COMP-3110 Final Project

Aditya Patel
School of Computer Science
University of Windsor
110129311
patel8a9@uwindsor.ca

Joshua Lebert
School of Computer Science
University of Windsor
110139519
lebertj@uwindsor.ca

Julia Ducharme
School of Computer Science
University of Windsor
110113245
duchar91@uwindsor.ca

Joanne Hou
School of Computer Science
University of Windsor
110178405
hou95@uwindsor.ca

***Abstract*—This project utilizes Python and various libraries to detect changes, specifically line deletion, line splitting, line insertion, and code modification across file versions. We used a hybrid, context-aware approach to complete this task, finding complex structural refactoring to be a challenge and exact matches or minor edits to be straightforward.**

Keywords—change detection, line mapping, SZZ algorithm, bug-introducing changes, software evolution, hybrid similarity metric, Simhash, Git blame

I. Introduction

The main problem we are addressing is tracking line changes across file versions. In practice, this tool can aid with locating bug introducing changes, tracking code fragments across versions, merging file versions, code review, and software analysis evolution. These are all important aspects of the software development cycle and can greatly benefit teams regardless of size.

We approached this problem by carefully laying out the base requirements and features, before selecting the language we would write it in. We evaluated and analyzed all of the existing solutions to gain a deeper understanding of what we were trying to implement, as well as if it was possible to reuse parts of the existing solutions to achieve the end goal.

Next, we began forming a detailed plan to divide up the work evenly for efficiency. The decision we made was to have each member focus on their own part of the project, which allows each member to work individually when we aren't able to get together in person. Whenever we were working together in person, we all worked on the features together to ensure continuity within all elements of the project.

Overall, this project had a somewhat straightforward set of requirements, though optimizing our algorithm for complex changes was challenging.

II. Data Collection

We selected files from GitHub to serve as our base pairs. We then determined the file mappings by reviewing the code and manually documenting the differences.

III. Technique Description and Evaluation

A. Technique Description

Our approach to the line-mapping problem utilizes a multi-stage pipeline designed to handle both exact matches and modified code blocks. We implemented a hybrid similarity metric that balances textual content with structural context.

a. Preprocessing and Exact Matching

First, we normalized all source files by removing excess whitespace and converting text to lowercase to ensure consistency. We utilized Python's difflib to perform an initial pass. Lines flagged as unchanged by difflib were immediately mapped as exact matches (1: 1), serving as "anchors" for the subsequent stages.

b. Candidate Generation and Filtering

For lines identified as modified (insertions and deletions), we generated candidate pairs. To avoid the computational expense of an $O(N \times M)$ comparison, we employed Simhash, a locality-sensitive hashing technique. For every unmatched line in the source file, we calculated the Hamming distance against lines in the target file, selecting only the top $k = 15$ candidates with the lowest distance for detailed scoring.

c. Hybrid Similarity Scoring

We evaluated candidate pairs using a composite score derived from two metrics:

1. *Content Similarity*: We calculated the normalized Levenshtein ratio between the text of the two lines.

2. *Context Similarity*: We examined a window of 4 lines surrounding the candidate. We constructed term frequency vectors for these windows and calculated the Cosine Similarity:

$$Similarity_{context} = \frac{A \cdot B}{||A|| \cdot ||B||}$$

These were combined into a final weighted score:

$$Score = \alpha \cdot Similarity_{context} + (1 - \alpha) \cdot Similarity_{conte}$$

We utilized an α value of 0.6, prioritizing line content over context.

d. Selection and Refinement

We used a greedy selection algorithm where pairs with the highest score above a threshold of 0.6 were mapped. Finally, we implemented a look-ahead and look-behind refinement step. This checked if merging adjacent lines in the target file (forming a one-to-many mapping) improved the Levenshtein ratio, effectively handling cases where a single line of code was split into multiple lines in the newer version.

B. Evaluation

To evaluate our solution, we developed an automated Python evaluation script (evaluate_all.py). This script automates the comparison between our generated line mappings and the manually verified "Ground Truth" provided in the dataset posted on D2L.

a. Evaluation Logic and Metrics

The evaluation script operates by parsing the hierarchical XML structures of both the generated output and the ground truth files using Python's xml.etree.ElementTree library. It extracts the mappings for each file version into hash maps, where keys represent source line numbers and values represent sets of target line numbers.

We employed a Strict Set Equality metric for accuracy. For a mapping to be considered correct, the set of target lines identified by our tool (S_{gen}) must be identical to the set of target lines in the ground truth (S_{gt}):

$$Accuracy = \frac{\sum_{i=1}^{n} ((S_{gen})^{(i)} = (S_{gt})^{(i)})}{N}$$

where N is the total number of mappings in the ground truth. This strict ruleset means that partially correct mappings (e.g., identifying only 2 out of 3 lines in a split block) are penalized as failures, ensuring that our reported accuracy reflects precise, complete matches.

b. Data Aggregation (results.csv)

The script processes all 23 test files, calculating individual accuracy scores. These results are collected and exported to results.csv for analysis.

