ASSESSMENT 1

Software Tools and Techniques for CSE

Siddharth Rajandekar

23110310

TABLE OF CONTENTS

LABORATORY SESSION 1

INTRODUCTION

The aim of the first lab session was to gain an understanding and get hands-on experience in using Version Control Systems, principally Git and GitHub, as well as understand the importance of Git-based Version Control systems in the domain of Software Development. I could understand the key concepts of the system, such as commit, push, pull, etc. After this, I also worked on pylint workflow based GitHub Actions in order to improve the code quality. This discussion leads us to the topic of the tools that were used for this lab session.

TOOLS

The most important tool that I used for this laboratory session was Git. I used the Git Bash terminal to execute the Git commands and the web-based interface of GitHub to create the repository and run the workflows. The pylint workflow based GitHub Actions were also used for statically analysing and improving code quality.

SETUP

In order to setup the Git, I first had to provide my credentials to the Git Bash Terminal. I used the git commands as shown below to enter my username as well as email and verified them as well.

```
PS C:\Users\HP\Desktop\CS_202> git config --global user.name "Siddharth"
PS C:\Users\HP\Desktop\CS_202> git config --global user.email "siddharth.rajandekar@iitgn.ac.in"
PS C:\Users\HP\Desktop\CS_202> git config --global color.ui auto
PS C:\Users\HP\Desktop\CS_202> 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=master
user.email=siddharth.rajandekar@iitgn.ac.in
user.name=Siddharth
color.ui=auto
(END)
pull.rebase=false
credential.helper=manager
credential.https://dev.azure.com.usehttppath=true
init.defaultbranch=master
user.email=siddharth.rajandekar@iitgn.ac.in
user.name=Siddharth
color.ui=auto
~
~
```

Once the setup was done, I could move on to the main assignment tasks.

METHODOLOGY AND EXECUTION

The first step was to create an empty directory for this task, using the command '*mkdir*' to make a new directory.

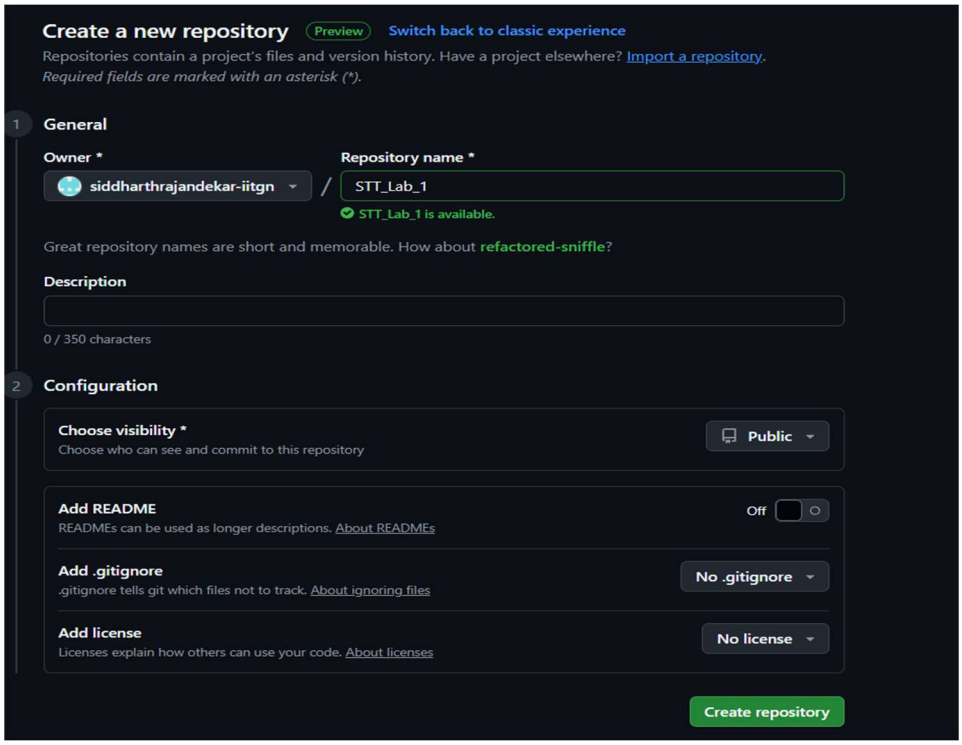
I then used the '*git init'* to initialize the directory as a git repository.

```
PS C:\Users\HP\Desktop\CS_202\Lab_1> mkdir "Any_Name"


    Directory: C:\Users\HP\Desktop\CS_202\Lab_1


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        04-08-2025     14:23                Any_Name


PS C:\Users\HP\Desktop\CS_202\Lab_1> cd Any_Name
PS C:\Users\HP\Desktop\CS_202\Lab_1\Any_Name> git init
Initialized empty Git repository in C:/Users/HP/Desktop/CS_202/Lab_1/Any_Name/.git/
```

After that, I started creating new files, such as README.md, and then added and committed the files. I used the '*git add*' command in order to stage the files I had changed before committing, and the *'git commit -m'* command along with its message to enable the changes to be committed to the repository which signifies the changes that Imade to the repository. Using the *'git log'* command, I could also see the history of commits and changes that were logged using git.

```
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git add .\REAMDE.md
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git commit -m "Added README.md"
[master (root-commit) 14bdf85] Added README.md
 1 file changed, 1 insertion(+)
 create mode 100644 REAMDE.md
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git log
commit 14bdf853f01dd9dc1a817a66a74c9efdf565d451 (HEAD -> master)
Author: Siddharth <siddharth.rajandekar@iitgn.ac.in>
Date:   Mon Aug 4 14:36:24 2025 +0530

    Added README.md
```

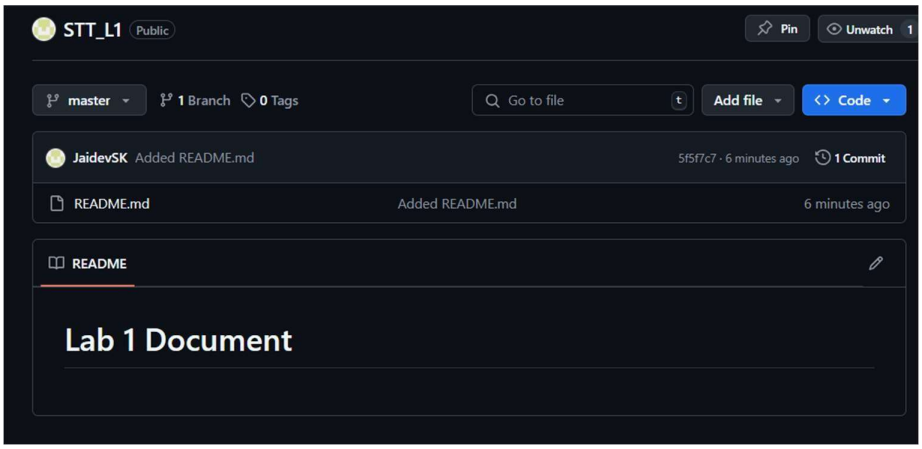
In the next step, I created a new GitHub repository as shown below..



I then had to remotely add the local repository to the GitHub repository. I achieved this using the '*git remote add'* command. Then, I changed the name of my branch from master to main using the 'git branch -M main' command. Finally, I pushed the changes to the upstream (GitHub based) repository using the 'git push -u' command.

```
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git remote add origin https://github.com/siddharthrajandekar/STT_Lab_1.git
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git branch -m Main
fatal: a branch named 'Main' already exists
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git branch -M main
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git push -u origin main
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 12 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/siddharthrajandekar/STT_Lab_1.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

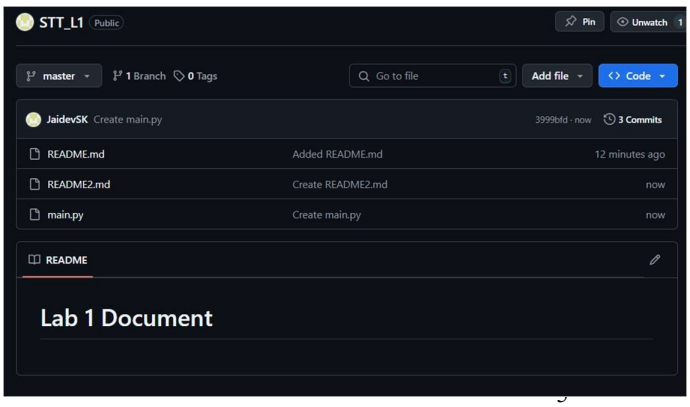
I had initialised an empty repository on GitHub but the Local Repository contained README.md. As soon as I pushed it, the changes were visible on the online repository as well.



In the next step, I cloned the repository using the '*git clone*' command.

