Heuristics-Based Induction of LLM for Automated Fault Localization

Alfonsus Rodriques Rendy (121040014)
School of Data Science
The Chinese University of Hong Kong, Shenzhen
121040014@link.cuhk.edu.cn

Abstract

Fault localization (FL) is one of the most intensive task in software engineering, aimed at identifying buggy code segments. This paper introduces InductFL, a heuristics-based automated fault localization framework that leverages large language models (LLMs) to improve the process. By harnessing LLMs' inductive reasoning capabilities, we infer high-quality heuristics that are relevant, precise, and generalizable across bug categories, facilitating the identification of faulty lines in source code. The study adopts a two-phase approach: induction and deduction. In the induction phase, LLMs generate heuristics based on various error types. During the deduction phase, some of the inferred heuristics are selected to guide the accurate identification of buggy lines. This research addresses two primary questions: the quality of heuristics generated across error types and the optimal selection of heuristics for debugging. Our results indicate that LLMs can generate heuristics relevant to various bug categories but face difficulties when identifying heuristics for complex logical bugs. Furthermore, LLMs often produce high-level heuristics that are too abstract for direct implementation. In addition for the heuristic selection in the induction process, a codeBERT-based classification model accurately maps input code instances to the corresponding heuristic class (e.g., bug type). These findings highlight the potential of heuristics-based fault localization while also identifying current limitations of the framework, particularly in generating executable heuristics.

1 Introduction

Fault localization (FL) is one of software engineering task with the objective of finding the parts of code that contain bugs. The parts identified by an FL system could be a line, statement, module, or even a file that has highest probability of causing faults. The process of testing and fault localization is fundamental for ensuring the reliability, functionality, and quality of software.

Manual fault localization, though feasible, is often laborious, time-consuming, and prone to errors, particularly in complex programs with thousands of lines of code. Automated FL methods promise to significantly reduce debugging time, minimize human error, and optimize the software development lifecycle. Traditional techniques such as Spectrum-Based, Slicing-Based, and Machine Learning-Based FL rely heavily on test results and coverage data. However, these methods struggle with extensive coverage measurement and often lack a clear rationale for their outputs, which is crucial for developers creating effective bug fixes [1].

The emergence of Large Language Models (LLMs) presents a promising opportunity for advancing automated FL. LLMs excel at understanding and processing natural language [2], especially with enhancements via human feedback, significantly improving their comprehension and execution of instructions [3]. In-context learning, where LLMs infer desired outputs from a set of demonstra-

[*] This project is supervised by Prof. He Pinjia with collaboration with Mr. Xu Junjielong.

Preprint. Under Review

tions, and inductive reasoning, which iteratively refines hypotheses to uncover high-level heuristics, represent two prominent approaches.

To explore more effective ways of applying LLMs to FL tasks, we propose a new framework, *InductFL*, which aims to uncover hidden heuristics via induction and deduce potential buggy lines based on these heuristics. This research addresses two main questions:

- **RQ1** How do the generated heuristics' quality (relevance, generalisability, precision) vary across heuristic classes based on error types?
- **RQ2** How can the appropriate heuristics be selected from the generated set during the debugging process?

2 Related Work

LLM for Software Engineering Many studies have adapted LLM for different software engineering application. LLM has been adapted into code completion task [4, 5], code generation [6], code summarization [7, 8], and test program generation [9, 10].

For fault localization task, several frameworks which integrate LLM has been proposed. One approach is zero-shot learning combined with other code artifacts such as log data and functional API to execute the input code [11, 12]. Other approaches include scientific debugging in iterative manners to localize buggy lines [13], and fine-tuned LLM with bidirectional adapter augmentation [14]

Induction approaches on LLM LLM's inductive ability has also been explored, however the scope is mainly limited to instruction and pattern induction leaving heuristics induction remained unexplored. Specifically, Qiu et al. [15] found interesting fact that LLMs are able to induct on rule without knowing how to apply the rule to instances. In contrast, Wang et al. [16] proposed a novel approach by extracting natural language hypotheses and materializing such hypotheses into programs, thereby enabling the verification and refinement of these proposed hypotheses. Similarly, the induction method was utilized by Zhong et al. [17] to develop a framework aimed at distinguishing between two text distributions by generating a set of hypotheses from samples and subsequently re-ranking them to assess their validity.

3 Approach

Problem Statement Given a method/function-level buggy code x, heuristics $\mathcal{H}$ with m heuristic classes $c = \{c_1, \cdots c_m\}$ such that $\mathcal{H} = \bigcup_{i \in c} h_i$, and heuristics class classifier $f : x \rightarrow c$ our objective is to generate potential buggy lines $\hat{y}$ in x, such that $\hat{y} = \bigcup_{i \in f(x)} h(x)$.

