N. A. SHAIKH

ATILIM UNIVERSITY 2024

ANALYSIS AND DEBUGGING OF CODE SAMPLES USING LARGE LANGUAGE MODELS

THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

NOMAN AHMED SHAIKH

A MASTER OF SCIENCE
THESIS
IN
COMPUTER ENGINEERING

JUNE 2024

ANALYSIS AND DEBUGGING OF CODE SAMPLES USING LARGE LANGUAGE MODELS

A THESIS SUBMITTED TO
THE GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES
OF
ATILIM UNIVERSITY

BY

NOMAN AHMED SHAIKH

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR
THE DEGREE OF MASTER OF SCIENCE
IN
COMPUTER ENGINEERING

JUNE 2024

Approval of the Graduate School of Natural and Applied Sciences, Atilim University.

_____

Prof. Dr. Ender KESKİNKILIÇ
Director

I certify that this thesis satisfies all the requirements as a thesis for the degree of **Master of Science in Computer Engineering, Atılım University**.

_____

Prof. Dr. Gokhan ŞENGÜL
Head of Department

This is to certify that we have read the thesis ANALYSIS AND DEBUGGING OF CODE SAMPLES USING LARGE LANGUAGE MODELS submitted by NOMAN AHMED SHAIKH and that in our opinion it is fully adequate, in scope and quality, as a thesis for the degree of Master of Science.

_____

Asst. Prof. Dr. Arda SEZEN
Supervisor

Examining Committee Members:

Assoc. Prof. Dr. Meltem ERYILMAZ
Software Eng. Department, OSTIM Technical University _____

Asst. Prof. Dr. Arda SEZEN
Computer Eng. Department, Atılım University _____

Asst. Prof. Dr. Güzin TÜRKMEN
Computer Eng. Department, Atılım University _____

Date: *03-06-2024*

I hereby declare that all information in this document has been obtained and presented in accordance with academic rules and ethical conduct. I also declare that, as required by these rules and conduct, I have fully cited and referenced all material and results that are not original to this work.

NOMAN AHMED, SHAIKH :

Signature :

ABSTRACT

ANALYSIS AND DEBUGGING OF CODE SAMPLES USING LARGE LANGUAGE MODELS

Shaikh, Noman Ahmed

M.S. Computer Engineering

Supervisor : Asst. Prof. Dr. Arda Sezen

June 2024, 76 pages

Code debugging and analysis is a challenging task. Specially the task of manual fault localization (FL) is resource-consuming and requires significant effort to identify the root cause of the fault. In this thesis, test-free, automatic line-level fault localization using large language models is explored. Different bidirectional attention-based code-understanding pre-trained large language models (CLMs) are used and adapter tuning is performed to fine-tune the CLM to output line-level faultiness scores of the input code. The resulting model is called FLICoder. Multiple FLICoder models with different settings are trained and the impact of various aspects of its architecture on its overall performance is investigated. The FLICoder model is also compared with the baseline LLM-based FL solution. The baseline is outperformed by FLICoder by 25% - 52%.

Keywords: code debugging, line-level, test-free, fault localization, large language models.

ÖZ

BÜYÜK DİL MODELLERİ KULLANARAK KOD ÖRNEKLERİNİN ANALİZİ VE HATA AYIKLAMA

Shaikh, Noman Ahmed
Yüksek Lisansı, Bilgisayar Mühendisliği
Tez Yöneticisi : Dr. Öğr. Üyesi Arda Sezen

Haziran 2024, 76 sayfa

Kod hata ayıklama ve analizi zorlu bir görevdir. Özellikle otomatik olmayan hata yerelleştirme görevi kaynak tüketir ve hatanın kök nedenini belirlemek için önemli bir çaba gerektirir. Bu tezde, büyük dil modellerini kullanarak testsiz, otomatik satır seviyesi hata yerelleştirme incelenmiştir. Çalışmada, çift yönlü dikkat tabanlı mekanizma ve kod-anlama için önceden eğitilmiş büyük dil modelleri kullanıldı. Aynı zamanda büyük dil modellerinde girilen kodun satır seviyesi hatalılık puanlarını çıktı olarak vermesi için adaptör ayarlaması yapılmıştır. Ortaya çıkan model FLICoder olarak adlandırıldı. Farklı ayarlarla birden çok FLICoder modeli eğitildi ve mimarisinin çeşitli yönlerinin genel performans üzerindeki etkisi incelendi. FLICoder modeli ayrıca temel LLM tabanlı hata yerelleştirme çözümü ile karşılaştırılmış olup, FLICoder modelinin %25 - %52 iyileştirme gösterdiği tespit edilmiştir.

Anahtar Kelimeler: Kod Hata Ayıklama, Satır Seviyesi, , Hata Yerelleştirme, Büyük Dil Modelleri.

To my beloved family, specially my dad, Dr. Nadeem Ahmed Shaikh.

ACKNOWLEDGMENTS

I would like to express my gratitude to my supervisor Asst. Prof. Dr. Arda Sezen for his utmost support, valuable time, and thoughtful guidance throughout the research.

I shall also thank my friends and family for supporting me and encouraging me throughout this journey.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

FL	Fault Localization
LLM	Large Language Model
PLM	Pre-trained Large Language Model
CLM	Code-understanding Large Language Model
FLICoder	Fault-Localizing Intelligent Coder
GPU	Graphics Processing Unit
ROC	Receiver Operating Characteristics
AUC	Area Under the Curve

CHAPTER 1

INTRODUCTION

Debugging and analysis of programming code is a challenging task with a vast amount of time and human developer resources spent on it [1]. In the realm of debugging, the tasks of fault localization (FL) and program repair have been studied widely in order to be automated [2].

In the past few years, after the development of deep learning techniques, there have been continuous efforts to apply them for FL and automated program repair (APR) [3]. Subsequently, after the large language models (LLMs) have demonstrated a great capability of code understanding recently [4], it is being widely researched to effectively incorporate them for FL and APR [5].

In this chapter, we discuss in detail the background of this domain, diving deeper into the basics of LLMs, their code understanding capabilities, and their incorporation for the debugging and analysis of code samples.

Thereafter, the details of the proposed research, research objectives, and the research questions are presented.

1.1 Background

1.1.1 Large Language Models

Language plays an important role in human life, from trivial day-to-day tasks to complex communication paradigms impacting our lives greatly in every possible

aspect [6]. However, language is not only crucial for humans anymore, but for machines as well. As more intelligent machines emerge day by day, language is the backbone of their revolutionizing intelligence. That's because language is the key aspect for the design and proper behavior of a machine, in the form of a programming language [7]. In addition to that, in order to establish superior intelligence, communication is an inevitable characteristic including both communication between different parts of the machine itself and communication with the human users [8].

Considering the factors mentioned above, there has been a tremendous amount of research to develop a common communication medium for human-machine interaction, not only recently but for decades [9]. To enable the machines to understand a communication medium that is easily and naturally understandable for humans [10]. That medium is natural language [11].

Finally, a breakthrough happened in this research with the development of large language models (LLMs). One of the earliest examples of large language models can be traced back to 1966 in the form of ELIZA, a program written by Joseph Weizenbaum at MIT [10]. However, from the modern-day LLMs, the first large language model to gain the most attention was BERT introduced by Google in 2018-2019 [12]. Thereafter, more large language models started to appear [13], and the one that truly marked a revolution in the research and development in this domain, was ChatGPT (a variant of the Generative Pretrained Transformer (GPT) [14], [15] model), a versatile model developed by OpenAI in 2022, trained on a huge amount of language data, that has the capability to be prompted in natural language for information retrieval in almost any topic that comes to mind [10], [16]. Figure 1.1 shows the trend in the emergence of LLMs from 2019 to 2023 [13].

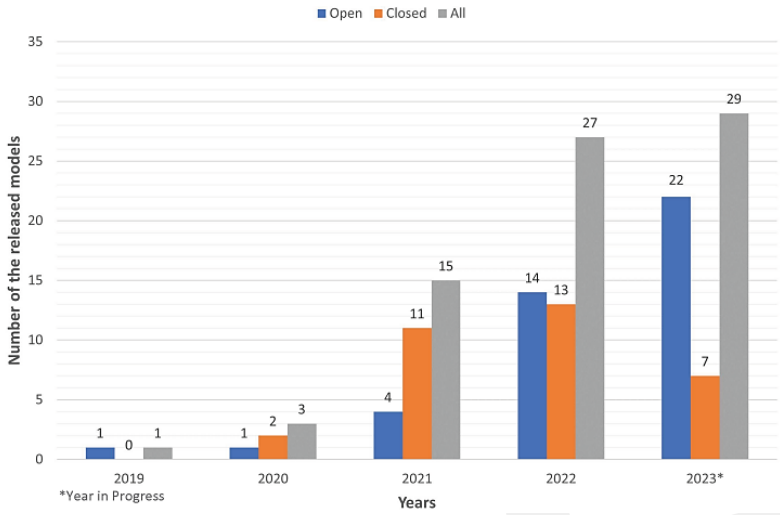

Figure 1.1 The Trend in the Number of LLMs Introduced Over the Years [13].

The recent breakthrough in developing these well-performing large language models is attributed mainly to deep learning techniques and the development of advanced neural network architectures like transformers. In addition, the progress and advancement in computing capabilities also played an important role here as significant computational power and optimization are required to effectively train and utilize these large language models. Further, the accessibility of the training data available from the internet is the base ingredient that enabled the development of such models [17].

