Team2 : Augmented Auto FL

Automatic Fault Localization with Context-aware LLM

20160494 Jaesung Lee
20160709 Seokwoo Hong
20233087 Donghyeok Kim
20243499 Inseok Yeo

CS453 - 2024, Spring

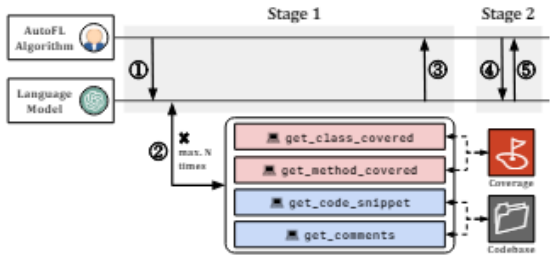
Abstraction

Various software engineering technologies are advancing, and numerous techniques for resolving software bugs are also evolving. Among them, technologies such as AutoFL, which applies fault localization (FL) using LLM models, have been developed. However, in LLMs, to address the deficiencies in error line analysis for specific bugs, the project aims to complement the weaknesses of AutoFL by incorporating real-world information inputs along with analysis of symbolic tables.

1 INTRODUCTION

With the advancement of technology and business, numerous software applications are being developed. However, verification and testing of software remain crucial. Additionally, research has been conducted to leverage AI technologies, such as GPT-3.5, for automation and convenience in various aspects of software testing. Consequently, there is considerable interest in utilizing techniques employing LLMs (Large Language Models) for bug diagnosis, patch generation, test case creation, and other software testing methodologies. An innovative technique called autoFL (Fault Localization) has been devised using LLMs to identify faults within large codebases and pinpoint the elements causing malfunctions. However, some aspects of fault localization intended to be provided by autoFL were too complex for LLMs to handle effectively, leaving them as future work in several cases. To address this, approaches have been proposed to tackle unresolved methods in autoFL, such as providing fault localization elements like the Moscow Time Zone case and symbolic tables preceding error code lines, which LLMs failed to grasp. However, there are financial constraints associated with using APIs of LLMs like GPT-3.5, and positive outcomes from applying other LLMs besides GPT-3.5 are also anticipated.

2 RELATED WORK



(a) Figure 1: Diagram of AutoFL

Listing 1: System Prompt for LLM

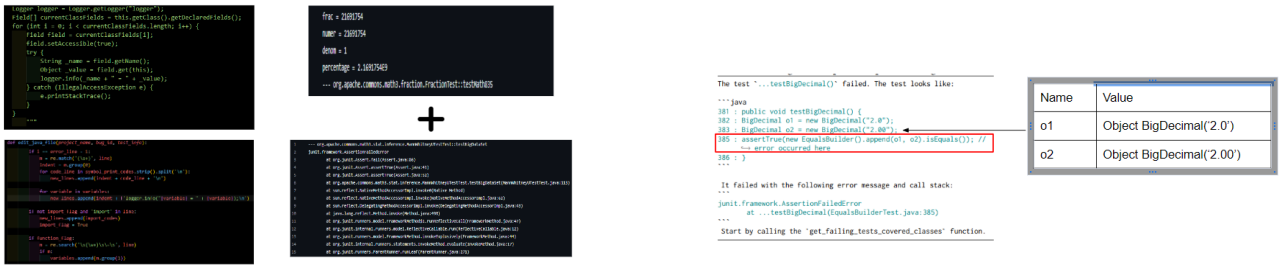
```
You are a debugging assistant. You will be presented with a failing
    ↪ test, and tools (functions) to access the source code of
    ↪ the system under test (SUT). Your task is to provide a step-
    ↪ by-step explanation of how the bug occurred, based on the
    ↪ failing test and the information you retrieved using tests
    ↪ about the SUT. You will be given 9 chances to interact with
    ↪  functions to gather relevant information. An example
    ↪ answer would look like follows.

<HANDCRAFTED ROOT CAUSE ANALYSIS EXAMPLE>
```