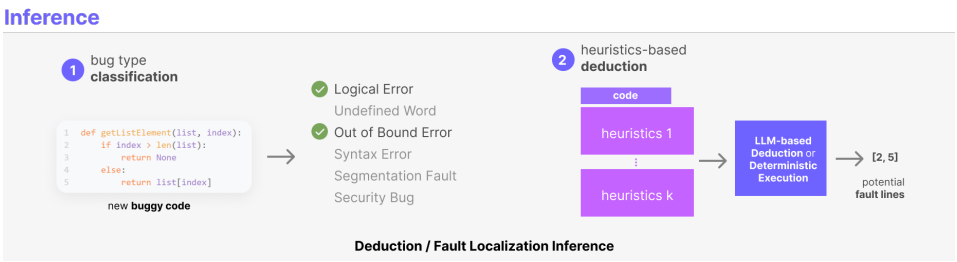


Figure 1: InductFL Inference

The training process will retrieve $\mathcal{H}$ and f, given a code dataset $\mathcal{D} = \{x_n, y_n\}_{n=1}^{N}$ where x_i is a buggy code snippets and y_i is the faulty line(s) of x_i

Two steps will be conducted in the training process: (1) training a bug clustering/classification based heuristic class classification to obtain f and (2) heuristics generation by LLM induction to obtain $\mathcal{H}$ for each heuristic class.

Bug Classification/Clustering The classification model relies on the code instance's embedding representation. Specifically, we employ the `codebert-base` model, a Roberta-based model trained specifically for programming task [18].

We investigated two approaches to guide the deduction process in selecting the appropriate heuristics. The first approach involves a multilabel classification model that maps a given input (a code snippet) to categories of programming errors. The classification layer is trained using cross-entropy loss across 15 distinct bug categories.

The alternative method we examined employs a code-to-bug embedding strategy. Instead of incorporating a classification layer into the embedding model, we suggest adding an embedding layer that directly associates bug representations with the embeddings. This embedding layer is trained using a multilayer perceptron (MLP) autoencoder, with an embedding dimensionality of 512. The training process utilizes a contrastive logistic loss function with random pair sampling:

$$\mathcal{L}(\hat{\mathbf{y}}_1, \hat{\mathbf{y}}_2, l) = \frac{1}{N}\sum_{i=1}^{N} -(1-l)\cdot\log(\|\hat{y}_1^{(i)} - \hat{y}_2^{(i)}\|) - l\cdot\log(1 - \|\hat{y}_1^{(i)} - \hat{y}_2^{(i)}\|)\}$$

where $(\hat{y}_1, \hat{y}_2)$ be the embedding of two instance pairs and l is label determines whether $\hat{y}_1, \hat{y}_2$ are in the same class (1 for different class).

The embedding-based approach offers greater flexibility in selecting heuristic classes, as different heuristic classes can be inferred from the bug embeddings based on the premise that each embedding cluster corresponds to a specific bug type. Conversely, relying on a fixed multi-label classification constrains the number of identifiable bug types.

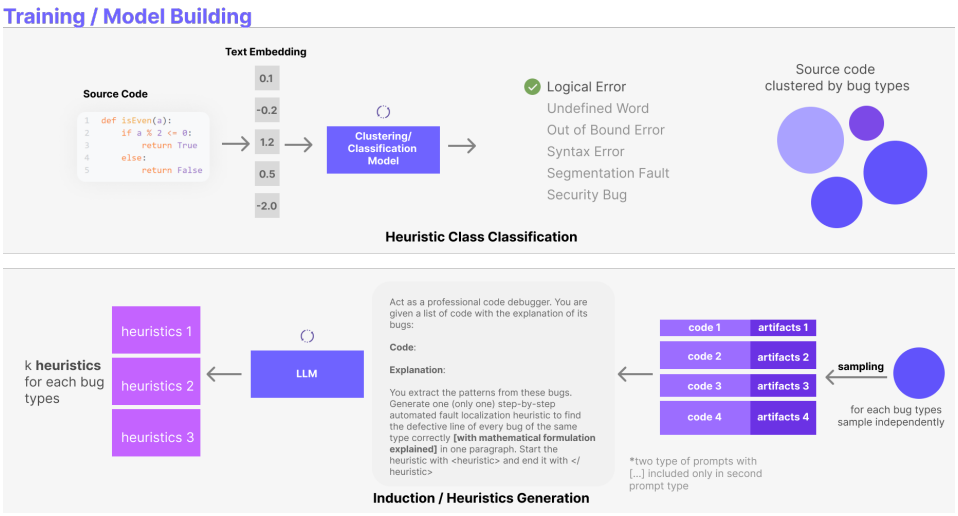


Figure 2: InductFL Training

Heuristic Generation For each designated heuristic class, distinct heuristics are produced. Our objective is to develop heuristics that are relevant, specific, and generalisable since the deduction process seeks to generate a program capable of implementing such heuristics.