Large language models bring to the machines the ability to handle complex tasks like conversational interactions, summarization, information retrieval, and translation. However, these tasks were also being performed in some capacity by using other techniques before LLMs. The highlight about LLMs is that on certain evaluation benchmarks, they can perform such tasks with human-level performance [18], [19].

Pretrained large language models have demonstrated exceptional capabilities to generalize for text generation and understanding tasks being trained on a large amount of data even only in a self-supervised manner [20], [21]. This characteristic motivated the researchers to further train the pre-trained LLMs on even a larger amount of data

and it was discovered that the LLMs' generalization performance increases significantly with scaling the model itself and the training data. Further, the performance of LLMs also increases significantly when trained and fine-tuned for a specific task with task-specific large datasets [13].

Therefore, there has been a vast development of LLMs fine-tuned for different specific domains of data. Currently, the LLMs have been used in including but not limited to the fields of medicine, education, science, maths, law, finance, robotics, coding, etc [13].

1.1.1.1 Code Language Models

Pretrained large language models have the capability to understand and work with code as coding data is often part of their pre-training [22], [23]. However, that is not their focused specialty, and as mentioned above, their task-specific functionality can be improved when further fine-tuned on the respective task-specific data [24], [25]. Code language models (CLMs) are large language models fine-tuned on code data, usually taken from online public repositories like GitHub [4], [26]. Code language models have the ability to efficiently work with programming codes [22], [27]. The tasks usually include code generation from natural language description, code summarization and explanation given a programming code, code translation from one programming language to another programming language, and code completion [28] [29].

CLMs tend to exhibit code representation learning (also known as code embedding) which aims to encode code semantics into vector representations. The main used architecture for CLMs to perform code embedding is the encoder-decoder architecture. The code is first fed to the encoder network to convert it to a vector representation. The vector representation is then fed to the decoder network to perform task-specific prediction [28]. The studies have also shown the ability of the CLMs to utilize syntactic information of code as well [30]. However, there are also LLMs and CLMs with encoder-only and decoder-only architecture [31], [32].

1.1.2 Transformers

Modern large language models are based on the transformer model [33] introduced by Google in 2018. The Transformer architecture revolutionized the world of natural language processing (NLP) and resulted in the development of LLMs that outperform almost all previous deep learning architectures for NLP. They possess the capability to embed deeper details of the input and learn to pay attention to the most important and related parts of the input, which is what makes these models perform better than the other older techniques [33].

1.1.2.1 Tokenization

All neural network models, including Transformer, dealing with text processing work with the concept of tokenization [34]. Tokenization is the process of parsing the text into non-decomposable units referred to as tokens. Tokens can be characters, symbols [35], subwords [36], or words depending on the task, size, and type of the model [37].

1.1.2.2 Self-Attention in Transformers

The Transformer is based on an attention mechanism referred to as self-attention. Let $c = \{w_1, w_2, \ldots, w_n\}$ represent a code sample with an n length of the sequence of tokens. A Transformer model is based on L layers of Transformer blocks to represent the code sample as textual representation at different levels, $H^l = [h_1^l, h_2^l, \ldots, h_n^l]$, where l denotes the number of layer. For every layer, the representation of the layer H^L is calculated by the Transformer block of the l-th layer $H^l = Transformer_l(H^{l-1}), l \in \{1, 2, \ldots, L\}$ [28], [33].

To aggregate the previous layer's output vector, multiple self-attention heads are used in each Transformer block. With the input c, a collection of weights is assigned to each token w_i in the input:

$$Attn(w_i) = (\alpha_{i,1}(c), \alpha_{i,2}(c), \ldots, \alpha_{i,n}(c)) ,$$

Here $\alpha_{i,j}(c)$ is the attention paid by w_i to w_j. The computation of attention weights involves taking the scaled dot-product of the *query vector* of w_i and the *key vector* of w_j, and then applying a softmax. In vectorized computing, the typical mechanism of attention can be defined as the weighted sum of the value vector V, utilizing the query vector Q and the key vector K:

$$\mathrm{Att}(\mathrm{Q},\mathrm{K},\mathrm{V}) = \mathrm{softmax}\left(\frac{\mathrm{QK}^T}{\sqrt{d_{model}}}\right).\mathrm{V},$$

Here d_{model} denotes each hidden representation's dimension. The vectors V, K, and Q are the previous hidden representation's mappings by different linear functions, i.e. $\mathrm{V} = \mathrm{H}^{l-1}\mathrm{W}_V^l$, $\mathrm{K} = \mathrm{H}^{l-1}\mathrm{W}_K^l$, and $Q = H^{l-1}W_Q^l$, respectively. Lastly, the final contextual representations $\mathrm{H}^L = [\mathrm{h}_1^L, \ldots, \mathrm{h}_n^l]$, are produced by the encoder and are outputted at the last Transformer block [28].

1.1.2.3 Encoding Positions

Now, to make use of the sequential tokens' order, the "positional encodings" are added to the input embedding.

$$\mathrm{w}_i = e(w_i) + pos(w_i)$$

Here e represents the word embedding layer and pos represents the positional embedding layer. The position of the code token is typically implied by positional encoding using the sine and cosine functions. It is to be noted that recent studies show that the positional encodings might not matter for the state-of-the-art decoder-only models [13], [28], [38].

1.1.3 Pre-Training of Language Models

As an example, BERT can be analyzed [12]. Before its pre-training, it takes two segments as input. These segments are defined as $c_1 = \{w_1, w_2, \ldots, w_n\}$ and $c_2 = \{u_1, u_2, \ldots, u_m\}$, where 'n' and 'm' represent the lengths of the two segments [28].

These segments are not just joined together. They are connected by a special token known as a separator token `[SEP]`. Additionally, each sequence of tokens starts with a classification `[CLS]` token and ends with an end-of-sequence `[EOS]` token [28].

So, each training sample is represented as a sequence that begins with `[CLS]`, followed by the first segment, the `[SEP]` token, the second segment, and finally the `[EOS]` token. A training sample's input can be formulated as follows:

$$s = [\text{CLS}], w_1, w_2, \ldots, w_n, [\text{SEP}], u_1, u_2, \ldots, u_m, [\text{EOS}] .$$

The input is then presented to the Transformer encoder. BERT's pre-training phase has two self-supervised learning objectives: *masked language modeling* (MLM) and *next sentence prediction* (NSP).

In masked language modeling, tokens from an input sentence are randomly selected and replaced with a `[MASK]` token. Typically, 15% of the input tokens are selected for potential replacement. After that, the model is trained to predict the original tokens based on the context [12]. Recently, self-supervised learning using masked language modeling has become prevalent in natural language understanding and generation [12], [39], [40], [41], [42], [43].

Next sentence prediction is a binary classification task predicting whether two segments are consecutive. Training involves positive and negative examples. Consecutive sentences form positive examples, while paired segments from different

documents form negative examples. Figure 1.2 shows a common Transform-based language model pre-training paradigm.

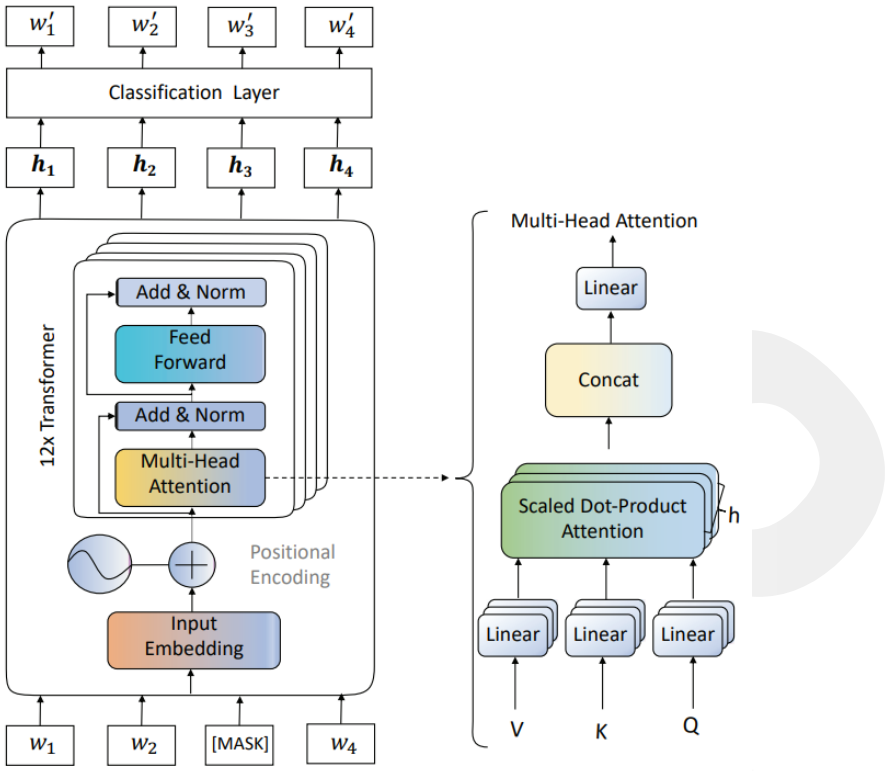

Figure 1.2 Typical Pre-Training of a Transformer-Based Language Model [4], [28].

In software engineering, several pre-trained code language models (CLMs) have been proposed for program comprehension, refer to section 2.1.1 for more details. In the context of CLMs, a training objective of only the next token prediction, allows for code completion, and not for code infilling. Such models can also be referred to as left-to-right language models. A few examples of this [27] are CodeGen [31], Codex [4], GPT-Neo [44], GPT-J [45], and GPT-NeoX [46]. On the other hand, a training setting with masked language modeling, or masked span prediction, allows the model to perform code infilling as well. A few examples of this are the InCoder [32], CodeBERT [47], and CuBERT [48].