(b) Prompt for LLM

Figure 1: Diagram of AutoFL. Each arrow represents a prompt/response between components, with the circled numbers indicating the order of interactions. Function invocations are made at most N times, where N is a predetermined parameter of AutoFL. In this paper, we introduce AutoFL, a novel automated and autonomous FL technique that harnesses LLMs to localize bugs in software given a single failing test. As mentioned earlier, dealing with large code repositories is a challenge for LLMs, but we tackle this issue by equipping LLMs with custom-designed functions to enable code exploration and relevant information extraction. An overview of AutoFL is depicted in Figure 1. We employ a two-stage prompting process, where the first stage involves inquiring about the root cause of the given failure, and the second stage requests output about where the fault location is. In the first stage, 1 AutoFL provides a prompt to the LLM containing failing test information and descriptions of available functions for debugging to LLM. 2 The LLM interacts with the provided functions autonomously, to extract the information needed for the debugging of the given failure. 3 Based on the gathered information, the LLM generates an explanation about the root cause of the observed failure. In the second stage, 4 the user queries for the location of the identified bug, and 5 the LLM responds by providing the culprit method (FL output). In doing so, we can explicitly acquire both the Root Cause Explanation and Bug Location.

3 METHOD

3.1 Providing information of debuggers



(a) This is the first figure.

(b) This is the second figure.

Figure 2: Diagram of AutoFL

We would like to solve the aforementioned problem in two ways. Our approach is providing more detailed information of global and local variables prior to the error line, similar to a symbolic debugger. The example on the red box is an actual bug introduced from autoFL. We believe that knowing what values the variables used in the error line had is important for determining the cause of the error. So we wanted to send the values of the variables to LLM along with the information in the error, just like the symbolic debugger does, so that LLM can better understand the cause of the error. First, we parse the error message to determine where the error occurred. Then, we insert the Java code in the top left corner just before the error occurs to output the value of the class field where the test exists to the test log. In addition to the class field, there are also local variables declared within the test. We read the Java test code line by line and store the names of the local variables. We then insert the code that prints the names of the local variables just before the error line, along with the code that prints the class field. The message with the values of the variables is then parsed and combined with the existing error information to form a new dataset.

3.2 Providing prior knowledge

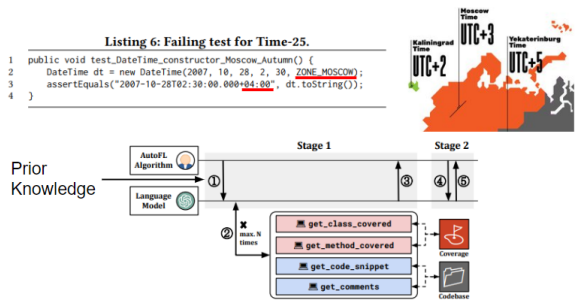


Figure 3: This is the first figure.

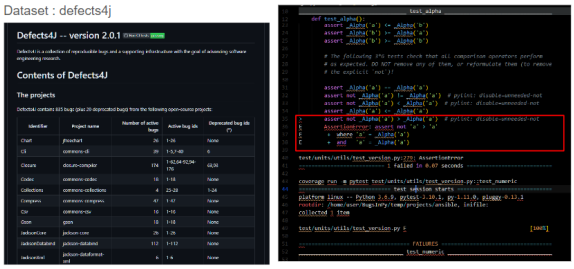
The second part is to provide LLM with prior knowledge to help it understand the code. The example on the top left is also from the paper, which is a test that compares datetimes in the Moscow time zone using assert statement. The LLM suggested modifying `"ZONE_MOSCOW"` as a solution to the above failure case, but the actual timezone in Moscow is GMT+3:00, so the real problem is GMT+4:00 in the assert statement. If we provide this prior knowledge of the real world to the first query, we expect to have a more meaningful approach to following queries, resulting in higher performance.

4 RESULT

4.1 Applying AutoFL to CodeLlama

The result of the first direction is as follow, We have faced some difficulties implementing autoFL with codellama model. The basic codellama and ollama did not support function calling feature, unlike chatgpt, which made it challenging for us to implement the autofl to code llama. We attempted two different methods. We first tried to modify the prompt to make code llama respond as we intended it to. However, it was challenging to autonomously parse the LLM's output. And sometimes it misunderstood the attempted prompts and hallucinates a function call chain on its own.