```
PS C:\Users\HP\Desktop\CS_202\Lab_1> git clone https://github.com/SET-IITGN/cs202-demo.git
Cloning into 'cs202-demo'...
remote: Enumerating objects: 69, done.
remote: Counting objects: 100% (69/69), done.
remote: Compressing objects: 100% (59/59), done.
Receiving objects:  73% (51/69)used 5 (delta 1), pack-reused 0 (from 0)
Receiving objects: 100% (69/69), 24.62 KiB | 4.92 MiB/s, done.
Resolving deltas: 100% (24/24), done.
```

Then, I updated the online repository and added new files named README2.md and main.py. These changes were made in the GitHub Online Repository and they were not reflected in the local repository.



Now, I had to pull these changes from the online repository to the offline one. So, I used the command '*git pull'*.

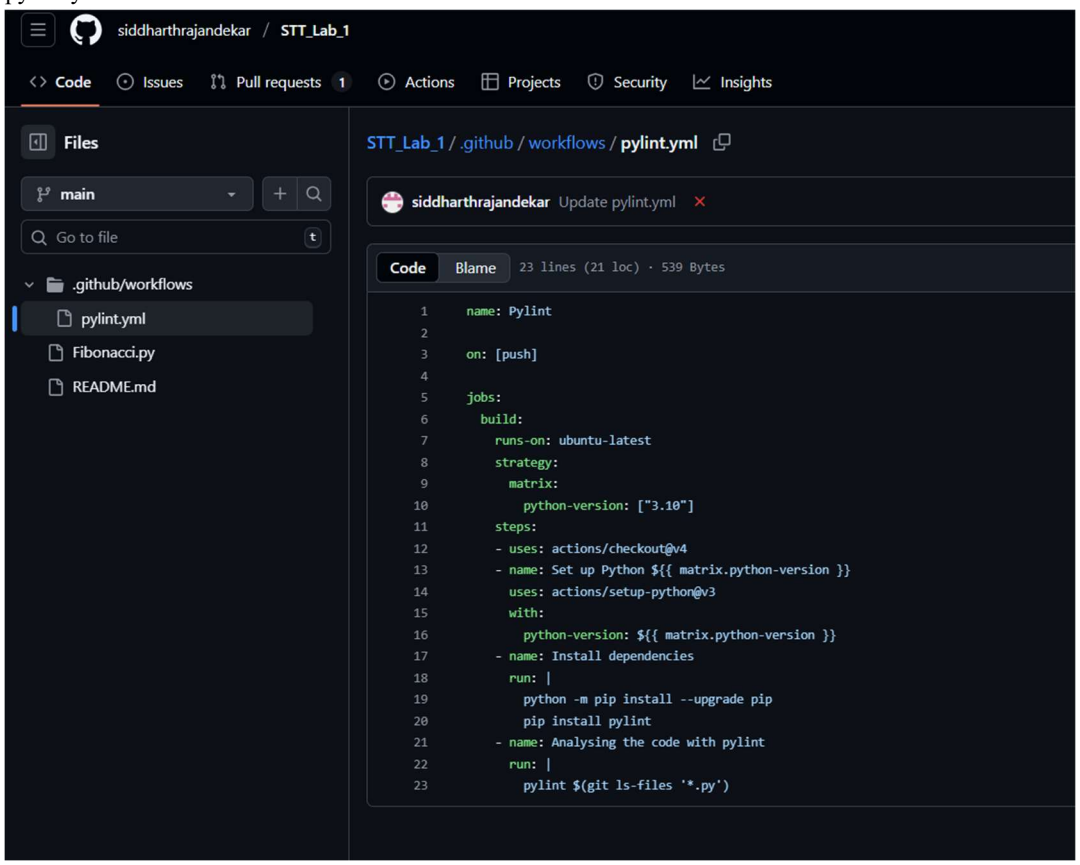
```
PS C:\Users\HP\Desktop\CS_202\Lab_1\STT_Lab_1> git pull
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (1/1), done.
remote: Total 2 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (2/2), 888 bytes | 98.00 KiB/s, done.
From https://github.com/siddharthrajandekar/STT_Lab_1
   4c2d1dc..f99fbbc  main       -> origin/main
Updating 4c2d1dc..f99fbbc
Fast-forward
 REAMDE.md | 1 -
 1 file changed, 1 deletion(-)
 delete mode 100644 REAMDE.md
```

After performing the '*git pull*' operation, the changes are visible in the local repository as well.

The next task was based on pylint. I created a python file named fibonacci.py as given below:

```python
#This function calculates the nth fibonacci number
# The nth fibonacci number Fn = Fn-1 + Fn-2
# F1 = 1 and F2 = 1
def fibonacci(n):
    if n <= 0:
        return -1
    if n == 1 or n == 2:
        return 1
    first = 1
    second = 1
    ans = 0
    for _ in range(n-2):
        ans = first + second
        first, second = ans, first

    return ans

def main():
    n = int(input("Enter an integer: "))
    fibo = fibonacci(n)
    if fibo == -1:
        print("The integer should be greater than 0")
    else:
        print(fibo)

if __name__ == "__main__":
    main()
```

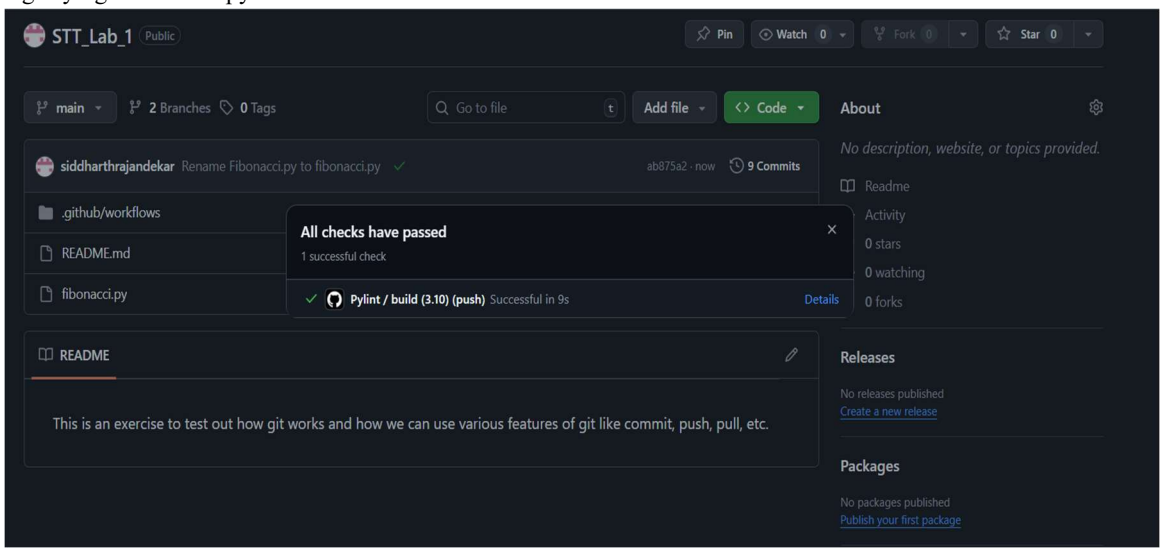
In order to establish a pylint workflow, I initiated a GitHub action corresponding to pylint. A corresponding pylint.yml file was created as shown below.



The errors pointed out by the workflow are provided below:

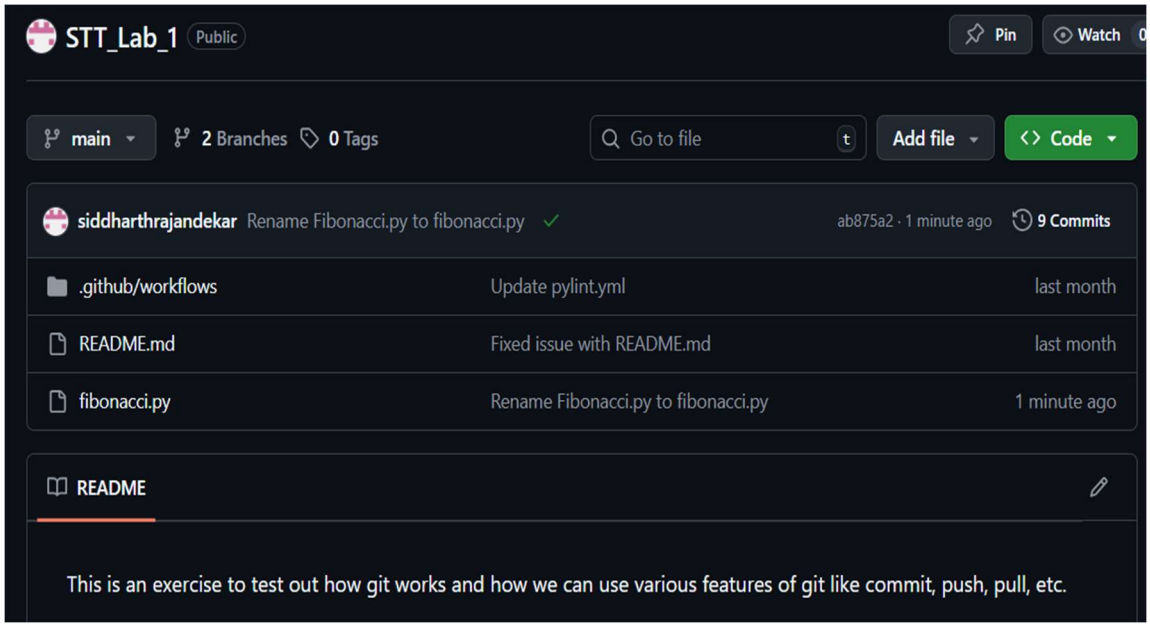
The corresponding workflow also showed a ✗ mark to indicate that the workflow was ending with a pylint error. I then explored the range of errors and made corresponding changes.

Once all the corresponding changes were made, the workflow ran again and this time, it showed a success message signifying that all the pylint tests were successful.



RESULTS AND ANALYSIS

Analysing the applications of git, I could use the git commands directly from the command line instead of going to the graphical interface on internet. The results of the corresponding steps and commands that I used for the git section are shown immediately next to the commands in the methodology section. Coming to the discussion of pylint, it was an interesting application and despite writing syntactically correct code, it was still showing some pylint workflow errors due to inconsistency in the code writing style. Once the code was made of a consistent style, the corresponding workflow was also returning the success message and the ✓ was displayed indicating success in the workflow.



DISCUSSION AND CONCLUSION

On the whole, this laboratory session was very important for me as I got a hands on experience in using command line interface based git commands, right from making a new empty repository to adding, committing, cloning and remotely connecting to an online repository along with pulling and pushing the changes to and from the remote repository. I initially faced some difficulties in configuring the git because of its new rules of mobile based login, but that was not a very big obstruction. The pylint was also an interesting tool aimed at improving the code quality of the python code that we have written. However, I think that there can be some more improvements in the pylint tool such as inclusion of more detailed error messages. We can also move a step ahead and make the pylint tool autocorrect the code, using LLM based models. These can be some research topics that I could find out from this laboratory session.

REFERENCES

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

LABORATORY SESSION 2

INTRODUCTION

The second laboratory session, was aimed at introducing us to bug mining in software repositories. In this lab, we used pydriller, which is a tool to analyse GitHub repositories. In this laboratory session, we performed many hands on activities in analysing the bug fixing commits in Open Source Software Repositories available on GitHub. We identified the bug fixing commits from the repository and developed a framework to generate an appropriate commit message using an LLM and rectify it if needed. The details of the analysis and results are provided in the subsequent sections of the report.

TOOLS

I used three main tools in this analysis, the most important of them being pydriller, which is an open source python library used to analyse GitHub repositories. I also used Commit Predictor T5 which is a fined tuned version of T5 LLM model to generate an appropriate commit message for the bug fixing commit. It is available on HuggingFace. I also used the SEART Search Engine for GitHub by which I could search the relevant repositories based on custom filters.

SETUP

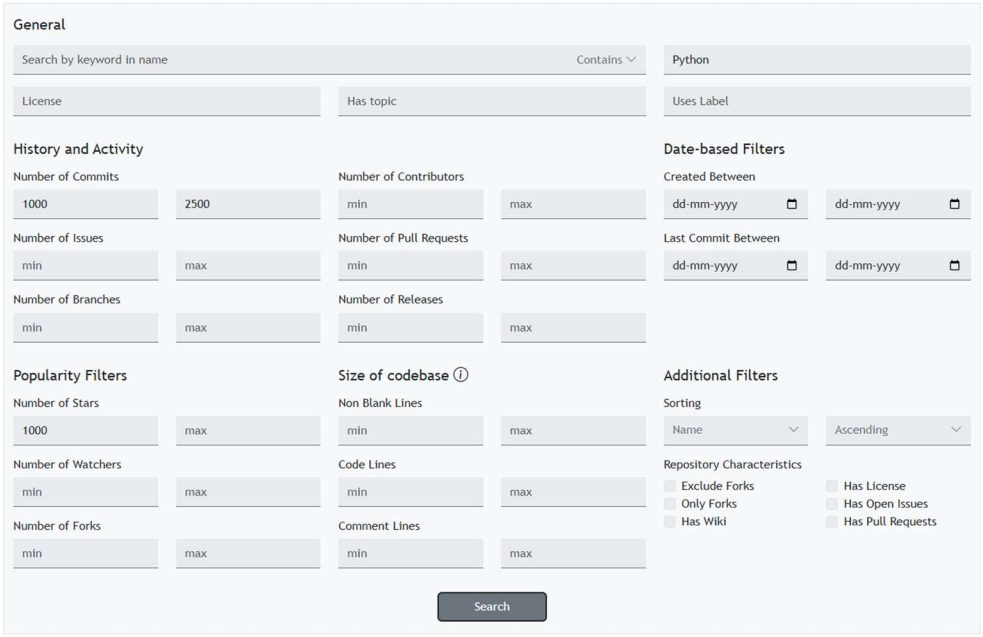
The first step was to set up pydriller. This was done using the *'pip install pydriller'* command. The corresponding screenshot is also attached below:

```
PS C:\Users\HP\Desktop\CS_202\Labs_1-4> pip install pydriller
Defaulting to user installation because normal site-packages is not writeable
Requirement already satisfied: pydriller in c:\users\hp\appdata\roaming\python\python311\site-packages (2.8)
Requirement already satisfied: gitpython in c:\users\hp\appdata\roaming\python\python311\site-packages (from pydriller) (3.1.44)
Requirement already satisfied: pytz in c:\users\hp\appdata\roaming\python\python311\site-packages (from pydriller) (2025.1)
Requirement already satisfied: types-pytz in c:\users\hp\appdata\roaming\python\python311\site-packages (from pydriller) (2025.2.0.20250809)
Requirement already satisfied: lizard==1.17.10 in c:\users\hp\appdata\roaming\python\python311\site-packages (from pydriller) (1.17.10)
Requirement already satisfied: gitdb<5,>=4.0.1 in c:\users\hp\appdata\roaming\python\python311\site-packages (from gitpython->pydriller) (4.0.12)
Requirement already satisfied: smmap<6,>=3.0.1 in c:\users\hp\appdata\roaming\python\python311\site-packages (from gitdb<5,>=4.0.1->gitpython->pydril
ler) (5.0.2)
```

This creates the initial basis for us to go to the methodology stage.

METHODOLOGY AND EXECUTION

The first task in this was to search for a medium to large sized repository based on our own selection criteria. I used a tool called SEART Search Engine for GitHub in order to filter the repositories. The corresponding screenshot is attached below:



The criteria I set for this were:

- The number of commits should be between 1000 and 2500.
- The minimum number of stars should be 1000.
- The major language of the repository should be Python.

Based on these criteria, I found a repository called textual paint. Link to the repository is in the References section. The next task was to "mine" this repository using pydriller to identify bug fixing commits. But before that, we first have to define what a "bug fixing commit' is. Using pydriller we get access to the commit message from the developer. Then, we check if the commit message contains one of the keywords in the screenshot. If it contains any of the keywords, we classify it as a bug fixing commit, else we move on to the next commit.

The corresponding screenshot is attached below.

```
keywords = ["fixed", "bug", "fixes ", "fix ", " fix", "fixed", " fixes", "crash", "solves", "resolves",
            "resolves "," issue", "issue", "regression", "fall back", "assertion", "coverity", "reproducible",
             "stack-wanted", "steps-wanted", "testcase", "failur", "fail", "npe ", " npe", "except", "broken",
              "differential testing", "error", "hang hang", "test fix", "steps to reproduce", "crash",
               "assertion", "failure", "leak", "stack trace", "heap overflow", "freez", "problem", "problem",
                "overflow", "overflow ", "avoid ", " avoid", "workaround ", "workaround", "break", "break", "stop", "stop"]
```

The repository was mined using pydriller in the following way:

```
for commit in Repository("https://github.com/1j01/textual-paint").traverse_commits():
    message = commit.msg
    if not any(keyword in message for keyword in keywords):
        continue

    modified_files = commit.modified_files
```

We loop through all the commits in the repository, check if the commit is a bug fixing commit. If yes, then we store the following information about the commit in a csv file: the hash of the commit, the commit message, hashes of

parent commits, is the commit a merge commit and the list of the modified files in the commit. Then for each modified file in the commit, we generate a commit message for the file by passing the git diff of the file to our Commit Predictor T5 LLM. We can get the LLM output using the transformers library in the following way:

```
#Loading the LLM model
tokenizer = AutoTokenizer.from_pretrained("mamiksik/CommitPredictorT5")
model = AutoModelForSeq2SeqLM.from_pretrained("mamiksik/CommitPredictorT5")
```

```
# LLM Inference
input_ids = tokenizer(diff, return_tensors="pt", max_length=max_input_len).input_ids
outputs = model.generate(input_ids, max_length=max_output_len)
LLM_output = tokenizer.decode(outputs[0], skip_special_tokens= True)
```

Sometimes developers may span changes to multiple files in a single commit and give a very generic commit message. But this does not give us proper information regarding the changes made to individual files. Hence, we need an LLM to generate a commit message per file. But, sometimes even the LLM might fail to generate a proper commit message, hence we need to further rectify the message generated by the LLM.

To get the proper context regarding the commit, we are considering both the message generated by the LLM as well as the original message by the developer. We iterate through the developer message line by line and if the line contains a generic phrase (defined below), we do not include it because it does not give us much information.

Then, we check the cosine similarity between the developer line and the lines generated by the llm. If the similarity is less than 0.8, only then we keep the developer line because otherwise we are getting that information from the llm message and the developer line is redundant.

After this, we iterate through the LLM lines and then add them to our combined message if they do not contain any generic phrase. Next, we normalize the tense of our message by replacing all the verbs with their normalized forms (defined below). This will ensure that our message stays in the same tense.

```
GENERIC_PHRASES = [
    "fix bug", "bug fix", "error fixed", "issue resolved"
]
```

```
replacements = {
    "fixed": "fix", "fixes": "fix", "fixing": "fix",
    "added": "add", "adding": "add",
    "removed": "remove", "removing": "remove",
    "updated": "update", "updating": "update",
    "changed": "change", "changing": "change",
    "implemented": "implement", "implementing": "implement",
    "refactored": "refactor", "refactoring": "refactor"
}
```

After this, we extract the first action verb (defined below) from our message and move it to the start of the message because good commit messages usually start with an action verb like add, update, fix, etc. This completes the rectification process.