We investigate prompt engineering to produce high-quality heuristics by incorporating specific instructions into the prompt, thereby generating precise heuristics with mathematical formulations. We aim to compare the results to verify if prompt injection enhances the ability of LLMs to generate higher-quality debugging heuristics.

4 Experiments

4.1 Data

We employed the DebugBench dataset to train and evaluate the architecture due to its provision of code samples labeled by bug type and accompanied by multiple artifacts. DebugBench includes code

in three languages: C++, Java, and Python 3, and identifies 18 distinct bug types by inserting synthetic errors into source code [19]. For our purposes, we focused exclusively on 15 distinct bug categories, merging instances with multiple bugs into their respective primary bug categories. In addition, for heuristic generations, we only focused on Python program as some bug types are language specific.

4.2 Evaluation method

We explore the architecture in two phases:

- **Induction**: We utilized LLM to assess the heuristics based on three criteria: relevance, precision, and generalizability. Relevance measures whether a heuristic pertains to the data's heuristic class, while precision evaluates the clarity and feasibility of the heuristic for implementation as a deterministic program. Generalizability determines if the given heuristic can be extended to all problems within the same category. To eliminate bias, no information used in heuristic generation was provided during the evaluation.
- **Deduction (Heuristics Selections)**: We assessed the classification model by comparing its accuracy against the dataset's true labels, and evaluate the embedding model through its cluster separability based on those labels.

4.3 Experimental details

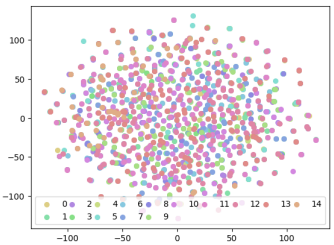
The experiment was conducted in two stages: induction and deduction. During the induction phase, we examined the LLM's ability to generate valuable heuristics that can be executed during the debugging process. In the deduction phase, we investigated approaches to accurately identify the appropriate heuristics to apply, based on the semantics of the input code.

Focus	**Experiment**	**Goal**
Bug Classification	explore two methods (1) classification model (2) embedding based clustering model	create an model that maps input code to relevant heuristic class
Heuristics Generation	prompt engineering and LLM induction	generate clear and generalized heuristics based on the cluster.

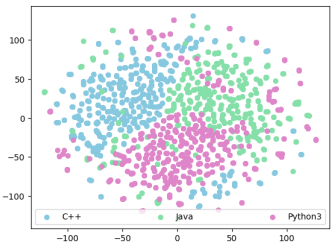
Table 1: Experiment Design

4.4 Results

Embedding Bug Clustering The first exploration that we did is to verify the quality of codeBERT embeddings. We verified that codeBERT embedding low performance on bug clustering task as it achieves very low NMI for bug clustering in the embedding space. Instead, the embedding performs better for clustering the programming language, achieving above 50% NMI score.



(a)



(b)

K-Means Evaluation	NMI	RI
C = 3 (language)	0.51	0.73
C = 15 (bugs)	0.11	0.85
C = 15 (python)	0.12	0.84
C = 15 (java)	0.29	0.85
C = 15 (cpp)	0.05	0.84

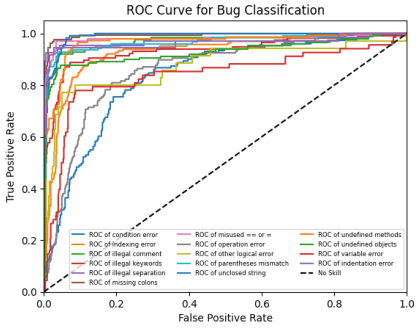
(c)

Figure 3: Embedding Visualization (a) on bug types (b) on programming language and (c) clustering evaluation using codeBERT

We tried to train an embedding model on top of the codeBERT embedding to better encode bug representation by also fine-tuning the weights of the base model. However, the training of such

model does not converge to low loss value, achieving lowest contrastive logloss value of > 0.6. The clustering evaluation on the trained embedding also achieves low NMI score of 0.075, worse than the original embeddings.