As detailed in the generated CSV report (See Fig. 1.), our technique evaluated a total of 545 line mappings across the dataset. The tool successfully identified 462 correct matches, resulting in an overall aggregate accuracy of 84.77%. While many files (e.g., ArrayReference, JavaCodeScanner) achieved 100% accuracy, complex files with significant refactoring (e.g., BaseTypes) highlighted specific areas for future optimization in our context-aware scoring algorithm.

Table 1: results.csv

Test File	Accuracy (%)	Correct Matches	Total Mappings
asdf	91.67	11	12
ASTResolving	91.67	33	36
ArrayReference	100	24	24
BaseTypes	42.86	30	70
BuildPathsPropertyPage	96.15	25	26
CPListLabelProvider	95.83	23	24
CompilationUnitDocumentProvider	97.06	33	34
DeltaProcessor	91.67	22	24
DialogCustomize	100	16	16
DirectoryDialog	100	6	6
DoubleCache	58.33	14	24
FontData	82.14	23	28
GC	96.15	25	26
GC2	100	22	22
JavaCodeScanner	100	26	26
JavaModelManager	100	12	12
JavaPerspectiveFactory	100	16	16
PluginSearchScope	75	12	16
RefreshLocal	100	12	12
ResourceCompareInput	96.67	29	30
ResourceInfo	92.31	24	26
SaveManager	63.64	7	11
TabFolder	70.83	17	24
TOTAL / AVERAGE	84.77	462	545

Fig. 1: Accuracy of line mapping technique across different file pairs, compared against the provided Ground Truth.

IV. Presentation of Line Mapping Information

Below is the visualization we made of the GUI users would see their line mapping information in:

V. Identifying Bug Introducing Changes

For the bonus mark, the technique we used to identify bug introducing changes was the SZZ algorithm, named

after its creators Sliwerski, Zimmermann, and Zeller. This algorithm answers the important question: "When a bug is fixed, which commit originally introduced it?". We carefully planned our approach on this problem by reading the paper on the algorithm [1].

A. How It Works

Our implementation follows the SZZ approach in three stages:

1. *Bug-Fix Commit Identification*: We identify potential bug-fix commits by analyzing commit messages for common patterns (see Fig. 3.):
    a. Keywords: "fix", "fixed", "fixes", "fixing", "bug"
    b. Issue references: "closes #", "resolves #"

```
# Find commits that look like bug fixes
fix_patterns = ['fix ', 'fix:', 'fix(', 'fixes ', 'fixed ', 'fixing ',
                'bug ', 'bug:', 'closes #', 'close #', 'resolves #', 'resolve #']
```

Fig. 3. Shows the keywords and issue references used to find a bug-fix commit

2. *Changed Line Extraction*: For each bug-fix commit, we use PyDriller to extract the diff and identify lines that were deleted or modified—these represent the buggy code that was removed during the fix.
3. *Origin Tracing with Git Blame*: We run git blame on the parent of the fix commit to trace each buggy line back to the commit that originally introduced it. This reveals the bug-introducing change.

B. Implementation

The tool is implemented in Python using:

1. PyDriller for repository mining and diff parsing
2. Git blame (via subprocess) for line-level origin tracing

C. Testing & Results

We validated our implementation on two repositories (See Fig. 4.):

1. Our test repo: found 2 fix commits, correctly traced bug to original commit
2. Pydriller: found 50+ fix commits, identified multiple bug origins

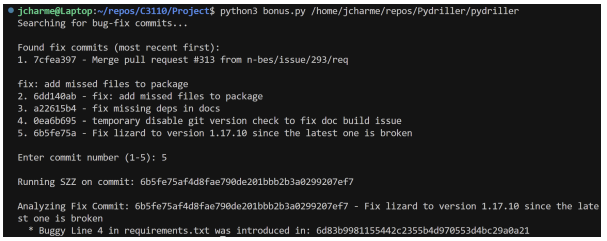

Fig. 4. Example showing the terminal when the algorithm is run. Displays the found commits, then prompts the user to enter a number for the commit to be checked out. The SZZ algorithm is ran on the commit, which then shows the buggy lines from each file in the commit.

D. Limitations

Our implementation uses the original SZZ algorithm, which may produce some false positives in cases of:

1. Cosmetic changes (whitespace, formatting)
2. Code refactoring (renames, moves)

These limitations were addressed in later work by Kim et al. [2], but are outside the scope of this project.

References

[1] Jacek Śliwerski, Thomas Zimmermann, and Andreas Zeller. When do changes induce fixes? On Fridays. Proc. International Workshop on Mining Software Repositories (MSR), Saint Louis, Missouri, USA, May 2005.

[2] S. Kim, T. Zimmermann, K. Pan and E. J. Jr. Whitehead, "Automatic Identification of Bug-Introducing Changes," 21st IEEE/ACM International Conference on Automated Software Engineering (ASE'06), Tokyo, Japan, 2006, pp. 81-90.