```
ACTION_VERBS = [
    "fix", "add", "remove", "update", "refactor", "implement", "change"
]
```

In the end we store the following data regarding the modified files in another csv file: hash of the commit, commit message, filename, source code before the commit, source code after the commit, git diff of the file, the LLM inference and the rectified message.

Next, we need to answer whether the commit message from the developer, the message generated by the LLM and the rectified message are precise. We define the precision criteria as follows: the length of the message should be between 3 and 20 words (not too long, not too short). The first word of the message must be an action verb (defined above), the message must contain atleast one noun (could be the name of the file or in general gives specific information). Lastly, the message should not contain any generic phrases (defined above).

The code for the rectifier is as follows:

```
def rectify_commit_message(dev_msg, llm_msg):

    dev_lines = [line.strip() for line in dev_msg.split("\n") if line.strip()]
    llm_lines = [line.strip() for line in llm_msg.split("\n") if line.strip()]

    # Combine lines that are unique or more informative
    combined_lines = []
    for d_line in dev_lines:
        # Skip if line is generic
        if any(phrase in d_line.lower() for phrase in GENERIC_PHRASES):
            continue

        # Check similarity to any LLM line
        if llm_lines:
            sims = util.cos_sim(similarity_model.encode([d_line]), similarity_model.encode(llm_lines))
            if sims.max() < 0.8:  # Keep dev line if not redundant
                combined_lines.append(d_line)
        else:
            combined_lines.append(d_line)

    # Add LLM lines not already in combined
    for l_line in llm_lines:
        if l_line not in combined_lines and not any(phrase in l_line.lower() for phrase in GENERIC_PHRASES):
            combined_lines.append(l_line)

    msg = " ".join(combined_lines)

    # Normalize verbs
    replacements = {
        "fixed": "fix", "fixes": "fix", "fixing": "fix",
        "added": "add", "adding": "add",
        "removed": "remove", "removing": "remove",
        "updated": "update", "updating": "update",
        "changed": "change", "changing": "change",
        "implemented": "implement", "implementing": "implement",
```

```
replacements = {
    "fixed": "fix", "fixes": "fix", "fixing": "fix",
    "added": "add", "adding": "add",
    "removed": "remove", "removing": "remove",
    "updated": "update", "updating": "update",
    "changed": "change", "changing": "change",
    "implemented": "implement", "implementing": "implement",
    "refactored": "refactor", "refactoring": "refactor"
}
msg = msg.lower()
msg = re.sub(r"^(this commit|commit)\s*", "", msg)
for k, v in replacements.items():
    msg = re.sub(rf"\b{k}\b", v, msg)

# Extract first action verb
tokens = nltk.word_tokenize(msg)
pos_tags = nltk.pos_tag(tokens)
verb_found = False
for i, (word, tag) in enumerate(pos_tags):
    if tag.startswith("VB") and word in IMPERATIVE_VERBS:
        # Move verb to start if not already
        if i > 0:
            tokens.insert(0, tokens.pop(i))
        verb_found = True
        break
if not verb_found and tokens:
    tokens.insert(0, "update")

msg = " ".join(tokens)

summary = msg.split("\n")[0][:50].rstrip()
return summary
```

The code for checking is the message precise is as follows:

```python
def is_precise_commit_message(msg):
    msg = msg.strip()

    # 1. Too short or too long
    words = msg.split()
    if len(words) < 3 or len(words) > 20:
        return False

    # 2. Action verb at start
    first_word = words[0].lower()
    imperative_verbs = ["fix", "add", "remove", "update", "refactor", "implement", "change"]
    if first_word not in imperative_verbs:
        return False

    # 3. Must contain at least one noun (component/location)
    tokens = word_tokenize(msg)
    pos_tags = pos_tag(tokens)
    nouns = [w for w, t in pos_tags if t.startswith("NN")]
    if len(nouns) == 0:
        return False

    # 4. Reject generic messages
    generic = ["fix bug", "bug fix", "error fixed", "issue resolved"]
    if msg.lower() in generic:
        return False

    return True
```

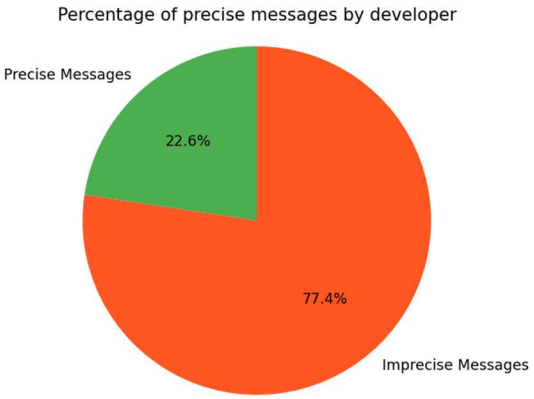
RESULTS AND ANALYSIS

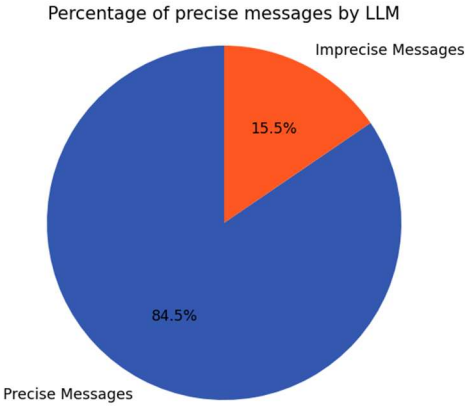
Once we ran our code we got two csv files: “Bug Fix Commit.csv” and “Diff Extraction.csv”. Below are two screenshots from both the files in the respective order

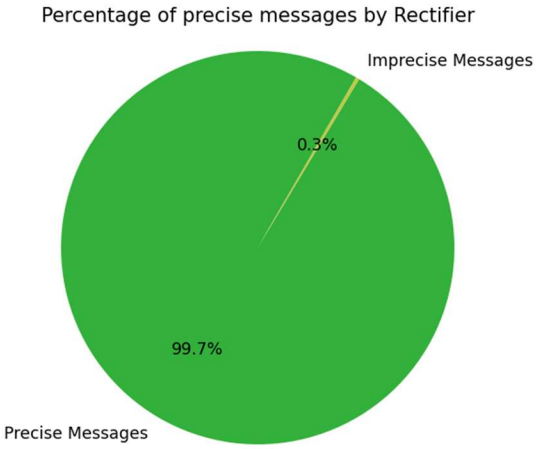
Hash	Message	Hash_Parent	Is_Mer	Modified_Files
b64869eae4c9b5962b0e71d3b060709c20931214	Let buttons fall back to the original color when deselected	['3f8a652fe2d61e15d29ba9f5b76758c3b29533f7']	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658D13EA10>, <pydriller.domain.commit.ModifiedFile object at 0x000002658C5C1E50>]
ad111e5d6c044ffbd562c5c35019f82e934d7620	Refactor: extract tool handling from Canvas I wanted to avoid duplicating tool-related state between PaintApp and Canvas and prepare for adding different tools with more state and which will want to live in a separate file. This makes it slower when running with `textual run --dev paint.py`; when running with `python3 paint.py` it's fine. When running in dev mode with `textual console` devtool connected it's extremely much slower. But if it was faster you'd have more messages to scroll through ha. So it's a tradeoff.*	['338d438470a7910a464940ea3a7b38f2a25d3639']	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658C655710>]
6d17f085f5a655ef42bce2569ba3	Debug: show regions when undoing/redoing (currently the whole	['4fa7dda9a070491b3c1529f89b5e5	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658D11D410>]
0aaa9db8395aff4b4df8f6544ab8	Make region update debug a flag	['15db3f453818e171ba68be449291	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658C5C2890>]
6d5430ad2eae8be87728ff582d3	Fix error due to action regions exceeding canvas bounds	['0aaa9db8395aff4b4df8f6544ab85	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658DBE5350>]
3e84ae7e17efd7cdf7d368f0c4a7	Disable region update debug	['6d5430ad2eae8be87728ff582d396	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658D62FE90>]
303d6942f4badb851cc755a0c9d	Switch out fill tool icon to avoid text row offset artifacts	['5b1c16ddc35ec8f76da1a46aad59d	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658C0DC690>]
3652b2bd8e1c905b4bb941a4280bf5e164313e9a	Add flood fill support * Implemented the Fill With Color tool using the algorithm described as "combined-scan-and-fill span filler" on Wikipedia. * I added handling for the affected region being None which turned out more complicated than I would like... Some cases may be able to be simplified or removed. * Also I moved	['3dbbbc1c5fe5c923a37eb6404ff64ca2c640636a']	FALSE	[<pydriller.domain.commit.ModifiedFile object at 0x000002658D62CF10>]
	Add textual screenshot to readme This took a lot of trial and error to get this working. - First I had to figure out how to copy the terminal output as HTML. I had to configure keybindings for Select All and Copy As HTML in the Ubuntu terminal. - GitHub doesn't support line-height in markdown so I came up with using an SVG with <foreignObject> to contain the screenshot HTML. - GitHub			

Hash	Message	Filename	Source_Code_Old	Source_Code_New	Diff	LLM_Inference	Rectified_Message
b64869eae4c9b5962b0e71d3b060709c20931214	Let buttons fall back to the original color when deselected	paint.css	Screen { overflow: auto; } #paint { layout: grid; grid-size: 2; grid-gutter: 1 2; grid-columns: 20 1fr; grid-rows: 1fr; margin: 0	Screen { overflow: auto; } #paint { layout: grid; grid-size: 2; grid-gutter: 1 2; grid-columns: 20 1fr; grid-rows: 1fr; margin: 0 0; min-	@@ -26 6 +26 10 @@ Screen { height: 100%; } +#tools_box Button.selected { + background: #aaa; +} + Button { width: 100%; height: 100%;	add button background to example	add let buttons fall back to the original color wh
b64869eae4c9b5962b0e71d3b060709c20931214	Let buttons fall back to the original color when deselected	paint.py	from enum import Enum from textual import events from textual.app import App ComposeResult from textual.containers import Container from	from enum import Enum from textual import events from textual.app import App ComposeResult from textual.containers import Container from	@@ -115 9 +115 8 @@ class PaintApp(App): def watch_selected_tool(self old_selected_tool: Tool selected_tool: Tool) -> None: """Called when selected_tool	update paintapp.py	update let buttons fall back to the original color
ad111e5d6c044ffbd562c5c35019f82e934d7620	Refactor: extract tool handling from Canvas I wanted to avoid duplicating tool-related state between PaintApp and Canvas and prepare for adding different tools with more state and which will want to live in a separate file. This makes it slower when running with `textual run --dev paint.py`; when running with	paint.py	from enum import Enum from rich.segment import Segment from rich.style import Style from textual import events from textual.app import App ComposeResult from textual.containers import	from enum import Enum from rich.segment import Segment from rich.style import Style from textual import events from textual.message import Message MessageTarget from textual.app import App ComposeResult from	@@ -3 6 +3 7 @@ from enum import Enum from rich.segment import Segment from rich.style import Style from textual import events +from textual.message import Message MessageTarget from textual.app import App ComposeResult from	add missing docstring	add refactor : extract tool handling from canvas i
6d17f085f5a655ef42bce2569ba3f0b1b1c497e9	Debug: show regions when undoing/redoing (currently the whole canvas)	paint.py	from enum import Enum from rich.segment import Segment from rich.style import Style from textual import events from textual.message import	from enum import Enum from random import randint from rich.segment import Segment from rich.style import Style from textual import events from textual.message import	@@ -1 5 +1 5 @@ from enum import Enum - +from random import randint from rich.segment import Segment from rich.style import Style from textual import events @@ -166 6 +166 7 @@ class	add ansiart example for the color selection	add debug : show regions when undoing/redoing (cu

The percentage of precise messages by developer, by the LLM and by the rectifier are as follows:







From these plots, the hit rates are as follows:

- **Developer Hit Rate:** 0.224
- **LLM Hit Rate:** 0.845
- **Rectifier Hit Rate**: 0.997

Based on these results, we can conclude that developers do not generate a precise commit message with only 22% of the messages being precise. This is expected behaviour, as in many cases developers might write a very generic message for a commit that spans changes over various files. On the other hand, the LLM is able to generate a mostly precise commit message with a hit rate of 84.5%, but even it fails for some cases. Lastly, our rectifier is able to rectify the message in maximum cases with a hit rate of near 100%

DISCUSSION AND CONCLUSION

Initially I had considered using another stronger LLM for rectifying the message generated by our Commit Predictor T5 model. But it can be argued that even that LLM is not good enough for generating a precise message. It was also taking a lot of time to run the code since we were using so many heavy LLM models and at some point the RAM on my laptop also fell short. Hence, I had to drop the idea of using an LLM as a rectifier. Thus, I made my custom criteria for the rectifier that uses the nltk library and natural language processing which is much lighter than using an LLM.

But after analysing the results in the csv, I found out that in some cases the rectifier was giving nonsense message that is not even grammatically correct. But our plots show that the hit rate of the rectifier is almost 100%. This is clearly a contradiction. This raises the point that either our rectifier and our precision criteria need further analysis. But due to time constraints, I was not able to analyse them further and hence the results reported here might not be 100% reliable.

But nonetheless, I learnt a lot from this assignment about mining github repositories and extracting useful information from them. Making of a rectifier was also an enlightening process as it forced me to use my creativity for creating a good rectifier.

REFERENCES

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

[2] https://seart-ghs.si.usi.ch/

[3] https://github.com/1j01/textual-paint

[4] https://huggingface.co/mamiksik/CommitPredictorT5

LABORATORY SESSION 3

INTRODUCTION

This lab extends the Lab 2 dataset of file-level bug-fix commits by integrating additional code quality metrics into the analysis. The emphasis is on structural measures such as Maintainability Index, Cyclomatic Complexity, and Lines of Code, evaluated for each bug-fix both before and after the change using *radon*. These structural metrics are then compared with Semantic and Token-Based Similarity measures to explore how code quality relates to the magnitude of changes

TOOLS

The most important tool used for this laboratory session was radon, which was used to calculate the cyclomatic complexity, the maintainability index and lines of code. We also used CodeBERT Score to calculate the semantic similarity between the old code and new code. Then we used SacreBLEU to calculate the token similarity between the old code and new code. Also, overall we used the pandas library to import data from our lab 2 csv and export our new data to another csv and matplotlib to visualize the results.

SETUP

I used the Visual Studio Code IDE on my own Windows system to perform all the experiments for this lab. I installed radon using the command '*pip install radon'*. I also installed code-bert-score and sacrebleu using the same 'pip install' command. The other libraries required for the data analysis of the obtained csv file such as Pandas and Matplotlib were already installed in my system, so I did not need to download them again.

These are all the import we are doing:

```
import pandas as pd
import csv
from radon.complexity import cc_visit
from radon.metrics import mi_visit
from radon.raw import analyze
import code_bert_score
from sacrebleu.metrics import BLEU
import ast
import matplotlib.pyplot as plt
import os
```

METHODOLOGY AND EXECUTION

Since we already had our 'Diff Extraction' csv ready from our lab 2, the first step for this lab was to simply import that csv using the pandas read_csv function. We also create a new dictionary which would be later converted to our output csv file for lab 3 using pandas.

```
def main():
    df = pd.read_csv(r"C:\Users\Student\Desktop\Siddhath\Diff_Extraction.csv", encoding='latin1')
    output_df_dict = {"MI_Change": [], "CC_Change": [], "LOC_Change": [], "Semantic_Similarity": [], "Token_Similarity": [],
                      "Semantic_Class": [], "Token_Class": [], "Classes_Agree": []}
    n = len(df)
```

The next step is to report the baseline statistics like total number of commits and files, average modified files per commit, etc. We report these metrics using the pandas library's builtin functions and matplotlib for visualization.

Total number of commits, files and average files per commit:

```
# Total number of commits and files
total_commits = df["Hash"].nunique()
total_files = df["Filename"].nunique()

print(f"Total number of commits: {total_commits}")
print(f"Total number of files: {total_files}")

# Average number of modified files per commit
files_per_commit = df.groupby("Hash")["Filename"].nunique()
avg_files_per_commit = files_per_commit.mean()

print(f"Average number of modified files per commit: {avg_files_per_commit:.2f}")
```

Plotting the distribution of fix types from LLM Inference:

```
# Distribution of fix types from LLM Inference
fix_type_counts = df["LLM_Inference"].value_counts()

print("\nDistribution of Fix Types:")
print(fix_type_counts)

plt.figure(figsize=(8,6))
bars = plt.bar(range(len(fix_type_counts)), fix_type_counts.values, color="cornflowerblue", edgecolor="white")

plt.title("Distribution of Fix Types", fontsize=14)
plt.ylabel("Count")
plt.xticks([])  # remove x-axis labels entirely
plt.tight_layout()
plt.show()
```

Most frequently modified file extensions:

```
# Most frequently modified filenames/extensions
# Count filenames
filename_counts = df["Filename"].value_counts().head(10)

print("\nMost Frequently Modified Files:")
print(filename_counts)

# Count extensions
df["Extension"] = df["Filename"].apply(lambda x: os.path.splitext(str(x))[1])
ext_counts = df["Extension"].value_counts().head(10)

print("\nMost Frequently Modified File Extensions:")
print(ext_counts)

# Plot extensions
plt.figure(figsize=(8,5))
ext_counts.plot(kind="bar", color="lightgreen", edgecolor="black")
plt.title("Most Frequently Modified File Extensions")
plt.xlabel("Extension")
plt.ylabel("Count")
plt.xticks(rotation=45, ha="right")
plt.tight_layout()
plt.show()
```

The next step was to find various metrics like cyclomatic complexity, maintainability index using radon. Since radon is only able to analyse python source code, we had to check whether each entry in our csv was a valid python code by parsing it. If it was not a valid python code, we simply leave that entry in the csv empty.

```
for i in range(n):
    old_code = df["Source_Code_Old"].iloc[i]
    new_code = df["Source_Code_New"].iloc[i]
    old_code = str(old_code) if pd.notna(old_code) else ""
    new_code = str(new_code) if pd.notna(new_code) else ""

    if not is_valid_python(old_code) or not is_valid_python(new_code):
        output_df_dict["MI_Change"].append("")
        output_df_dict["CC_Change"].append('')
        output_df_dict["LOC_Change"].append('')
        output_df_dict["Semantic_Similarity"].append('')
        output_df_dict["Token_Similarity"].append('')
        output_df_dict["Semantic_Class"].append('')
        output_df_dict["Token_Class"].append('')
        output_df_dict["Classes_Agree"].append('')
        continue
```

This is the function to check if the code is a valid python code:

```
def is_valid_python(code):
    try:
        ast.parse(code)
        return True
    except:
        return False
```

Next, we find the cyclomatic complexity, maintainability index and lines of code in our old and new source code using radon and add the entries to our dictionary.

```
# Cyclomatic Complexity
cc_before = sum(func.complexity for func in cc_visit(old_code))
cc_after = sum(func.complexity for func in cc_visit(new_code))

cc_change = cc_after - cc_before

# Maintainability Index
mi_old = mi_visit(old_code, multi=False)
mi_new = mi_visit(new_code, multi=False)

mi_change = mi_new - mi_old

# Lines of Code
analyzer_old = analyze(  (variable) new_code: str
analyzer_new = analyze(new_code)
loc_before = analyzer_old[0]
loc_after = analyzer_new[0]
loc_change = loc_after - loc_before
```

After this, we calculate the semantic similarity between the old code and new code using CodeBERT score and token similarity using SacreBLEU and add these entries to our dictionary.

```
if not old_code or not new_code:
    token_similarity = 0
    semantic_similarity = 0

else:
    # Semantic Similarity
    reference = [[old_code]]
    hypothesis = [new_code]
    bert_score = code_bert_score.score(cands=hypothesis, refs=reference, lang="python")
    semantic_similarity = bert_score[2].item()

    # Token Similarity
    bleu = BLEU()
    token_similarity = bleu.corpus_score(hypothesis, reference).score / 100

output_df_dict["MI_Change"].append(mi_change)
output_df_dict["CC_Change"].append(cc_change)
output_df_dict["LOC_Change"].append(loc_change)
output_df_dict["Semantic_Similarity"].append(semantic_similarity)
output_df_dict["Token_Similarity"].append(token_similarity)
```

Next, we had to classify the commit as major or minor based on the semantic similarity and token similarity and also report whether both the classes agree. We used the following criteria:
Semantic Similarity >= 0.80 -> Minor Fix else Major Fix
Token Similarity >= 0.75 -> Minor Fix else Major Fix

```
semantic_class = "Minor" if semantic_similarity >= 0.8 else "Major"
token_class = "Minor" if token_similarity >= 0.75 else "Major"

output_df_dict["Semantic_Class"].append(semantic_class)
output_df_dict["Token_Class"].append(token_class)
class_agree = "Yes" if semantic_class == token_class else "No"
output_df_dict["Classes_Agree"].append(class_agree)
```

In the end, we simply convert our dictionary into a pandas dataframe and export it to a csv.