1.1.4 Fine-Tuning Language Models

As discussed above, LLMs can be pre-trained on very versatile data [49]. After the pre-training, although they can perform and generalize well for a variety of tasks [14], [15], [16], [50], to incorporate them for a specific task, they can be fine-tuned for that task, in order to increase the task-specific performance and accuracy [24], [25], [41]. There are different techniques and methods for fine-tuning large language models [13].

1.1.4.1 Transfer Learning

Transfer learning is the most straightforward fine-tuning method. For transfer learning, we train the pre-trained language model further on a large amount of task-specific data and the training method and objective remain similar to the pre-training method described in section 1.1.3 [51], [52]. However, transfer learning is a computationally expensive method of fine-tuning as it requires training the whole large language model with all its parameters which are usually in millions and billions. Therefore, it requires working with high-powered GPUs for long periods required for training. Another limitation in transfer learning is that it requires sufficiently large datasets compatible with the size of the LLM.

1.1.4.2 Instruction-tuning:

This fine-tuning is performed to tune the model's output format to respond to the user queries more efficiently. In this method of fine-tuning, the LLM is presented with a set of an instruction and an expected input-output pair. Instruction is text in plain natural language that guides the model to respond based on the prompt and the input. The input-output pair can be task-specific, for example, for CLMs, it can be programming code. Zero-shot generalization and downstream task performance can be improved with instruction-tuning [13]. The limitation of such a training method is that it requires carefully curated instruction and input-output pairs, which are not readily available in most cases and have to be created manually. Also, the format of

the instructions and the input-output pairs can be different from one CLM to another. Therefore it can be a labour-intensive fine-tuning method.

1.1.4.3 Adapter Tuning

This is a kind of *parameter-efficient tuning*, here we add extra layers of encoder-decoder architecture-based network in sequence or parallel to the attention and feed-forward layers of the transformer block. Then we fine-tune only these layers, and the rest of the model is kept frozen [13]. This kind of fine-tuning can be used in case the task-specific data is small and training the whole LLM is not suitable. So, we can train a smaller network (the adapter layers) with the limited available data. This method of fine-tuning can also be helpful if you want to utilize the *language understanding* capabilities of the LLM but you desire the output to be in a different format, which can be other than text for example. Here one can use the hidden representation output from the LLM and using that output, the adapter layers can be trained for a different objective. For example, Yang et al., [53] trained the adapter layers for fault localization in programming code. They trained the adapter layers to output suspiciousness score probabilities for each line in the input code.

1.1.5 Debugging and Analysis of Programming Code

Code debugging and analysis is an important field of software engineering. As long as we keep working with programming codes we will have to be dealing with debugging and analysis of the code for proper working and for improving quality. There is a significant amount of time every developer spends on debugging in his daily work routine. Debugging and analysis can be divided into two categories, fault localization and program repair.

1.1.5.1 Fault Localization

Fault localization (FL) is the first step in debugging. It is the task of determining exactly where the fault lies in the program being debugged. Fault localization can be

performed with different precision, like, line-level, statement-level, module-level, or file-level. Studies show that fault localization, although can be trivial and quick in some cases, in other cases, can be extremely time-consuming and difficult. As much lower level we go, FL can become more and more challenging. If fault localization is performed manually, a large amount of time and resources can be consumed. Therefore, there have been continuous efforts to make fault localization less manual and more autonomous as much as possible [54], [55], [56], [57].

Traditional Automated Fault Localization

In general terms, current Fault Localization (FL) methods merge or utilize both static and dynamic program analysis data to calculate a score that represents the likelihood of a program component contributing to a specific bug. Techniques like Tarantula [58] or Ochiai [59], which are Spectrum-Based Fault Localization (SBFL) approaches, employ statistical analysis on the coverage data of successful/failed tests to determine the suspiciousness of code elements. SBFL is heavily dependent on test coverage, making it less suitable for data-driven defects; it is also sensitive to the characteristics of the underlying test suite, such as coverage and the number of passing and failing tests [54]. Mutation-based Fault Localization (MBFL) methods, such as FIFL [60] or Metallaxis [61], also examine test case behavior to pinpoint faults, but they use mutation analysis to evaluate the specific impact of certain code lines on test results. Despite their effectiveness, MBFL methods are computationally demanding and their performance can vary greatly [62].

Machine Learning-Based Fault Localization

Recent progress in Machine Learning-based Fault Localization (MLFL), such as DeepFL [55], DeepRL4FL [63], and GRACE [56], employs machine learning to associate code, test, or execution characteristics with the probability of a program component being faulty. MLFL methods learn to identify erroneous lines of code using information like suspiciousness scores from existing SBFL and MBFL methods (for instance, TRANSFERFL [64]), features indicative of fault-proneness like code

complexity metrics (like in DeepFL [55]), or the test coverage matrix (as in DeepRL4FL [63]), among other things. These strategies highlight the potential of increasingly robust machine learning models in aiding debugging tasks. Large language models are also machine learning-based models, using them for the task of automated fault localization is discussed in the next section. Studies report that LLM-based FL techniques outperform non-LLM-based FL techniques [5], [53], [65].

Fault Localization with Large Language Models

Certainly, Deep Learning (DL) has demonstrated potential in various code-related tasks, including program synthesis [31], [66]. Large Language Models (LLMs) have been demonstrated to be the most effective DL models in both natural language and code-related tasks [4], [5], [49], [65]. These models, which train billions of parameters on extensive training data, are highly flexible and powerful text generators.

LLMs, specially CLMs are useful for code generation due to their training on a vast amount of publicly available code. This suggests that these models can be utilized for specialized development tasks [4].

Researchers have used LLMs, specially CLMs, for fault localization as is, by providing the model with the appropriate prompts [50], [67], [68]. However, as discussed above in section 1.1.4, there is a potential to achieve significantly higher performance by fine-tuning the models for specific tasks, in this case, fault localization. Studies have been performed to utilize LLMs for FL with different settings and report them to outperform the non-LLM-based FL techniques [53], [67].

However, a notable characteristic of LLMs is that their performance consistently improves with the scale of their computational budget, which is a function of the model and training data size [69]. For example, LLM performance on program synthesis benchmarks increases linearly with the log scale of the number of parameters in the model [27]. This indicates that there is significant performance potential for software engineering tasks by utilizing the largest publicly available language models.

However, most existing work either trains small models from scratch [55], [63], [70] or fine-tunes medium-sized models [64], not fully leveraging the scale of state-of-the-art LLMs. On the other hand, researchers tend to use the LLMs as is for FL localization. Which does give better results compared to the non-LLM-based techniques [71]. But they are missing the potentially better performance that can be achieved by fine-tuning the large-sized LLMs

This is partly because LLMs are not immediately suitable for coding tasks that do not involve code generation, such as fault localization. Many state-of-the-art LLMs for code are trained to generate code in a left-to-right manner, predicting each token from its preceding context [4], [31], [46], [72]. It is hypothesized that models trained in this manner are less suitable for token-level discriminative tasks, like line-level fault localization, as the representation for any given token is only conditioned on the context to the left [53]. Recently, there has been a development of masked span prediction-based state-of-the-art CLMs as well [30], [73], [74], and studies show them already performing well in fault localization with only the pre-trained model as well [5], [32]. Therefore, there is a high potential for better results with fine-tuning.

1.2 Motivation and Research Gap

As discussed above there lies a strong potential for the LLMs to be used for code debugging and analysis. The reason is the code-understanding capabilities of the recently developed code-understanding large language models (CLMs). However, the CLMs are not originally pre-trained for the task of code debugging, specially, fault localization (FL), but rather code generation or infilling.

Nevertheless, researchers in recent years have tested the pre-trained CLMs directly for the task of code debugging and fault localization by applying appropriate prompt engineering and have observed results outperforming the non-LLM-based fault localization techniques. However, as we have discussed above in section 1.1.4, the LLMs have a great tendency and capability to perform better when fine-tuned for downstream tasks.

As observed from the literature review, fine-tuning LLMs for the task of fault localization has not been sufficiently explored by researchers. This could be due to the costs associated with fine-tuning the entire LLM. Yet, as discussed in section 1.1.4, there are cost-effective and parameter-efficient fine-tuning methods, such as adapter-tuning. These methods can yield the desired results without the need to train the whole LLM. Despite this, the literature survey revealed that many researchers have not yet discovered the potential of parameter-efficient tuning for FL tasks.

From the mentioned studies, only one research was found that performs such fine-tuning for the task of line-level fault localization, done by Yang et al., [53]. They use a pre-trained CLM and perform adapter-tuning on it to obtain line-level faultiness scores on the input code samples. They present the results outperforming the non-LLM-based FL techniques.

However, some research opportunities were obtained from their study and the literature that is investigated in this thesis study. Firstly, they used only one LLM model, CodeGen (although different sizes but from the same family). It is vital to explore the technique with different LLMs as each LLM can have different pretraining setups which can result in varied performance. Secondly, they used the CodeGen model as the pre-trained CLM. CodeGen has a unidirectional attention mechanism and attends to only the prefix of input at a time [31]. Therefore, it lacks the capability to have the full context of the complete input code sample. Using a CLM with a bidirectional attention mechanism is expected to give better results.

On top of these, through the literature review, a research gap is also recognized in *test-free* fault localization on a bigger scale. Even though some previous studies have been identified to be performing test-free FL, the research is limited and it has not been explored on a wider level.

