The Impact of Large Language Models on Automated Software Testing

Abdelrahman Amr Effat

Ibrahim Nasser Darwish   Abdrelrahman Mostafa Rifaat

Abstract

Your abstract.

1 Introduction

Your introduction goes here! Simply start writing your document and use the Recompile button to view the updated PDF preview. Examples of commonly used commands and features are listed below, to help you get started.

Once you're familiar with the editor, you can find various project settings in the Overleaf menu, accessed via the button in the very top left of the editor. To view tutorials, user guides, and further documentation, please visit our help library, or head to our plans page to choose your plan.

2 Literature Review

Automated unit test generation has been increasing in popularity for a long time now as way to reduce manual effort (at least regarding minimal tests that require low effort).

In recent years, many test generating tools have emerged, such as Pynguin, a python-focused framework that uses dynamic typing and evolutionary search algorithms (such as genetic algorithms) to produce mutation tests[LF22]. We should also mention Evosuite, one of the most popular unit test generation tools that JUnit tests for java-based projects while ensuring flakiness protection.

Despite these advances, traditional testing tools suffers while dealing with some domain specific or complex tests, that is due to their type of nature of being more focused on covering code paths, either line, branch or any type of covering, rather than evaluating corner cases and business logic tests.

It is no secret that the rise of large language models (LLMs) has made a significant impact on many fields, one of them is code and test generation, that is due to them being trained on large corpora of text and code, in addition to the improvement in their reasoning which made them a good candidate in competing with traditional tools.

However, the application of LLMs for the generation of unit tests is not free of challenges, and these tests may fail due to incorrect assertion logic, missing setup/imports code, or syntax errors. All of that causes the tests to fail while running not due to errors or failure of the tested code itself but because of the defects in the tests.That is why the LLM generated tests should be checked and processed after generation to ensure their quality [JWS+24].

One of the recent advancements in this field was a research[YYG+24] done in 2024, where they did a a comprehensive study evaluating several LLMs using different prompting rules, the analyzed about 17 java projects using defects4J data and compared the LLM tests versus that generated by Evosuite. Their findings highlighted the potential of open source models in this field, but it also pointed out to the limitations mentioned before, showing the way for future enhancement in LLM-based test generation methodologies.

3 Methodology

3.1 Objective

This study aims to answer two research questions :

1. Can open source LLMs compete with traditional testing tools (maybe even surpassing and replacing them) ?
2. Does a larger model always offer a better performance, and It is not worth the additional cost ?

3.2 Dataset

For the data, We used CodeRM-UnitTest dataset, a publicly available benchmark dataset that can be accessed through here: `https://huggingface.co/datasets/KAKA22/CodeRM-UnitTest`.

The dataset consists of a collection of python function with some of their corresponding unit tests (generated by LLMs too). It contains a wide range of different functions, with varying complexity, that would be very useful and is actually used in code and test generation

3.3 Large Language Models

For our study, We selected five open source models of different sizes :

1. LLaMA 4 Maverick 17B 128E Instruct (FP8)
2. Qwen2.5 Coder 32B Instruct
3. Meta LLaMA 3.1 8B Instruct Turbo
4. Meta LLaMA 3.2 3B Instruct Turbo

5. Meta LLaMA 3.3 70B Instruct Turbo

At first, we only used the first two models for generation of nearly less than half of the data, then started using the five of them together.

Each model was prompted using zero-shot detailed prompt, that explain the required properties from the generated tests along side any setup configuration to minimize post processing efforts.

All models were accessed through APIs provided by TogetherAI, ensuring consistent inference settings and simplified integration for generating unit tests.

3.4 Baseline: Pynguin

We used Pynguin (v0.25.0) as a search-based baseline for automated test generation. Pynguin applies evolutionary algorithms to generate tests that aim to maximize line and branch coverage. For each function, Pynguin was configured to generate tests within a timeout limit of 10 minutes, using the default configuration settings. The output was a Python test suite written using the `pytest` framework.

3.5 Work Pipeline

The work was divided to two different pipelines use each on its own :

3.5.1 Test Generation

1. Each code in the dataset was passed five times to the called API, one time for each model as not to crowd the single prompt with several codes, along with the system prompt that required test generation following some instructions like :
   - Normal cases
   - Edge cases
   - Invalid inputs
   - Error handling
   - Function correctness
2. The LLMs response was parsed, and the generated test cases was saved in a structured JSON file, which would be used as logs for the evaluation.
3. The codes then were given to Pynguin to generate its tests, while putting a restriction of 10 minutes, which is somehow large but some Pynguin tests actually a very long time, then it was saved in the previously mentioned JSON.
4. The Pynguin codes were saved in intermediate .py file, that was overwritten each time to save storage.

3.5.2 Evaluation & Analysis

1. The Test were run using `pytest.py` tool, whose output was parsed and stored in JSON logs along with the tests and also the results stats were summarized in another JSON file.

2. Then further statistical analysis was done

3.6 Evaluation Metrics

- **Pass Rate:** The ratio between the successful ran tests and the total number of tests evaluated.

- **Coverage:** It is a measurement of how much of the function code was exercised by the tests, line coverage is measured using `coverage.py` tool.

- **Failure Categorization:** We check whether the failed tests was due to deects in test or problems with the tested fucntions.

- **Resources Needed:** This is normally measured in the amount and specs of hardware requirements, but here we valued it as the monetary cost needed to call the model per 1M tokens .

These metrics provide insight into the correctness, reliability, and utility of the generated test cases and help distinguish between genuinely rigorous testing and flawed test generation.

4 Results

5 Future Work

While the results have shown huge potential regarding the utilization of LLMs in test generation , specially with their creativity and reasoning capabilities , they still have many flaws regarding their consistency and reliability.

One of the solution could be to fine-tune the LLMS on domain-specific test generation tasks using code-test pairs , this would leverage their power and performance.

Also, even if they didn't reach the level of human tester, they can still be used for generating candidate tests that would be refiend by the developer , maybe even integrate them into development environments (IDEs).

6 Conclusion

References

[JWS$^+$24] Juyong Jiang, Fan Wang, Jiasi Shen, Sungju Kim, and Sunghun Kim. A survey on large language models for code generation. *arXiv preprint arXiv:2406.00515*, 2024.

[LF22] Stephan Lukasczyk and Gordon Fraser. Pynguin: Automated unit test generation for python, 2022.

[YYG$^+$24] Lin Yang, Chen Yang, Shutao Gao, Weijing Wang, Bo Wang, Qihao Zhu, Xiao Chu, Jianyi Zhou, Guangtai Liang, Qianxiang Wang, and Junjie Chen. On the evaluation of large language models in unit test generation. *arXiv preprint arXiv:2406.18181*, 2024. Accepted by ASE 2024, Research Paper Track.