4.2 Providing information of debuggers



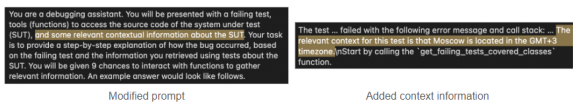
(a) defect4j processing result

	autoFL	autoFL + symbol table
acc@top1	**129**	121
acc@top3	166	**168**
acc@top5	**183**	**183**

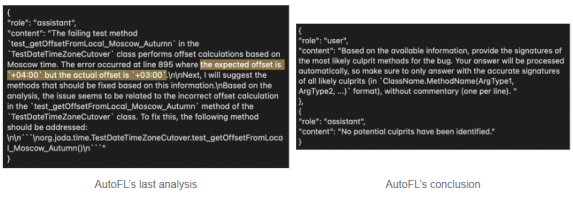
(b) FL Result table

we tried to utilize the langchain wrapper library's function calling feature, however it not only seemed to be able to handle more than 1 type of function, and it did not support Natural language response and function call at the same time. The dataset we used is defect4j, a dataset that collects test code errors from java libraries. While previous baseline studies have used both Bugsinpy and defect4j, we ran our experiments using only defect4j. We used defect4j because bugsinpy contained error information from tests run through pytest, but as you can see from the message in the red box, pytest already shows information about what value the variable has, so we used defect4j without information about the variable. Here is the Top 1, 3, 5 accuracy for the existing d4j and Top 1, 3, 5 accuracy for the d4j where we added text for the symbol table. Since the work on replacing the LLM base with codellama was not completed due to the lack of function call functionality, we based our experiments on gpt3.5. The results are rather worse for top 1 accuracy, and almost similar for top 3 and top 5. We believe that the addition of unnecessary information, such as fields from classes that are not used in the test, makes it more difficult for GPT to determine the cause of the error. Also, we believe that GPT is capable enough to determine the values of the currently declared variables based on the code snippet alone.

4.3 Providing prior knowledge



(a) Modified prompt & adding context information



(b) autoFL's conclusion

Generating relevant prior knowledge of erroneous code automatically is a non-trivial task, so only hand-crafted prompts for selected test cases could be used. Realistically, in an actual fault localization scenario, any contextual information would be found within the code comments, but for simplicity and to first verify if contextual information is useful to the LLM in the first place, the information was added to the initial set of instruction prompts. We conducted a preliminary test on the effect of

real-world prior knowledge using the aforementioned `"Time_25"` data. The initial instruction prompt was modified to inform the LLM that it will be receiving contextual information, and the information about Moscow time zone was added to the simulated user input describing the failed test. The bottom left image shows LLM's last analysis before its final decision, and as seen in the highlighted text, it accurately identifies that there is a mistake in the timezone offset. However, in its response to the actual final fault localization decision, it did not mention the identified bug. We could not identify the cause of this behavior, and plan to attempt using different prompts to resolve this. The effectiveness of this approach may stem from the given context being too direct, almost as if the fault was already identified, so we will attempt with more subtle contextual hints as well.

5 DISCUSSION

This task was carried out as a complementary project of autoFL. However, due to the limited resources of the project to use the autoFL work of the GPT base as it is, we proceeded with LLMs other than GPT as base, and we judged that it would help identify base LLMs that are more effective for LLMs with different transformer structures, such as CodeLlama. We can devise a way for LLM to provide more meaningful results in identifying bugs compared to conventional autoFL by providing environmental setting and development information on error lines in the Detects4J data to explore source code and provide bug-related information in experiments. Depending on the performance of LLM, there may be variations in FL results, but when we apply our proposed method with LLM base such as good performance GPT, we expect to be able to present a method to help software engineers with debugging and bug identification.

6 CONCLUSION

The results of our project differ due to the performance of the GPT-based autoFL and LLM models. However, by searching for bug-causing source code in a large code line and providing bug-related information in advance, it provides a more advanced method as an LLM-based FL performance improvement and debugging assistance tool. The FL results we provide can help us explore how to enable developers to provide meaningful data in making debugging or code-modification decisions. In particular, it will be possible to provide information on environmental variables that are difficult for developers to consider in real time and to provide various ways to develop them based on them.

References

1. Sungmin Kang, Gabin An, Shin Yoo. 2023. A Preliminary Evaluation of LLM-Based Fault Localization arXiv abs/2308.05487 https://doi.org/10.48550/arXiv.2308.05487

Task REPOSITORY

1. https://github.com/inseok-yeo/Aug_auto_fl