1.3 Research Objectives

The target and scope of this thesis are based on the intention to achieve the objectives mentioned below:

- Explore line-level test-free fault localization using code language models (CLMs).
- Explore the impact of using different CLMs.
- Compare the line-level FL performance of a CLM with a bidirectional attention mechanism versus a CLM with a prefix-only attention mechanism.
- Using adapter-tuning, a parameter-efficient fine-tuning method, to fine-tune the CLM to output line-level faultiness scores of the input code sample.
- Explore the impact of the adapter model dimensions on the adapter-tuning of the CLM.
- Explore the impact of performing fault localization on faulty/buggy code samples in different programming languages, namely, Java and Python.

1.4 Research Questions

Based on the discussed research gap, motivation, and research objectives, Multiple adapter-tuned, CLM-based models for line-level fault localization were trained. We call these models FLICoder or Fault Localizing Intelligent Coder models. The following research questions are investigated for which we would like to find the answers in this study.

RQ1: How does using different CLMs affect the fault localization performance of FLICoder models?

RQ2: How does the size of the used CLM affect the performance of the FLICoder models?

RQ3: How does using different sizes and dimensions of the adapter model affect the FLICoder's performance?

RQ4: How well do the FLICoder models perform on different datasets of different programming languages, namely Java and Python?

RQ5: How does using a bidirectional attention-based CLM affect the FLICoder's performance compared to the prior approaches that used a prefix-only CLM?

CHAPTER 2

LITERATURE REVIEW

In this chapter, the literature reviewed for the thesis is discussed in detail. First, the code debugging and analysis techniques that utilize large language models are discussed. Then, the code debugging and analysis techniques that utilize other deep learning-based techniques without the use of large language models but rather utilizing other neural networks like convolutional neural networks, and sequence-to-sequence architectural models like recurrent neural networks and transformers are discussed. Lastly, the widely used benchmark datasets for code debugging and analysis tasks like fault localization and automated program repair are discussed.

Under the code debugging and analysis with large language models section, a deep dive into the topics of widely used code-understanding large language models, which can be referred to as code language models, is taken. The latest code debugging and analysis techniques with large language models that do not require any test cases and can directly localize faults and perform automated program repair given the buggy code only as the input are then discussed. Lastly, the works that utilize and require test cases and execution results in order to perform code debugging and hence require the input code to be complete which can be compiled successfully are discussed.

2.1 Code Debugging and Analysis with Large Language Models

2.1.1 Code Language Models

Xu et al., [27] systematically evaluate code-understanding large language models, which can be referred to as code language models (CLMs). Their motivation for this study is that although there has recently been a fast development in CLMs that show

great promise for code completion and synthesis, many of the state-of-the-art CLMs (like Codex [4]) are not fully open-sourced and therefore lack some detailed information about their model and data design decisions. In the study [27], their target is to explore different CLMs and their capabilities to fill in this information gap.

They evaluate Codex, GPT-J [45], GPT-Neo [44], GPT-NeoX-20B [46], and CodeParrot [72] across different programming languages. They also fill a gap in open-source CLMs' availability and release a new model called PolyCoder with 2.7B parameters based on GPT-2 [15]. They report Codex to be performing the best among all compared models.

Codex is a large language model from OpenAI. It is based on the GPT-3 [14] architecture and trained on code samples in multiple languages to enable code understanding. They released multiple different-sized versions of the model, 12B being the biggest one. Although it is trained on multiple programming languages, they separately tested and benchmarked the model for Python as well. Codex has been used in multiple studies and shows promising results [27], [50], [65], [67].

GPT-J and GPT-Neo are medium-sized models, and GPT-NeoX-20B is a large-sized model, based on GPT-3. Although they have been used in different studies, they do not show that much promising results [5], [27], [65], specially compared to the bigger state-of-the-art models, like Codex.

CodeParrot is a HuggingFace project. It is a small-sized model with only 1.5B parameters, based on GPT-2 architecture. Xu et al., [27] test and compare it with other language models. Although it gives results comparable to some of its similar-sized models (like GPT-J and GPT-Neo), it is outperformed by the state-of-the-art model, Codex of similar (2.5B) or smaller (300M) size.

Jiang et al., [5] also evaluated multiple CLMs for the purpose of automated program repair. They tested PLBART [73], CodeT5 [74], CodeGen [31] and InCoder [32].

PLBART is also a small-sized model, that consists of 140M parameters. It is based on the BART-base model [20]. BART is a natural language (NL) model, and PLBART (Programming Language BART) is trained on top of BART on Java and Python code along with their NL (English language) summary. The study of Jiang et al. [5], shows that while PLBART shows comparable results, other state-of-the-art models like CodeGen and InCoder outperform PLBART in the task of APR.

CodeT5 is based on the T5 [51] from Google. CodeT5 was trained on multiple programming languages like, Java, Python, PHP, C, CSharp, Ruby, JavaScript, etc. CodeT5 has different-sized versions available as CodeT5-small (60M parameters), CodeT5-base (220M parameters), and CodeT5-large (770M parameters).

CodeGen is one of the state-of-the-art models used in multiple studies and shows promising results [5], [53]. It is an open-source model. CodeGen is available in different sizes, 350M, 2B, 6B, and 16B parameters. In addition to releasing different sizes of the model, they also released different versions of the models, based on training data. They CodeGen-NL, CodeGen-Multi, and CodeGen-Mono. CodeGen-NL is trained only on the THEPILE [75] dataset, containing mostly English text. CodeGen-Multi was in addition, trained on different programming languages like Java, C++, C, Python, JavaScript, and Go. Lastly, CodeGen-Mono specifies that the model was further pre-trained on a huge Python dataset collected from GitHub.

It is to be noted that, CodeGen is trained for downstream code generation tasks and is unable to perform code infilling. CodeGen only attends to the prefix of the code line at any given instance [5]. Therefore, if it is used for such tasks as line-level fault localization or debugging, some additional Transformer layers, with attention mask spanning the whole length of the input code, including both the prefix and the suffix [53].

InCoder [32] is also a recently (2023) released open-source, code-understanding large language model. It was released by Meta. The model was pre-trained on code data from GitHub, GitLab, and StackOverflow posts along with big data of English text.

The code data consisted of 33% Python, 26% JavaScript, 10% C/C++, and some small percentages of other programming languages. Almost one-third of the pre-training data is Python, which makes this model a good candidate for Python-based tasks. The developers released two different size versions of InCoder-1B and InCoder-6B.

Another important highlight of the InCoder model is that it was trained for masked span prediction which makes it capable of code-infilling tasks. It has a bidirectional context window and attends to both the prefix and the suffix of the code which makes it suitable for both code generation and code infilling.

2.1.2 Test-Free Code Debugging and Analysis with LLMs

Jiang et al., [5] did a comprehensive study on applying large language models (LLMs) for the task of automated program repair (APR). They tested 10 code-understanding large language models (CLM) for this study. First, they tested the CLMs without any fine-tuning and then they tested again after fine-tuning the models on four APR benchmark datasets. They do not perform any architectural modification on the model. Rather just use appropriate prompts for each model after carefully studying their documentation.

Their tested models are PLBART, CodeT5, CodeGen and InCoder. Developers of all these models released different versions of the models with varying sizes. So in this study, all respective versions were used.

They also addressed the issue of data leaking by curating an APR benchmark dataset themselves, named HumanEval-Java [4]. The leaking issue can be described as the fact that as the LLMs are trained on a huge amount of data, there is a probability that they have seen and have been trained on the benchmark datasets as well.

In the study [5], the methodology of the tests includes firstly prompting the models with the buggy line removed and special formatting depending on the type of model. For the models capable of performing infilling tasks, the researchers inserted special

characters in place of the buggy line, indicating to the model, the place where the patch needs to be inserted. The special characters work as the mask for the model. On the other hand, for the models capable of performing only left-to-right code generation, they removed the code from the buggy line till its end, and the model performed code completion on the input code.

For the second kind of test, they included the buggy line in its place as a comment along with the heading “buggy line:”. With this prompting setup, they first performed the tests with the pre-trained LLMs and then fine-tuned them with the same prompting setup.

That study provides insight into the capabilities of the CLMs, that even without any fine-tuning, they have the capability to fix more bugs compared to the state-of-the-art deep learning (DL)-based APR techniques. According to the findings, the InCoder model fixes the most bugs. To be specific, it fixes 72% more bugs than the compared DL-based APR techniques. While, they report that with fine-tuning, they can fix 46%-164% (based on the specific CLM) more bugs than the existing DL-based approaches. Their work also concludes that the CLMs do not make use of the buggy lines (input as comments in the prompts) as is, without any fine-tuning. On the other hand, the fine-tuned CLMs can potentially rely too much on the buggy line.

Xia et al., [65] also explored using LLMs for automated program repair. The study evaluates different kinds of CLMs as is without any fine-tuning and compares the results with non-LLM, DL-based APR techniques. The researchers also compare the CLMs with each other and study the effects of the size of the CLM and their attention mechanism; whether they utilize only the prefix to the buggy line or both the prefix and the suffix. The models that attend to only the prefix are called generative models while the models that take into account the suffix as well, are referred to as the infilling models. They evaluate nine different models, namely GPT-3, Codex, CodeT5, and InCoder by using various versions (based on size) of these models.

The researchers' methodology consists of two styles, one for the chat models and the other for the code-specific models. For the chat models, they prompt the model first with a couple of programs with bug and their bug-fixed version along with comments indicating the required task of performing a bug fix and the location of the bug. This is used to explain to the model, the format of the input and output. For the code models capable of infilling, they simply input the code with the buggy line removed, and the mask placed in its location. 5 different benchmark datasets are used in that study to evaluate the proposed techniques.