Bug Classification We fine-tuned the base codebert model with DebugBench dataset for classifying bug inherent in code snippets input. After 20 epochs of training with $5e-5$ learning rates, we obtained a model that is capable of classifying bug with an accuracy rate of 93% evaluated on independent test dataset.



(a)

Class	AUC	Class	AUC
Condition err.	0.83	Misused ==/=	0.98
Indexing err.	0.93	Operation err.	0.86
Illegal comment	0.97	Other logical err.	0.88
Indentation err.	0.96	Parentheses err.	0.97
Illegal keywords	0.93	Unclosed string	0.99
Illegal separation	0.97	Undefined methods	0.97
Missing colons	0.99	Undefined objects	0.93
Misused ==/=	0.98	Variable err.	0.84

(b)

Figure 4: Evaluation for The Classification Model

Debugging Heuristics We generated 150 heuristics using the first prompting technique (10 for each bug type) and 76 heuristics using the second prompting (5-6 for each bug type). The evaluation of the generated heuristics is as follows:

Bug Type	**Relevancy**		**Generalisability**		**Precision**	
	Prompt 1	Prompt 2	Prompt 1	Prompt 2	Prompt 1	Prompt 2
Condition err.	8.5	9.3	7.9	7.5	6.0	7.7
Indexing err.	9.6	9.4	8.3	8.4	6.7	7.2
Illegal comment	8.6	9.4	7.9	7.8	6	7.8
Indentation err.	9.9	9.8	8.9	8.6	6.6	9.2
Illegal keywords	8.1	10.0	7.4	8.6	6.3	8.4
Illegal separation	7.4	8.2	8.0	7.8	6.6	9.8
Missing colons	9.9	9.2	8.6	8.2	6.9	8.6
Misused ==/=	9.7	9.4	8.6	7.6	6.9	7.8
Operation err.	8.8	8.2	7.7	7.4	5.6	7.2
Other logical err.	8.7	8.2	7.5	7.2	5.3	7.3
Parentheses err.	10.0	9.4	8.5	9.2	6.9	8.8
Unclosed string	10.0	10.0	9.7	9.4	7.0	9.6
Undefined methods	9.4	10.0	8.6	9.6	6.4	9.0
Undefined objects	8.9	9.4	8.3	8.4	6.9	8.0
Variable err.	8.8	7.8	8.2	7.6	5.9	8.0
Average	9.08	**9.2**	**8.27**	8.22	6.39	**8.29**

Table 2: LLM-based Heuristic Evaluation (scoring from 0 to 10)

The results and codes can be accessed at https://github.com/alfonsusrr/CSC4050-121040014-AlfonsusRendy

5 Analysis

5.1 Heuristic Generation Using LLM via Induction Process

Our findings indicate that LLMs can generate relevant heuristics for program debugging. However, we observe that these heuristics are primarily expressed in natural language without specific step-by-step instructions, complicating their implementation. While enhancing LLMs with additional constraints in the prompt increases heuristic precision, this improvement comes at the expense of generalizability, as more precise heuristics tend to be applicable only to specific instances.

We also observe that LLMs can generate more general, relevant, and precise heuristics for syntax errors, such as parenthesis errors, unclosed strings, undefined methods and objects, and indentation errors. However, they struggle with logical and semantic errors, such as condition errors, indexing errors, and operation errors.

5.2 Bug-Based Heuristic Class Classification

Our research finds that training embeddings to represent program bugs is challenging. The pre-trained `codebert-base` embedding primarily encodes syntax information, clustering well across programming languages (Python, Java, and C++). However, we did not observe latent representations for bugs in these embeddings, raising the question of whether an embedding model can be trained to cluster different program bugs effectively.

Our attempt to train bug-based embeddings yielded low-performing results that failed to capture bug representations. We hypothesize three potential reasons for this failure: (1) overlapping code instances across predefined bug types hinder contrastive training and result in unclustered embeddings; (2) the input code lacks distinctive features across bug types, which could be mitigated by incorporating additional artifacts like test logs; and (3) insufficient datasets representative of the 15 bug categories were used, which could be addressed by curating more buggy code samples.

Conversely, applying a classification model on top of the base encoder achieved high accuracy ($>$ 90%) and AUC ($>$ 80) across all bug categories. This demonstrates the potential of integrating a classification model into the *InductFL* framework to select relevant heuristics effectively.

Analyzing the model's performance across various bug types, we identified a lower accuracy in detecting logic-related bugs, including variable errors, operational errors, and conditional errors (AUC $<$ 0.9). This observation indicates that logic-related bugs are not adequately represented when only the source code is provided. Consequently, incorporating additional features is essential to improve the classifier's performance in identifying these bugs.

