistently accounted for most mismatches**.
- **README mismatches** appeared in all repositories, reflecting the algorithms' sensitivity to whitespace and formatting changes in text files.
- Test file mismatches were exceptionally high in repo3, showing that test code may be more prone to diff algorithm variation than production code.

- Overall, the percentage of mismatches relative to total commits remained **low (<5%)**, but non-trivial in larger projects, highlighting that algorithm choice can meaningfully affect downstream analyses.

DISCUSSION AND CONCLUSION

Challenges

- **Diff sensitivity**: Myers and Histogram differ in aligning moved or slightly modified lines. This produced mismatches even when human observers would consider the changes equivalent.
- **Documentation files:** Non-code artifacts like README files consistently produced discrepancies due to whitespace and formatting handling, underlining that diff algorithms are not optimized for natural language text.
- **Scaling to large projects:** Extracting diffs for tens of thousands of commits was computationally expensive. Careful project selection (1000–5000 commits) kept the runtime manageable.

Reflections

- Source code is more stable across algorithms than test and documentation files, which are more sensitive to formatting.
- Test code mismatches matter: since tests are supposed to be small, modular, and frequently edited, discrepancies in diff algorithms can mislead tools that rely on diff outputs for fault localization or coverage analyses.
- Choice of algorithm matters for downstream tasks: If one were to build datasets for bug-fix mining or automated program repair, even minor discrepancies in diffs could influence the labeling of fixes.

Lessons Learned

- Algorithmic differences matter most in non-code and test artifacts, where line order, whitespace, and reordering are standard.
- Cleaning diffs is essential before comparing, otherwise false mismatches due to metadata dominate.
- Pydriller with local repositories provided a stable pipeline; remote-only analysis often broke or missed diffs.
- Histogram may better capture structural movements, while Myers tends to minimize textual differences.

Conclusion

The lab demonstrated that while diff algorithms usually agree, a non-trivial portion of commits (~1–5%) produce divergent results. Most mismatches are concentrated in source and test files, with some spillover into documentation. These differences highlight that algorithm selection is not neutral — it directly affects the downstream analysis of software evolution.

If I were to automatically decide which algorithm "performs better," I would benchmark their diffs against human-annotated gold standards or use task-specific metrics (e.g., which algorithm's diff leads to more accurate bug localization). For now, my findings emphasize the need for careful algorithm choice and awareness of its limitations when mining large-scale repositories.

REFERENCES

[8] https://github.com/yusufsn/DifferentDiffAlgorithms

[9] CS 202 STT Lecture 4 Slides

GITHUB REPO LINK OF CODES

https://github.com/Revathi-katta/STT_Lab_Assessment_1.git