Their conclusion includes, firstly, the size of the LLM affect greatly the number of correctly generated fixes. However, they also emphasize that pretraining has an important effect. For instance, the Codex model (12B), although smaller than GPT-NeoX (20B) (a GPT-3-based model), performs better. That is because the Codex is a model specifically pre-trained for code generation tasks. Secondly, they compare two versions of Codex, one able to do only generation, and one, which is able to perform code infilling as well. The infilling version fixes 40% of the bugs, compared to the generation-only version, which fixes only 28% of the bugs. Lastly, the researchers also report similar to Jiang et al, [5], that the InCoder model shows better results than the other tested CLMs.

Yang et al., [53] explored and developed a new approach for line-level fault localization using large language models (LLMs). The highlight of their technique is that it does not require any tests to localize the faults. Rather, a program in any stage, whether complete or incomplete, and hence not ready for a test, can be put as the input to their model, and the model can predict the suspiciousness of a line to be faulty or not.

Test-free fault localization makes the approach applicable for code debugging at any stage of development and the code does not have to be complete or fully executable, as it is required for performing tests.

The proposed model in that study consists of a pre-trained, code-understanding large language model, and a few more layers of a transformer model plus a binary prediction layer that the researchers train for their tasks.

The researchers used the pre-trained large language model to understand the code samples and obtain the hidden states of the model's output that represent each line of code. Then they feed these hidden states to the self-trained and self-developed transformer model in order to further modify the hidden states for the main task of fault-localization. That is done because the processing by the pretrained model is not performed for this specific task and hence can not predict faultiness without fine-tuning, or using some extra layers on top of it, as the researchers did in this work, to perform the task of fault localization. Finally, the last layer is a binary prediction layer which takes as input the output of their self-trained transformer model and outputs the suspeciousness scored for each line of the input code.

An important reason why the researchers needed to train extra transformer layers on top of their pre-trained large language model is the underlying attention mechanism. The model they used is CodeGen which is a left-to-right code generation model. Essentially, what it means is that it attends to only the prefix of code at any given point and not the suffix. It makes it not perfectly suitable for the task of fault-localization where the need is to provide attention to the complete code. Therefore, in the extra layers of the transformer trained, a full attention mechanism to attend to each line of the code equally is also applied to predict the faultiness of a line considering the whole code before and after it.

They compare their model with prior non-LLM deep learning-based techniques and the proposed model performs 14.4% better than the previous approaches. The study concludes that training and fine-tuning even a comparatively smaller transformer model on top of the pre-trained large language model can improve test-free line-level fault localization.

Wu et al., [71] test the fault localization capabilities of ChatGPT-3.5 and ChatGPT-4. The researchers perform the tests in two settings, one where they only provide the ChatGPT with the code along with the prompt to localize the fault, and in the other setting they also provide the code-related test case and execution logs. In the test-free approach, although ChatGPT performs in some cases better, its performance is comparable to the baseline state-of-the-art non-LLM-based technique SmartFL [76]. Also, the effect of the code context length is analyzed. According to the findings, ChatGPT performs well with small codes containing a single function, while its accuracy decreases to 49.9% lower than SmartFL when examining the code on the class level.

2.1.3 Test Execution and Results-Based Code Debugging and Analysis with LLMs

Kang et al., [50] explore automated debugging with LLMs by using test results and prompt engineering. The main motivation behind the study is to make the LLM-suggested code fixes more understandable to the developers. They claim that in current applications of AI and neural models for code debugging, the studies often lack the understanding of why the model suggested a specific fix for the code and what is the reasoning behind that specific suggestion. The researchers suggest that making the AI code suggestions more understandable will make the application of LLMs for code debugging more adaptable and trustworthy for the developers.

The purpose of such an approach is to make LLM-based automated debugging more aligned and closer to how human developers perform debugging. The aim was to provide an intelligible explanation for an LLM-generated patch so that the developers could take the right actions. It makes the incorporation of automated debugging more efficient and gives the developers the freedom to include the suggested patch or not.

To make the process more systematic, they adopt the Scientific Debugging process described in the book “Why Programs Fail” [77]. According to the book, a scientific code debugging process includes the following steps:

- Hypothesis - a possible explanation for the bug that aligns with the known observations.

- Prediction - the anticipated result if the hypothesis is correct.

- Experiment - a way to confirm the prediction.

- Observation - an experiment's outcome.

- Conclusion - an assessment of the hypothesis in light of the observation.

The researchers used OpenAI's ChatGPT (GPT 3.5 [78]) as their study was done before the release of the GPT 4 version model for the study. They firstly provide a detailed explanation of the above-mentioned scientific debugging steps with multiple examples for each step so that the LLM understands the task and can generate desired results.

The proposed technique is called Automated Scientific Debugging (AutoSD). In this technique when the LLM is provided with a buggy code along with the test results that revealed the bug, it generates a *hypothesis* for fixing the code in the form of a patch and *predicts* the possible and desired output. Along with providing a hypothesis for the code fix, the technique also suggests an *experiment* or a test case generated by the LLM itself that will reveal whether the suggested patch actually fixes the bug or not. After the execution of the suggested test, AutoSD *observes* if the results match the prediction of the originally desired outcome. Based on the observation, it makes a conclusion whether the hypothesis should be accepted or not. If the observation matches the predictions, then it finalizes the patch based on the original hypothesis, if not then the whole process is reiterated [50].

To evaluate the performance of the proposed technique, an empirical analysis with 20 human developers with six real-world bugs is performed. Although the study revealed

that the developers could judge the correctness of the suggested patches both with and without the explanation, their accuracy improved in 5 out of the 6 bugs. Further, 70% of the developers expressed that they would definitely prefer an explanation while using the automated repair tools [50].

Chen et al, [68] explored teaching a model to self-debug in order to generate better code. For complex programming tasks, it is challenging even for the human programmer to write a correct program in one go. Rather it is an iterative process with improvement in each cycle. Similarly, it is challenging to synthesize correct code from the LLMs in one go. Therefore, one can incorporate some techniques for the LLM to self-repair its generated program and improve it iteratively. The researchers did it without the need for any human feedback for code debugging.

The proposed approach does not involve any additional training of the model. It consists of firstly, tasking the model to generate code for a specific task. This is done by inputting a description of the problem along with unit tests. Then the model generates the code and if there is a unit test available for the task then it sends the code for execution to be tested. With the execution results and its self-generated code, the model performs rubber-duck debugging and checks whether the generated code passes the unit test or not. During this self-debugging, the model is able to identify any implementation errors and rewrite/regenerate the code.

They evaluate the technique on multiple models, namely StarCoder [79], code-DaVinci-002 [4], and GPT-3.5-turbo [78] and GPT-4 [49] in the GPT model family. They test with three benchmark datasets, Spider for text-to-SQL [80] and MBPP [81] for text-to-Python generation, and TransCoder [82] for C++-to-Python translation. For the Spider datasets, there are no unit tests available so the researchers only worked with code re-explanation as the feedback or self-debugging.

The researchers conclude that without unit tests available, the proposed method improves the baseline by 2-3% and the accuracy of prediction on the hardest level

problems improves by 9%. On the other hand, by utilizing unit tests, the technique improves the baseline accuracy by up to 12%.

Liventsev et al., [67] also explored a self-debugging approach. Named it the Synthesize, Execute, Instruct, Debug, and Rank, or SEIDR framework for generating and then debugging and improving/fixing code with large language models by using execution results utilizing unit tests. They emphasize that even though code-understanding large language models have the capability to produce code, that semantically resembles the correct answer, it often has some imperfections that result in no compilation or incorrect execution results [83]. Therefore, they study the prompts best suitable for such a task. Plus the researchers investigated which approach is better, whether to make the LLM to regenerate the whole code or to repair the already written code.

In the study, the OpenAI Codex model is used for program synthesis and debugging, and the Program Synthesis Benchmark 2 [84] is used as the dataset for this task.

The proposed technique incorporates three LLMs, two Codex (CLM) models, one for code synthesis and one for code debugging, plus a text LLM, GPT-3 for bug summarization. The researchers first, input the task description and a code template to one CLM. The generated draft programs are executed using the unit test, if they pass, they are accepted as the solution, if they do not pass, then taking into account the failing location, the text LLM instructs the second CLM to debug the originally proposed program candidate, and hence generate new program candidates.

They conclude that CLMs have better capability to regenerate a more accurate whole program, rather than fixing the originally written program. That indicates in the direction that LLMs lack the ability to effectively localize faults. Regarding prompt engineering, they report that the models perform comparatively similarly with different prompts and a fixed or GPT-assisted prompt does not necessarily make a big difference.

Kang et al., [85] explore method-level fault localization using ChatGPT. The researchers provided repository access to the LLM so that it has the full context of the project. The proposed method outputs an explanation and location of the fault. They utilize a failing test for the technique as the input to the LLM. The proposed approach is known as AutoFL. The researchers also compared their technique with non-DL and non-LLM-based techniques and obtained promising results.

2.2 Code Debugging and Analysis with Non-LLM-Based Deep Learning Techniques

Jiang et al. [5] compare their LLM-based APR results with four different DL-based (non-LLM-based) APR techniques and refer to them as the best open-source DL-based APR techniques, namely KNOD [86], Recorder [3], RewardRepair [87], and CURE [88]. By comparing the results, they also conclude that Recorder outperforms other studied models in terms of fixing the most bugs on their HumanEval-Java dataset.

KNOD is one of the recent deep learning-based automated program repair techniques. It uses a graph transformer [89] and a three-stage tree decoder they designed themselves to generate an abstract syntax tree (AST) [3]. Additionally, domain-knowledge distillation [90] is used to assist the model in picking up the syntax and semantics of code.

