LLM-Based Automated Code Repair Agent

Documentation of the FastCodeCorrectionAgent Approach

Overview

The FastCodeCorrectionAgent is a Gemini-powered automated repair agent for Python programs in the QuixBugs dataset. It identifies and fixes single bug functions using structured prompt engineering, validates fixed code against ground truth, and tests it with real input-output cases. It is a robust agent fixing fixing all the bugs present in Quixbugs dataset.

Approach Summary

1. Input: Buggy Python programs are imported from the QuixBugs dataset.

2. Prompt Generation: Formulates one or more repair prompts for Gemini using either pattern-based or step-by-step logic. Use of react based prompt templates.

3. Model Inference: Uses Gemini (either 1.5-flash or 1.5-pro) to generate corrected code.

4. Postprocessing: Cleans output and validates Python syntax.

5. Structural Comparison: Compares fixed code against the correct solution using AST-based structural similarity and exact string match.

6. Test-Driven Validation: Executes the fixed program on unit test inputs using timeouts to prevent hangs.

7. Reporting: Aggregates accuracy, similarity, and success rate metrics for batch repair.

Key Components

- code_corrector.py: Contains the FastCodeCorrectionAgent class.

- test_fixed_only.py: Runs only the fixed versions of repaired code with timeout protection.

- tester.py: Executes programs with fallback to multiprocessing for slow test cases.

- fixed_programs/: Stores the output of the repaired code.

- correct_python_programs/: Ground-truth correct implementations.

- json_testcases/: Input-output test cases for validation.

Problems Faced

The tool faces several significant operational challenges that users should be aware of. API key exhaustion is a frequent issue, especially when processing large datasets, as Google Gemini has strict rate limits and quota restrictions that can halt processing mid-batch. Performance bottlenecks occur due to network latency, API response times (averaging 2-4 seconds per request), and the inherent slowness of sequential processing required to avoid rate limiting. Memory consumption can become problematic when processing many files concurrently, and the tool may experience timeout issues with complex code repairs that require multiple AI model calls. Additionally, the accuracy vs speed trade-off creates a dilemma where faster processing (using gemini-1.5-flash with minimal delays) yields lower success rates (75-85%), while higher accuracy modes (using gemini-1.5-pro with multi-attempt strategies) significantly increase processing time and API costs. Cost management becomes critical as each program fix can require 1-3 API calls, making large-scale processing expensive, and users often encounter billing limits or unexpected charges when running extensive batch operations. Overall, there were a lot issues, but it was one of unique problems that I faced.

Performance

In a recent run (standard mode, gemini-1.5-flash), the agent processed 41 buggy programs, successfully fixing 40.

- Success Rate: 97.56%

- Average AST Similarity: 62.07%

- Programs per Minute: ~22

Configuration

- Accuracy Mode: fast, standard, high, maximum

- Fast Mode: Skip deep validation

- Concurrent: Optional multithreaded batch repair

- Model: gemini-1.5-flash (fast) or gemini-1.5-pro (accurate)

Conclusion

This agent demonstrates efficient, scalable code repair using LLMs with integrated validation and evaluation, suitable for benchmarking, education, and research in automated program fixing. This was a good experience overall and I would like to thanks AIMS DTU for this opportunity.