```
output_df = pd.DataFrame(output_df_dict)
output_df.to_csv(r"C:\Users\HP\Desktop\CS_202\Labs_1-4\STT_Lab_3\Output.csv")
```

RESULTS AND ANALYSIS

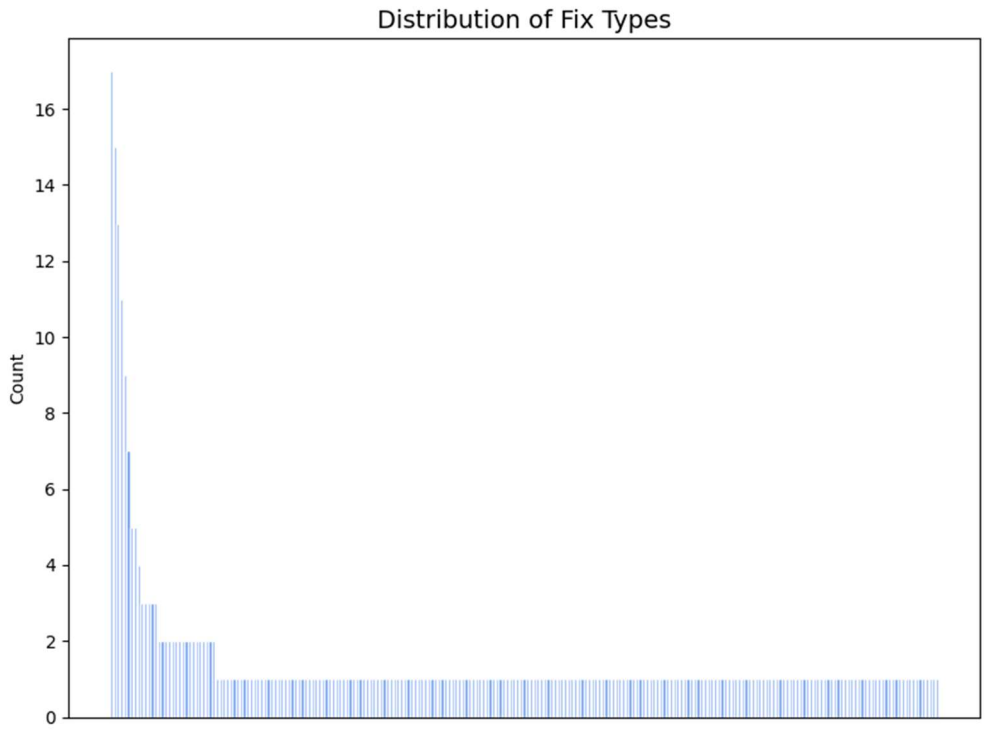
These are the results we obtained from our initial baseline desciptive statistics:

- Total number of commits: 252
- Total number of files: 48
- Average number of modified files per commit: 1.38

Note that these metrics include only bug fixing commits, the total number of commits in the repository is actually close to 1000.

```
Total number of commits: 252
Total number of files: 48
Average number of modified files per commit: 1.38
```

The distribution of the fix types can be visualised from the figure provided below:



Since there were a lot of unique entries for fix types, I have not include the name of the fix types in the graph, but here is some more information about the fix types:

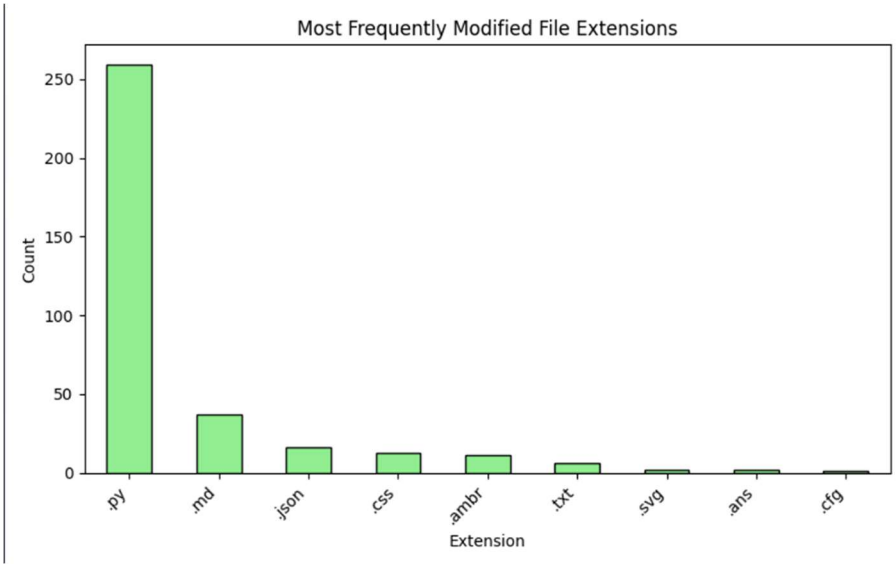
```
Distribution of Fix Types:
LLM_Inference
add missing import                          17
add missing docstring                       15
add missing docstrings                      13
add comment                                 11
add a comment                                9
                                            ..
add note about focused character             1
add missing newline                          1
add debugpip install to test-cov             1
add example for paintapp                     1
add description of selection box border      1
Name: count, Length: 243, dtype: int64
```

This result shows us that the most common fix type is *'add missing import'* with it being the commit message for 17 modified files. From the graph we can observe that most of the fix types occur only once. This can be attributed to them being specific bug fixes, for e.g. *'add example for paintapp'*. Since this is a very specific fix, it does not occur in any of the subsequent commits.
But, more generic fix types like '*add comment'* occur multiple times because this type of bug is common and can occur multiple times in the entire commit history.

Below are the top 10 files (by name) that have been modified the most:

```
Most Frequently Modified Files:
Filename
paint.py                    128
README.md                    32
inspector.py                 16
test_snapshots.py            14
enhanced_directory_tree.py   11
test_snapshots.ambr          11
paint.css                    11
windows.py                   11
wallpaper.py                  8
gallery.py                    7
Name: count, dtype: int64
```

The below figure shows us the most frequently modified file extensions:



As is clear from the graph, the most frequently modified file extension is .py i.e. python source code files. This makes sense as most of the code in this repository was written in python, so it follows that python files would be modified the most.

Next we report the results from the csv that we obtained after running our code on the data:

MI_Change	CC_Change	LOC_Change	Semantic_Similarity	Token_Similarity	Semantic_Class	Token_Class	Classes_Agree
0	0	0	0.99375236	0.969281776	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	0.987149239	0.95986539	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	0	0	Major	Major	Yes
0	0	0	0	0	Major	Major	Yes
0	0	0	0	0	Major	Major	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	1	1	Minor	Minor	Yes
0	0	0	0.99999994	1	Minor	Minor	Yes
0	0	0	0.99999994	1	Minor	Minor	Yes
0	0	0	0.99999994	1	Minor	Minor	Yes

We can see from the csv that for most of the cases, the values of semantic similarity and token similarity are very close to 1. This implies that there were just minor changes in the code and hence the similarity scores are very high. This is also consistent with the data in our Semantic_Class and Token_Class columns, who classify these entries as minor commits.

We observe some entries where the similarity scores are 0. This happens when a new file was added to the repo in the current commit, hence there is no corresponding entry before the commit. That is why the similarity scores are reported to be 0.

We also observe that there are some missing entries in the csv's columns. As was discussed earlier radon can only analyse python source code so if any non python file was modified in the commit, we skipped it and these missing entries correspond to these non-python files.

DISCUSSION AND CONCLUSION

In this lab, we looked at bug-fix commits using both code quality metrics and similarity measures. The quality metrics (Maintainability Index, Cyclomatic Complexity, and Lines of Code) showed how the fixes affected the structure of the code, while the semantic and token similarity scores gave an idea of how big the changes were.

We expected that these measures would not always agree. But after analysing the data, it largely seemed that the metrics were in agreement with each other. We might have got this results based on the repository we have chosen. It is possible that most of the bug fixing commits were minor and there were not many logical bugs, hence we get similar results from multiple metrics.

Overall, the lab highlighted the importance of combining different types of analysis. Using both structural and similarity-based measures gives a clearer understanding of bug fixes and their impact.

I initially did not know that radon only works for python source code as initially I had chosen a repository that did not have any python code. So, I had to choose a new repo, redo the experiment of lab 2 on it and then use the results for this lab. Even then, I got many empty values because not all fixes involved python code. That is why the results we have reported might be a little incomplete.

REFERENCES

[1] https://pypi.org/project/radon

[2] https://github.com/mjpost/sacrebleu

[3] https://pypi.org/project/code-bert-score/

LABORATORY SESSION 4

INTRODUCTION

The aim of the fourth laboratory was to explore the differences in diff outputs for different git diff algorithms like myers and histogram for Open-Source Repositories in the wild. We analyzed the diff output due to variants of the diff algorithm applied in the wild and we also analyzed the impact of different diff algorithms on code versus non-code artifacts.

TOOLS

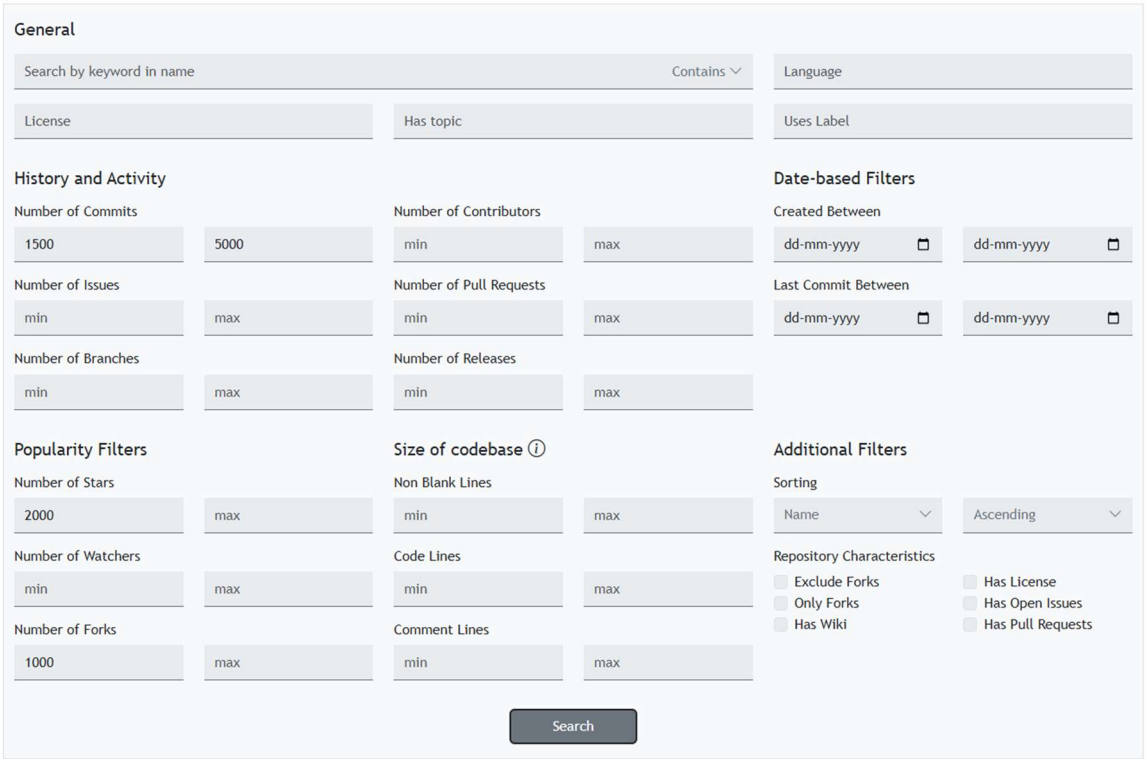
The major tools I used for this laboratory session include SEART GitHub Search Engine which I used for finding the GitHub repositories based on the filters, pydriller (which I used for mining the GitHub repositories for the for finding the git diff using two different algorithms), along with Pandas and Matplotlib for processing the obtained dataset. Apart from these tools, I used Visual Studio Code to perform the experiments on my Windows system.

SETUP

There was not much setup needed to be done as all the libraries I needed for this lab were already downloaded on my system. I had done this using the 'pip install <toolname>' command. Visual Studio Code was already setup on my system, so all I had to do was write the code and perform the experiments.

METHODOLOGY AND EXECUTION

The first stage of the tasks was to search for 3 medium sized GitHub repositories on which I will be performing the analysis. I used the GitHub SEART tool for filtering the repositories. The criteria for filtering the repository are displayed in the screenshot below:



The criteria were:

- Between 1500 and 5000 Commits
- More than 2000 Stars
- More than 1000 Forks

Based on these criteria, I found 3 repositories: ggml, 1Panel and v2rayN. Links to all the repositories are mentioned in the references section. After finalizing the repositories, I mined them using pydriller. I used the Repository.traverse_commits() method from pydriller to traverse over all the commits in these repositories and obtained information regarding these commits like. old and new file paths, commit SHAs of the current and parent commits, commit messages, myers difference and the histogram difference. But, I had to run the loop twice because I had to find both the myers and histogram difference. Now let us dive into how the code works.

I first imported the necessary libraries:

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

Then I declared my repositories in an array which pydriller will traverse sequentially. I also declared an empty dictionary with all the necessary column. This will later be converted to a pandas DataFrame to export it into a csv file.

```
def main():
    repos = ["https://github.com/ggml-org/ggml", "https://github.com/1panel-dev/1panel", "https://github.com/2dust/v2rayn"]

    final_dataset_dict = {"old_file_path": [], "new_file_path": [], "commit_SHA": [], "parent_commit_SHA": [], "commit_message": [],
                          "diff_myers": [], "diff_hist": [], "Discrepancy": []}
```

Then, I traverse through all the modified files in every commit and obtain the parameters such as old file path, new file path, etc. and add them to my dictionary. A key thing to note here is that this loop will calculate the git diff using the myers algorithm. I will run a second loop to find the git diff using histogram algorithm and then calculate the discrepancy between the two git diff algorithms. I have also set the skip_whitespaces parameter equal to True so that the diff algorithm will ignore the white spaces and empty lines while analysing the code.

```
for commit in Repository(repos, skip_whitespaces=True, histogram_diff=False).traverse_commits():
    for file in commit.modified_files:
        final_dataset_dict["old_file_path"].append(file.old_path)
        final_dataset_dict["new_file_path"].append(file.new_path)
        final_dataset_dict["commit_SHA"].append(commit.hash)
        if commit.parents:
            final_dataset_dict["parent_commit_SHA"].append(commit.parents[0])
        else:
            final_dataset_dict["parent_commit_SHA"].append("")
        final_dataset_dict["commit_message"].append(commit.msg)
        final_dataset_dict["diff_myers"].append(file.diff)
```

After this, I ran a second loop to calculate the git diff using the histogram algorithm. Then I checked whether this diff exactly matched the git diff using the myers algorithm and then based on this, I added the appropriate value in the discrepancy column.

```
idx = 0
for commit in Repository(repos, skip_whitespaces=True, histogram_diff=True).traverse_commits():
    for file in commit.modified_files:
        final_dataset_dict["diff_hist"].append(file.diff)
        if file.diff == final_dataset_dict["diff_myers"][idx]:
            final_dataset_dict["Discrepancy"].append("No")
        else:
            final_dataset_dict["Discrepancy"].append("Yes")
```

The next step was to classify the discrepancy according to the file type: namely for source code files, test code files, README files and license files. First I initialized 8 variables to 0 to track the total number of files and number of files having discrepancy for each file type:

```
n_src_files = 0
mis_src_files = 0
n_test_files = 0
mis_test_files = 0
n_readme_files = 0
mis_readme_files = 0
n_license_files = 0
mis_license_files = 0
```

The next step was to classify the file type. We get access to the path of the file using the file.old_path and file.new_path attributes. So, I used this feature to get the path of the file and if the keyword ‘test’ was present in the path, I classified the file as a test code file. Otherwise, if the name of the file was ‘README.md’, I classified it as a readme. Else, if the name of the file was ‘License’ of ‘License.txt’ I classified it as a license file. Otherwise I classified it as a source code file.

```
path = ""
if file.old_path:
    path = file.old_path
else:
    path = file.new_path

if "test" in path:
    if final_dataset_dict["Discrepancy"][-1] == "Yes":
        mis_test_files += 1
    n_test_files += 1

elif file.filename == "README.md":
    if final_dataset_dict["Discrepancy"][-1] == "Yes":
        mis_readme_files += 1
    n_readme_files += 1

elif file.filename == "LICENSE.txt" or file.filename == "LICENSE":
    if final_dataset_dict["Discrepancy"][-1] == "Yes":
        mis_license_files += 1
    n_license_files += 1

else:
    if final_dataset_dict["Discrepancy"][-1] == "Yes":
        mis_src_files += 1
    n_src_files += 1

idx += 1
```

Then we visualized our results by plotting a bar graph using matplolib:

```
categories = ["Test Files", "README", "License", "Source Files"]
discrepancies = [mis_test_files, mis_readme_files, mis_license_files, mis_src_files]
totals = [n_test_files, n_readme_files, n_license_files, n_src_files]

print("Discrepancies by Category:")
for cat, mis, total in zip(categories, discrepancies, totals):
    print(f"{cat}: {mis} / {total} files had discrepancies")

plt.figure(figsize=(7,5))
bars = plt.bar(categories, discrepancies, color=["#66c2a5","#fc8d62","#8da0cb","#e78ac3"], edgecolor="black")

for bar, count in zip(bars, discrepancies):
    plt.text(bar.get_x() + bar.get_width()/2, bar.get_height() + 0.2, str(count),
             ha='center', va='bottom', fontsize=9)