Recorder is also a DL-based APR technique. It modifies the AST to patched AST in order to generate edits on the input code. Working on the AST level makes this model perform well and generate more syntactically correct patches.

RewardRepair on the other hand, is based on the transformer architecture and is closely related to CLM-based APR techniques, architecture-wise. During its training, it also uses the patch execution information like its compilability and correctness for calculating loss. This training technique makes the model generate more compilable and correct patches.

CURE is an encoder-decoder model implemented with convolutional networks. It also applies a small version of code-understanding GPT [91], pre-trained on a small 4M Java function dataset. It enables the model to learn code syntax as well.

Yang et al., [53], also use recent machine (deep) learning-based fault localization (MLFL) techniques as their baseline. They study and discuss three MLFL techniques, namely DeepFL [55], DeepRL4FL [63], and TRANSFER-FL [64].

The DeepFL method employs Recurrent Neural Networks (RNN) and incorporates a bidirectional variant called BiRNN (Bidirectional RNN). It enables the models to take into account both the prefix and suffix of the code line/statement being debugged at any given time. The training of this model included datasets with different dimensionalities. Namely, spectrum-based (information related to program execution paths), mutation-based (derived from mutation testing), complexity-based (metrics related to code complexity), and textual-based (measures of similarity between code snippets). These features play an important role in precise fault localization.

DeepRL4FL which stands for Deep Representation Learning for Fault Localization, is another deep learning-based approach for fault localization. It is designed to identify buggy code at both the statement and method levels. The researchers employ Convolutional Neural Network architecture for classification. They convert the vector representation of the input code and feed it as the input to the CNN for detecting buggy statements/methods. They also utilize unit test cases and execution results for calculating data dependencies and code coverage matrix, which allows for better accuracy for fault localization.

TRANSFER-FL is another novel approach for fault localization automated program repair. It is based on bidirectional LSTM [92] architecture. In the study, the researchers apply a transfer knowledge learning approach and train their model on a large open-source Java projects-based dataset they created from bug-fixing commits on GitHub. They compared the fault localization capabilities of their model with various DL-based techniques, including DeepFL and DeepRL4FL, mentioned above, and reported it to

be outperforming all other methods. In addition, Yang et al., [53] also highlight TRANSFER-FL to be performing better than the above-mentioned methods.

2.3 Bug Fixing Benchmark Datasets

Defects4J v1.2 and v2.0 [93] are the most widely used [5], [50], [53], [65] real-world bug benchmarks containing data from 6 famous open-source Java projects like "Google Closure compiler" and "Apache Commons-math" [5], [53], [65]. Defects4J v1.2 contains 395 bugs while Defect4J v2.0 is the latest version of this dataset containing 438 more bugs from 9 new Java projects. The dataset also contains the developer test that revealed the bug for each bug in the dataset.

BugsInPy [94] is another real-world bug benchmark used by researchers for fault localization and APR tasks [50], [53]. It has 493 Python programming bugs from 17 different projects.

QuixBug-Python and Java [95] is another widely used benchmark for fault localization and APR [5], [65], [67]. It is based on 40 classic programming challenges, regarding famous algorithms like "merge sort" and "quick sort" [5]. It was created from a programming challenge in which programmers were challenged to fix a small bug. While originally being in Python it has been translated to Java as well, and both programming language versions contain the same 40 bugs. Every bug in this benchmark is also coupled with multiple test inputs and expected results [65].

HumanEval [4] is a manually created Python benchmark containing single Python functions. It was created for program synthesis specially to test LLMs in order to mitigate the issue of data contamination as LLMs might have seen the other benchmarks that were already available while training LLMs. It has also been used by multiple researchers for benchmarking debugging and APR techniques [4], [5], [50].

CHAPTER 3

METHODOLOGY & EXPERIMENTAL SETUP

In this section, the approach followed in this study is presented, including the architecture of the developed FLICoder models, details about the LLM used, and the fine-tuning method adopted. Also, the experimental setup is described to perform debugging and analysis of code samples, specially focusing on fault localization.

Figure 3.1 shows the FLICoder architecture investigated in this study. The input to the FLICoder model is a buggy code containing multiple lines and the output of the model is the suspiciousness score of each line for being buggy/faulty. As can be seen in the figure, the first step is to tokenize the input code. Then feed the tokens to the pre-trained large language model. Thereafter, the high-dimensional hidden representations of each line are obtained from the pre-trained LLM and then fed to the small adapter model which is trained to predict the faultiness of each line. In this approach, the pre-trained large language model stayed frozen, and only the adapter model was trained. This is a form of parameter-efficient fine-tuning of LLM, called adapter tuning, as described in section 1.1.4.3. It also makes FLICoder customizable, where the pre-trained large language model can be easily replaced with any other LLM without any major modifications required.

In the following sections, all components of the FLICoder models are described. First, the pre-trained large language models used are explained, then the adapter model trained for this study is discussed, and finally, each step the input goes through from the beginning till obtaining the final output of faultiness predictions is given.

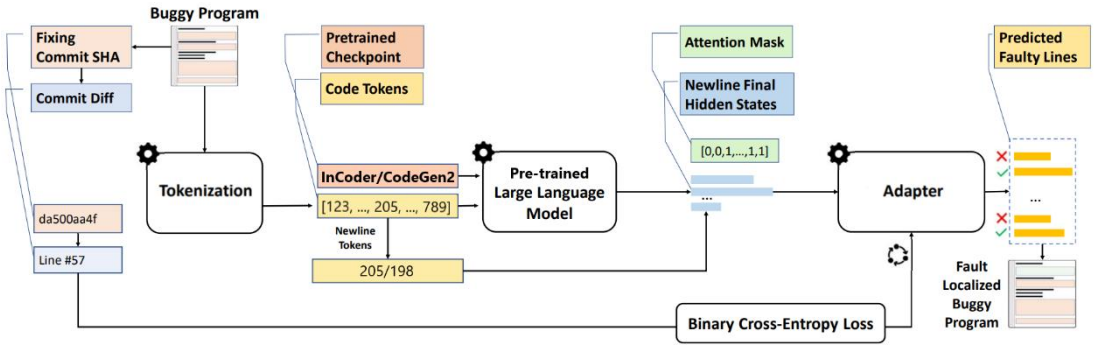

Figure 3.1 FLICoder Architecture [53].

3.1 Used Pre-trained Large Language Models and Their Sizes

In the work of Yang et al. [53], the researchers used CodeGen, which is a left-to-right large language model. Left-to-right LLMs attend to only the prefix of input at any given point and mask out the suffix. It is thought that this makes left-to-right LLMs not the best choice for code debugging tasks since incorporation of the whole code, including the prefix and suffix is essential for an effective modeling of the code context.

Therefore, in this thesis, the use of the LLMs capable of masked language modeling, hence attending to both the prefix and the suffix of the code is investigated. For that purpose, we test with two such CLMs. Firstly, the InCoder model is used, which is pre-trained for both left-to-right generation, as well as masked span prediction. The other one is the new version of CodeGen, called CodeGen2 [96], released in 2023, which has a bidirectional attention mechanism and code-infilling capabilities, as opposed to the older version.

The InCoder model was developed by Meta. Two size variants of the model were released, InCoder-1B and InCoder-6B. They also incorporated their own customized tokenization scheme. The tokenization is the same for both size variants [32].

The InCoder developers also show in their work that for code generation tasks, it gives comparable results to the 6B size variant of CodeGen on the HumanEval [4] metrics. Furthermore, the recent study done by Jiang et al. [5], reports that the InCoder model outperforms other state-of-the-art code language models for the task of automated program repair.

The InCoder model was pre-trained on code data from GitHub, GitLab, and StackOverflow posts. Along with code data, it was also pre-trained on a large dataset of English text. The code data was composed of 33% Python, 26% JavaScript, 10% C/C++, and smaller percentages of other programming languages.

On the other hand, CodeGen2 is a new version of CodeGen that was released in 2023 by Salesforce. In CodeGen2 they improved the model by enabling bidirectional attention and code infilling capabilities. As Yang et al., [53] used CodeGen in their original studies, we also tested using the same model but with bidirectional attention to observe its impact. CodeGen2 was trained on Java, Python, C, C++, C#, Javascript, and other programming languages.

Now, language models are typically designed to predict the next token in a sequence. However, they can also provide the "hidden" states from their final Transformer layer. These states are transformed into a prediction for the next token during text generation. Therefore, these states represent the model's understanding of the context at each point in the sequence, which is inherently valuable.

As illustrated in Figure 3.1, the final hidden states for each newline token in each training sample are extracted from the CLM. This results in a condensed sequence representation where each token represents one line. We then pair these tokens with their corresponding location in the file (e.g., line #5 of a 50-line file) and store them on the disk.

During the FLICoder's training, these encoded lines are loaded in batches. Samples of up to 128 contiguous newline states are retrieved at a time. This number was chosen

because the used CLMs can process a maximum of 2,048 tokens; inputs with 128 lines almost always fit within this limit. Samples with fewer lines are padded, along with the label vector, to achieve a uniform length. Padding entries are not considered in the loss computation.

For files with more than 128 lines, a series of 128-line windows that cover each faulty line in the file are sampled. A sample with up to 128 lines is repeatedly created, starting from a random offset before the next faulty line that has not been covered by a previous segment. All faulty lines in this segment are marked as covered and this process is repeated until all lines are covered by at least one segment. Random starting offsets are chosen to prevent the model from memorizing certain index locations as faulty lines. This approach allows the technique to handle inputs longer than 2048 tokens.