6 Conclusion, Limitations, and Future Works

We introduced InductFL, a novel framework for automated fault localization (FL) utilizing large language models (LLMs). Our findings suggest that while LLMs can generate relevant debugging heuristics, they still lack the ability to provide detailed and precise instructions. We also demonstrated the potential for selecting relevant heuristics using a classification model.

Our research is limited due to reliance on 15 predefined bug categories and the use of a synthetic dataset that may not capture the complexity of real-world programs. Additionally, we did not evaluate the framework's end-to-end performance since executing heuristics remains challenging. One approach worth considering is using zero-shot FL inference with heuristic injection via LLMs. Future research should involve training and evaluation on more complex code instances and building an end-to-end framework. Developing bug-based embeddings should also explore alternative contrastive learning loss functions like InfoNCE. Further exploration is also needed on how to execute the generated heuristics.

References

[1] Pavneet Singh Kochhar, Xin Xia, David Lo, and Shanping Li. Practitioners' expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 165–176, New York, NY, USA, 2016. Association for Computing Machinery.

[2] Humza Naveed, Asad Ullah Khan, Shi Qiu, Muhammad Saqib, Saeed Anwar, Muhammad Usman, Naveed Akhtar, Nick Barnes, and Ajmal Mian. A comprehensive overview of large language models, 2023.

[3] Long Ouyang, Jeff Wu, Xu Jiang, Diogo Almeida, Carroll L. Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, John Schulman, Jacob Hilton, Fraser Kelton, Luke Miller, Maddie Simens, Amanda Askell, Peter Welinder, Paul Christiano, Jan Leike, and Ryan Lowe. Training language models to follow instructions with human feedback, 2022.

[4] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. Codegen: An open large language model for code with multi-turn program synthesis, 2023.

[5] Raymond Li, Loubna Ben Allal, Yangtian Zi, and et al. Niklas Muennighoff. Starcoder: may the source be with you!, 2023.

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, and et al. Nicholas Joseph. Evaluating large language models trained on code, 2021.

[7] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jin. Deep code comment generation with hybrid lexical and syntactical information. *Empirical Softw. Engg.*, 25(3):2179–2217, may 2020.

[8] Weisong Sun, Chunrong Fang, Yudu You, Yun Miao, Yi Liu, Yuekang Li, Gelei Deng, Shenghan Huang, Yuchen Chen, Quanjun Zhang, Hanwei Qian, Yang Liu, and Zhenyu Chen. Automatic code summarization via chatgpt: How far are we?, 2023.

[9] Max Schäfer, Sarah Nadi, Aryaz Eghbali, and Frank Tip. An empirical evaluation of using large language models for automated unit test generation, 2023.

[10] Zhuokui Xie, Yinghao Chen, Chen Zhi, Shuiguang Deng, and Jianwei Yin. Chatunitest: a chatgpt-based automated unit test generation tool, 2023.

[11] Sungmin Kang, Gabin An, and Shin Yoo. A preliminary evaluation of llm-based fault localization, 2023.

[12] Yonghao Wu, Zheng Li, Jie M. Zhang, Mike Papadakis, Mark Harman, and Yong Liu. Large language models in fault localisation, 2023.

[13] Sungmin Kang, Bei Chen, Shin Yoo, and Jian-Guang Lou. Explainable automated debugging via large language model-driven scientific debugging, 2023.

[14] Aidan Z. H. Yang, Ruben Martins, Claire Le Goues, and Vincent J. Hellendoorn. Large language models for test-free fault localization, 2023.

[15] Linlu Qiu, Liwei Jiang, Ximing Lu, Melanie Sclar, Valentina Pyatkin, Chandra Bhagavatula, Bailin Wang, Yoon Kim, Yejin Choi, Nouha Dziri, and Xiang Ren. Phenomenal yet puzzling: Testing inductive reasoning capabilities of language models with hypothesis refinement, 2023.

[16] Ruocheng Wang, Eric Zelikman, Gabriel Poesia, Yewen Pu, Nick Haber, and Noah D. Goodman. Hypothesis search: Inductive reasoning with language models, 2023.

[17] Ruiqi Zhong, Charlie Snell, Dan Klein, and Jacob Steinhardt. Describing differences between text distributions with natural language, 2022.

[18] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[19] Runchu Tian, Yining Ye, Yujia Qin, Xin Cong, Yankai Lin, Yinxu Pan, Yesai Wu, Zhiyuan Liu, and Maosong Sun. Debugbench: Evaluating debugging capability of large language models, 2024.