plt.title("Number of Discrepancies by File Type")
plt.ylabel("Discrepancy Count")
plt.tight_layout()
plt.show()
```

After this, the only thing left to do was to convert the dictionary to a pandas DataFrame and export it to a csv file.

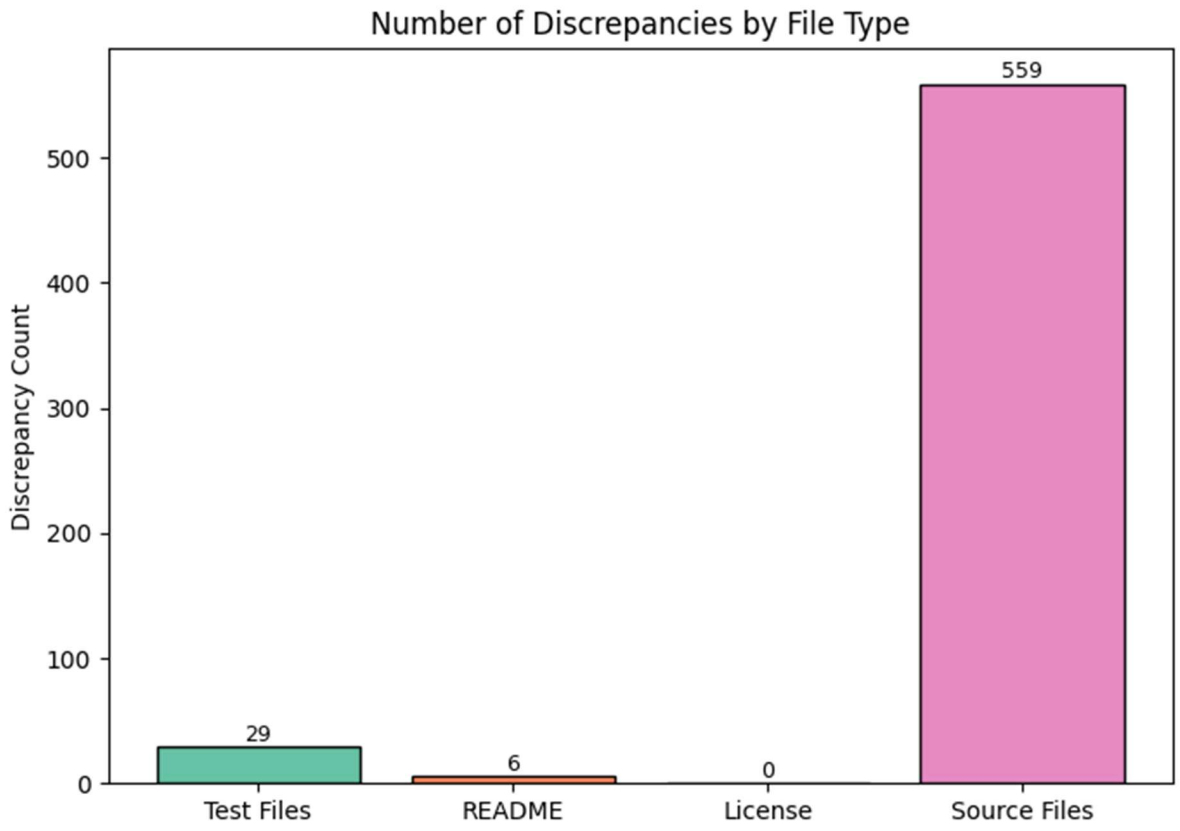
```
    df = pd.DataFrame(final_dataset_dict)
    df.to_csv(r"C:\Users\HP\Desktop\CS_202\Labs_1-4\STT_Lab_4\Discrepency.csv", index=False)
    mismatches = {"Source Code": [mis_src_files, n_src_files], "Test Code": [mis_test_files, n_test_files],
                  "README": [mis_readme_files, n_readme_files], "License": [mis_license_files, n_license_files]}

    print(mismatches)

if __name__ == "__main__":
    main()
```

RESULTS AND ANALYSIS

Below is the screenshot of the csv that we obtained after running the experiments. The columns in order are: commit SHA, parent SHA, commit message, myers diff, histogram diff and discrepancy.

commit_SHA	parent_commit_SHA	commit_message	diff_myers	diff_hist	Discrepancy
0f4e99b1cc357cfff21178dfd5027	fb558f78d905f85c5481360264	Update README.md	@@ -86,7 +86,7 @@ The most performance critical par	@@ -86,7 +86,7 @@ The most perfo	No
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -1,24 +1,23 @@ # ggml -Tensor library in C for mac	@@ -1,24 +1,23 @@ # ggml -Tensor l	No
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -1,7 +1,6 @@ # gpt-2 This is a C++ example running	@@ -1,7 +1,6 @@ # gpt-2 This is a C	No
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -4,25 +4,23 @@ Local GPT-J inference on your com	@@ -4,25 +4,23 @@ Local GPT-J infe	Yes
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -14,7 +14,7 @@ struct gpt_params { int32_t see	@@ -14,7 +14,7 @@ struct gpt_para	No
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -48,6 +48,10 @@ set(TARGET ggml) # endif() #end	@@ -48,6 +48,10 @@ set(TARGET gg	No
f21b84cd217a4cb39cc21c6dcfce	0f4e99b1cc357cfff21178dfd50	Update README.md + minor stuff- Changed default thre	@@ -12,12 +12,11 @@ #include <pthread.h> #define G	@@ -12,12 +12,11 @@ #include <pth	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -25,6 +25,8 @@ option(GGML_SANITIZE_UNDEFIN	@@ -25,6 +25,8 @@ option(GGML_S	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -13,7 +13,7 @@ Tensor library for machine learnir	@@ -13,7 +13,7 @@ Tensor library f	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -3,3 +3,4 @@ target_include_directories(ggml_utils	@@ -3,3 +3,4 @@ target_include_dir	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -0,0 +1,6434 @@ +/*+WAV audio loader and writer.	@@ -0,0 +1,6434 @@ +/*+WAV audio	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -366,8 +366,6 @@ bool gpt2_eval(const int n_h	@@ -366,8 +366,6 @@ bool gpt2_ev	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -558,7 +558,7 @@ bool gptj_eval(inpL);	@@ -558,7 +558,7 @@ bool gptj_eva	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -57,6 +57,25 @@ void gpt_print_usage(int argc, cha	@@ -57,6 +57,25 @@ void gpt_print_	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -28,10 +28,10 @@ struct gpt_params { std::strin	@@ -28,10 +28,10 @@ struct gpt_pa	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -0,0 +1,6 @@ +#+# whisper++set(TEST_TARGET whi	@@ -0,0 +1,6 @@ +#+#+# whisper++set	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -0,0 +1,29 @@ +#+# whisper++Port of [OpenAI's Whis	@@ -0,0 +1,29 @@ +#+#+# whisper++Port	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -0,0 +1,328 @@ +#+# Convert Whisper transformer m	@@ -0,0 +1,328 @@ +#+# Convert Whis	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -0,0 +1,2292 @@ +#+#include "ggml.h"++#define USE_	@@ -0,0 +1,2292 @@ +#+#include "ggm	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -12,6 +12,7 @@ extern "C" { #define GGML_MAX_	@@ -12,6 +12,7 @@ extern "C" { #de	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -59,6 +59,7 @@ add_library(${TARGET} target_incl	@@ -59,6 +59,7 @@ add_library(${T	No
787efb4d2e8ec80ccf48168cf5b9	f21b84cd217a4cb39cc21c6dcfc	Adding Whisper inference example	@@ -1,4 +1,4 @@ -#include "ggml/ggml.h"+#include "gg	@@ -1,4 +1,4 @@ -#include "ggml/gg	Yes
0116c03fb7d697e92072d368646	787efb4d2e8ec80ccf48168cf5b	whisper : various updates and improvements	@@ -9,13 +9,13 @@ Checkout https://github.com/gger	@@ -9,13 +9,13 @@ Checkout https:	No
0116c03fb7d697e92072d368646	787efb4d2e8ec80ccf48168cf5b	whisper : various updates and improvements	@@ -158,11 +158,11 @@ const std::map<e_model, size	@@ -158,11 +158,11 @@ const std::	Yes
dd1f4dfbab5d405d27f0f7c95935	0116c03fb7d697e92072d36864	whisper : various fixes	@@ -1859,7 +1859,7 @@ whisper_vocab::id whisper_s	@@ -1859,7 +1859,7 @@ whisper_v	No

Now let us take a look at the total number of discrepancies according to file type. The bar graph below gives the count of the total number of discrepancies according to file type:



Here is a more detailed breakdown along with total number of files:

```
Discrepancies by Category:
Test Files: 29 / 551 files had discrepancies
README: 6 / 135 files had discrepancies
License: 0 / 2 files had discrepancies
Source Files: 559 / 7905 files had discrepancies
```

Total number of discrepancies in percentage are:

- Source Code Files: **7.07%**
- Test Code Files**: 5.26 %**
- README Files: **4.44%**
- License Files**: 0%**
- Overall: **6.91%**

After analysing the results we can see that the overall discrepancy is almost 7%, which is the expected value as we were expecting it to be between 0 – 10%. If we look at the individual discrepancies, we can see that the license files have 0 discrepancies which is reasonable because in the entire commit history a license file was edited only twice so there was not much chance to get a discrepancy. As for the other file types, the percentage of discrepancies is roughly similar and in the expected range.

Now to answer the question on how to automatically find which algorithm would perform better, here is how I would proceed: To decide which algorithm performs better, I would first define evaluation criteria such as conciseness of the diff, number of changed lines, readability, and runtime efficiency. Then, I would run both Myers and Histogram on the same set of commits, record metrics like patch size and number of hunks, and compare them. If a ground truth (e.g., human-reviewed diffs) is available, I would use it to validate which

algorithm aligns better with expected results. Finally, aggregating these results would reveal which algorithm consistently produces more meaningful and efficient diffs.

DISCUSSION AND CONCLUSION

These findings suggest that while Myers and Histogram generally agree, there are still cases where they diverge. This divergence may arise from the different ways each algorithm minimizes and groups changes. For example, Histogram tends to generate diffs that are more context-aware, whereas Myers aims to minimize the number of changes. Such differences may be small in percentage but can have practical implications when analyzing large repositories or when readability of diffs is critical.

Overall, the results highlight that discrepancies, though limited in frequency, are not negligible. They demonstrate the importance of understanding how diff algorithms behave in practice, especially when analyzing open-source repositories where even minor differences in reported changes could influence downstream analyses, such as bug localization or patch quality evaluation.

In conclusion, this lab showed that while both algorithms are generally reliable, their subtle differences underline the need to carefully choose the diff variant depending on the analysis goal, whether it is minimizing changes or improving readability.

One significant challenge I faced in this lab was the runtime of the code. Since I had to analyse three big repositories, pydriller took a lot of time to run. Also, the csv generated had more rows than the limit of MS Excel, so opening the csv file was also a challenge and I had to figure out other ways to open the file

REFERENCES

[1] https://github.com/2dust/v2rayn

[2] https://github.com/1panel-dev/1panel

[3] https://github.com/ggml-org/ggml

[4] https://seart-ghs.si.usi.ch/

[5] https://pydriller.readthedocs.io/en