3.2 Adapter Model

Since the pre-trained LLMs (PLMs) are not suitable directly for outputting suspiciousness scores, as discussed in section 1.1.4, an adapter model in series with the PLM to convert the hidden states from the PLM to line-level suspiciousness score is trained in this study.

A bidirectional attention Transformer [33] architecture is chosen to make sure the complete context of the code is taken into account. This allows the model to exchange the information between both the earlier and the later lines of the code.

3.3 Model Pipeline and Adapter Training

The precise step-by-step details of the proposed approach can be split into five steps, given below.

3.3.1 Tokenization

Beginning with a sequence of code tokens $C = [c_0, c_1, \ldots, c_N]$, a pre-trained Transformer is used, denoted as T_{PT}, to convert these tokens into representational "states" $S \in \mathbb{R}^{N \times D}$. Here, D represents the dimensionality of the pre-trained model. This process is performed "offline" since we do not train/fine-tune the PLM itself.

3.3.2 Hidden States Extraction from the PLM

In this step, extraction occurs. Each new line token's representations are extracted, which, in essence, contain the state of each line in the input program: $S_{NL} \in \mathbb{R}^{M \times D} = S[c_i = \backslash n]$, where M is the number of lines in the input code. These representations will contain the information of the complete line [53].

3.3.3 Dimensionality Reduction

The dimensionality, D, of the hidden vector representation obtained from the PLM, is very high. Since the adapter layers in this study are trained on a much smaller dataset, a smaller dimension $d \ll D$ is used for the adapter model. We reduce the S_{NL} dimension to $R_{NL} \in \mathbb{R}^{M \times d} = S_{NL} W_d$ where $W_d \in \mathbb{R}^{D \times d}$ is a weight that can be learned and is equivalent to a fully connected layer. The experimented dimensions in this research are $d \in \{256, 512, 1024\}$.

3.3.4 Processing Through the Adapter Model

A bidirectional Transformer adapter of n layers with an internal dimension of d is trained in this stage. The dimension-reduced hidden representations from the above step are passed through this adapter to output the final states based on each line of the input code. This final representation can be expressed as $A_{NL} \in \mathbb{R}^{M \times d}$, which captures the representation of each new line token and its effect in a bidirectional context.

3.3.5 Line-Level Faultiness Score Predictions

This is the final step of the architecture. Here we transform the representation of each new line token to a single value between 0 and 1. The sigmoid function is used for this purpose to create a dense projection $B = \sigma(R_{NL}W_b)$ where $W_b \in \mathbb{R}^{d\times 1}$. The resulting predictions are the probabilities of each line being buggy/faulty according to the model. A binary cross-entropy loss, $\mathcal{L}_{CE} = T\ln(B) + (1-T)\ln(1-B)$ is calculated and used to compare the obtained predictions against the ground-truth labels $T \in \{0,1\}^M$. This loss is then backpropagated through all the layers of the model up to but excluding the last layer of the PLM since we do not modify the parameters of the PLM. The gradients are averaged across the minibatch of the samples, and the model states are updated to minimize the loss and hence make the output predictions closer to the ground truth. The researched hyperparameter settings to optimize and improve the training are discussed in the next section.

3.4 Datasets

In this study, we used two bug-fixing datasets, namely, Defects4J and BugsInPy. Defects4J and BugsInPy benchmarking datasets are the most widely used in the domain of code debugging and fault-localizing models. Table 3.1 presents the details of these datasets.

Defect4J [93], contains 395 Java code samples with bugs and relevant bug-fixing commits. Similarly, BugsInPy [94] is a Python dataset with 493 bugs and their corresponding bug-fixing commits. These and some other prevalent bug-fixing datasets are also discussed in detail in the Literature Review section.

3.4.1 Preprocessing of the Datasets

We take the preprocessed versions of these datasets from the previous open-source study done by Yang et al., [53]. The datasets were preprocessed to get the line numbers of the buggy statements/lines in each code sample. The statements that were changed

in the bug-fixing commit are considered buggy statements. The line numbers of these buggy statements were recorded and used as the ground-truth labels during the training and validation of the models in this study.

As seen in Table 3.1, it is to be noted that although BugsInPy has more buggy code samples than Defects4J, it has on average, less number of lines in the code samples, which essentially makes it a smaller dataset for such an application, since individual lines are the actual samples for line-level fault localization.

Table 3.1 Datasets: Defect4J vs BugsInPy

	Defects4J	**BugsInPy**
Total Buggy Code Samples	395	493
Total Number of Code Lines	168,960	76,672
Programming Language	Java	Python

3.5 Adapter Dimensions

Due to limited data, we configure smaller dimensions for the adapter model following prior techniques [53]. The FLICoder models are trained as default on an adapter input dimension of 512 which is projected down from the CLM's hidden state dimensions to perform dimensionality reduction. Although 512 is used as the default size for the adapter model, dimension sizes 256 and 1024 are also tested to explore their effect on the model's performance.

3.6 Training Hyperparameters

In Table 3.2, we present the hyperparameters used to train the models.

3.6.1 Learning Rate

A maximum and a minimum learning rate for the models' training are defined. The maximum learning rate was decided with trial and error with a few training cycles in the beginning. While the minimum learning rate was decided to be 1e-6, following prior techniques [53].

For maximum learning, we started with 1e-3 and performed a few training cycles noticing the validation loss converging patterns. If the improvement in the model's training stalled after only the first 50 epochs, depicting an overshoot in gradient descent, we reduced the maximum learning rate by 5% and performed another training cycle. The process was repeated until we saw improving patterns in the model training at least until and after the first 100 epochs. This process was performed once for the small-sized CLMs (CodeGen2-1B and InCoder-1B) and once for the larger-sized CLMs (CodeGen2-3.7B and InCoder-6B) considering a difference in the model's training behavior due to the change in dimensions.

During the training, learning rate scheduling was performed between the minimum and maximum learning rates. The learning rate was decreased until a stable convergence was observed in both the training and validation loss. Consistent with standard procedures in language model training [97], a learning rate warm-up of 1000 steps was used, followed by a cosine learning rate decay to the minimum learning rate of 1e-7 over 20,000 steps.

3.6.2 Model Dimesnion

Model dimensions of 2048 and 4096 were configured according to the CLM's hidden states' dimension, which is then projected to the adapter's input dimension of 256, 512, or 1024.

Table 3.2 Hyperparameters for Adapter Training

HYPERPARAMETERS	CodeGen2-1B/ InCoder-1B	CodeGen2-3.7B/ InCoder-6B
Max Learning Rate	8e-5	6e-5
Min Learning Rate	1e-6	1e-6
Model Dimension	2048	4096
Layers	2	2
Batch Size	8 & 32	8 & 32
Epoch	300	300

3.6.3 Layers

The number of layers was chosen following the prior approach [53]. Only two layers are used to keep the adapter model small due to the limited amount of training data available.

3.6.4 Batch Size

A batch size of 32 is tested at the beginning for both Defects4J and BugInPy following prior approaches [53]. However, different batch sizes were also tried and a few training cycles were performed for each dataset with batch sizes of 8, 16, 32, and 64. We found out that although Defects4J gives the best results with a batch size of 32, BugsInPy gives the best results with a batch size of 8. Therefore, we used a batch size of 32 for Defects4J and a batch size of 8 for BugsInPy. A comparison of FLICoder performance with both datasets and batch sizes of 8 and 32 is presented in Appendix B.

3.6.5 Number of Epochs

For a fair comparison, a fixed number of epochs (300) was used for training all FLICoder models consisting of different CLM sizes and adapter dimensions.

3.7 Evaluation Metrics

We evaluate FLICoder on multiple metrics to understand different aspects of the model's performance. The used metrics are explained below.

3.7.1 Precision, Recall, F1 Score, and Accuracy

The precision, recall, F1 score, and accuracy metrics are used. Precision is useful to understand how many times the model is correct whenever it classifies a sample as the positive class. On the other hand, recall tells us how many times the model can correctly classify a sample as the positive class out of the total positive class samples. While the F1 score provides an overall measure of the model's performance considering both the precision and recall. F1 score is the harmonic mean of precision and recall, computed as follows:

$$F1\ Score = 2\ x \frac{Precision\ \times Recall}{Precision + Recall}$$

Lastly, the accuracy tells how many times the model correctly classified both the positive and negative class samples out of the total positive and negative class samples.

3.7.2 Receiver Operating Characteristic (ROC) Curves

We also draw the Receiver Operating Characteristic (ROC) curves for our models. The ROC curve shows a classification model's performance for different thresholds. It is useful to understand the model's overall ability to perform accurate and precise predictions.

3.7.3 TOP-N

To compare FLICoder models with prior models and techniques, for Research Question (RQ) 5, we also evaluate them on the Top-N metric. The Top-N metric quantifies the number of faults located by the model, that contain at least one faulty element within the first N positions (N=1, 3, 5). Developers typically scrutinize a small subset of elements, deemed most likely to be faulty, within a ranked list [98], with a particular focus on the Top-5 elements [99]. To benchmark against prior techniques, the Top-N metric is utilized, as done in prior studies [53], [55].

3.8 Validation

A ten-fold validation is performed by executing 10 training and validation iterations. The first iteration starts with the validation partition starting from the beginning of the dataset, index 0, to an index that covers 10% percent of the dataset. Then in the next training cycle, the validation indices are moved ahead by 10% of the dataset, starting at the index that marks 10% of the dataset and ending at the index that marks 20% of the dataset. This way, 10 training iterations are completed where each time a different 10 percentile of the dataset is held for validation and the rest of the 90% dataset serves for training. By doing this, it is ensured that each sample in the dataset is held out for validation exactly once.

The model's performance is kept track of in each validation cycle whenever the validation precision and recall improve. In the end, after the ten-fold validation process, the model is evaluated on the Top-N metrics. This Top-N evaluation is performed following prior techniques [53] for evaluating FL models with small datasets.

Furthermore, an early stopping mechanism is incorporated by keeping track of the model's performance in terms of precision and recall, and saving the model checkpoint only whenever the model's performance improves. However, the model is kept training for a sufficient number of epochs (300) in case the performance improves

further. This training procedure is kept for the entire ten-fold iterations. In the end, the best-performing model is used to evaluate the precision, recall, F1 score, and accuracy metrics. The ROC curve of the best-performing model from the ten-fold training-validation cycle is also drawn.

3.9 Environment

The study is conducted on Google Colab which is a hosted Jupyter Notebook service allowing run sessions fully remotely and program in Python3. It allows Google Drive to be mounted to your session in order to access your files. Further, it gives access to various GPUs of different sizes [100].

In this study, different GPUs at different stages of the model pipeline are preferred according to the requirement. The GPUs, their sizes, and the stage of the model pipeline where they were used in the study are presented in Table 3.3.

As discussed above in section 3.3.2, the data is passed through the CLM, and the hidden states are loaded and saved offline as a preprocessing step, before training the adapter model. For this step, larger GPUs were required according to the CLM's size. After this stage, we use the smallest available GPU for adapter training as it is a small model and does not require too many resources.

Table 3.3 GPUs Used in the Study

GPU	Memory Size	The Task in our Study
NVIDIA Tesla T4	15 GB	For training the adapter models.
NVIDIA Tesla V100	16 GB	For loading InCoder-1B hidden states.
NVIDIA L4	24 GB	For loading CodeGen2-1B and CodeGen2-3.7B hidden states.
NVIDIA A100	40 GB	For loading InCoder-6B hidden states.

Table 3.4 shows the libraries and modules that we used and their corresponding tasks.

Table 3.4 Python Libraries and Modules Used in this Study

LIBRARIES	DETAILS
os	Operating System (OS) module, is a part of the Python Standard Library. It was used in this research to perform directory-related tasks, like changing the working directory and creating new folders.
argparse	Argument Parser, or argparse, is also a module from the Python Standard Library. It was used to parse arguments when a program was run through a command-line interface.
numpy	NumPy or Numerical Python is an open-source Python library. It was used to work with multidimensional arrays.
torch	Torch or PyTorch is a machine learning library in Python. It was mainly used for working with Tensors and training the models.
torchdata	TorchData is also a part of the PyTorch project, used for handling data loading.
json	JSON is also a built-in Python module used to work with JSON files.
csv	Similar to JSON, used for working with CSV files.
random	Random is also a built-in Python module. It was used in this study for tasks requiring randomness, like providing a random seed to the model while training and randomly shuffling the data.
glob	The glob module is used to easily find pathnames and files in a directory. Used in this study to efficiently work with files.
sklearn	Scikit-learn is a Python module that contains useful methods for machine learning-related computations. We used it in our study for creating model performance graphs like ROC curves.
matplotplib	Matplotlib is a Python library for working with visualizations in Python. We used it to optimally form our ROC graphs.

CHAPTER 4

RESULTS AND DISCUSSION

In this chapter, we present and discuss the results of our experiments performed in order to find out the answers to the research questions established in section 1.4. For each research question, multiple experiments were performed and the outcomes were evaluated on the metrics defined in section 3.7.

The results are presented in the form of tables and graphs. As we compare different aspects of the FLICoder models in each table, the main differentiating aspects of the model are formatted in *italics* for easy understanding. For example, in Table 4.1, we compare FLICoder models using different CLMs. So we italicised the name of the CLMs as *InCoder-6B* and *CodeGen2-3.7B*. While in Table 4.3, we compare the FLICoder models using the same CLM but only different sizes. So we italicise only the number of parameters like CodeGen2-*1B* and CodeGen2-*3.7B*.

The code lines where we perform the calculation of the evaluation metrics are presented in Appendix A, at the end of this document.

4.1 RQ1: How does using different CLMs affect the fault localization performance of FLICoder models?

To explore the impact of using different CLMs on the performance of FLICoder, we train and test FLICoder models with two different CLMs, CodeGen2 and InCoder. To facilitate a fair comparison we compare the similar-sized CLM versions. Moreover, in these experiments, we fix the dimensionality-reduced input size of our adapter model to 512 and train all FLICoder models for 300 epochs on Defects4J.

4.1.1 FLICoder with InCoder-6B vs CodeGen2-3.7B

First, we test FLICoder with InCoder-6B versus FLICoder with CodeGen2-3.7B. Although a more similar-sized version of the CodeGen2 model was available with 7B parameters, it could not fit on the largest GPU (NVIDIA A100, see section 3.9) of 40GB we had access to. Therefore, we used CodeGen2-3.7B.

Table 4.1 shows the results of this experiment in terms of the model's precision, recall, F1 score, and accuracy. We can see that FLICoder with CodeGen2-3.7B has considerably higher fault localization precision (77%) as compared to FLICoder with InCoder-6B (37.4%). On the other hand, InCoder-6B results in a higher recall (about 6% higher) than CodeGen2-3.7B, and hence it also has a higher F1 score. At the same time, both models show similar performance in terms of accuracy.

Table 4.1 FLICoder Performance with InCoder-6B vs CodeGen2-3.7B on Defects4J

	FLICoder with *InCoder-6B*	**FLICoder with *CodeGen2-3.7B***
Precision	37.4%	77.0%
Recall	20.2%	14.1%
F1 Score	26.2%	23.8%
Accuracy	96.9%	96.8%

We also present a Receiver Operating Characteristic (ROC) curve for both FLICoder models with InCoder-6B and CodeGen2-3.7B in Figure 4.1. It is visually quite prominent that CodeGen2-3.7B has a higher area under the ROC curve (AUC), and hence can be said to be better at distinguishing faulty lines from non-faulty lines at the majority of classification thresholds.

4.1.2 FLICoder with InCoder-1B vs CodeGen2-1B

Secondly, we tested FLICoder with InCoder-1B and CodeGen2-1B. Table 4.2 shows a comparison of the two models. We can see a very similar trend here as in Table 4.1.

Table 4.2 FLICoder Performance with InCoder-1B vs CodeGen2-1B on Defects4J

	FLICoder with *InCoder-1B*	**FLICoder with *CodeGen2-1B***
Precision	30.9%	77.3%
Recall	20.5%	11.3%
F1 Score	24.7%	19.7%
Accuracy	96.4%	97.0%

Although with CodeGen2-1B, FLICoder is able to perform way more (about 47% more) precise fault localization, it has a lower recall and F1 score than FLICoder with InCoder-1B.

Furthermore, in Figure 4.2, it can be seen that here as well, the models have similar a trend in the ROC curve as Figure 4.1. FLICoder with CodeGen2-1B has a visibly higher AUC compared to FLICoder with InCoder-1B.

4.1.3 RQ1 Summary

Using different CLMs leads to different results with FLICoder although trained on the same data in the same environment. Although FLICoder with both InCoder and CodeGen2 can correctly localize faults with high accuracy, CodeGen2 brings in a higher precision but a lower recall. On the other hand, InCoder shows a more balanced understanding of both the faulty and non-faulty lines and delivers a higher F1 score.

The decision of preferring one CLM over the other is based on the application-specific needs of what characteristic is desired more. For example, with high precision, FLICoder with CodeGen2 avoids false alarms and has a high rate of correct classification whenever it classifies a line as being faulty. This way it can minimize wasted resources in investigating non-existent faults. On the other hand, it has lower recall, which might lead to not recognizing some of the faults.

In contrast, InCoder-based FLICoder provides a comparatively higher recall, so it can recognize more bugs than CodeGen2-based FLICoder, and hence contribute more in minimizing software failure. However, it has a considerably lower precision, so it can sometimes result in classifying a non-faulty line as a bug.

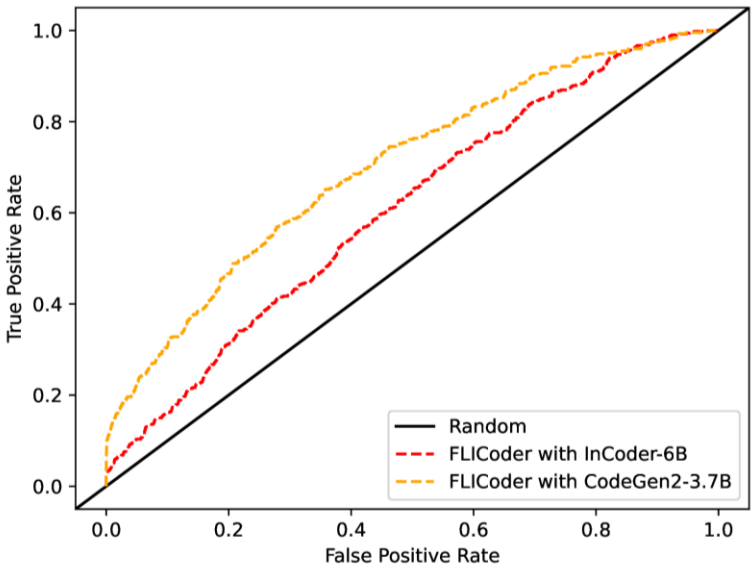

Figure 4.1 ROC Curve – InCoder-6B vs CodeGen2-3.7B on Defects4J

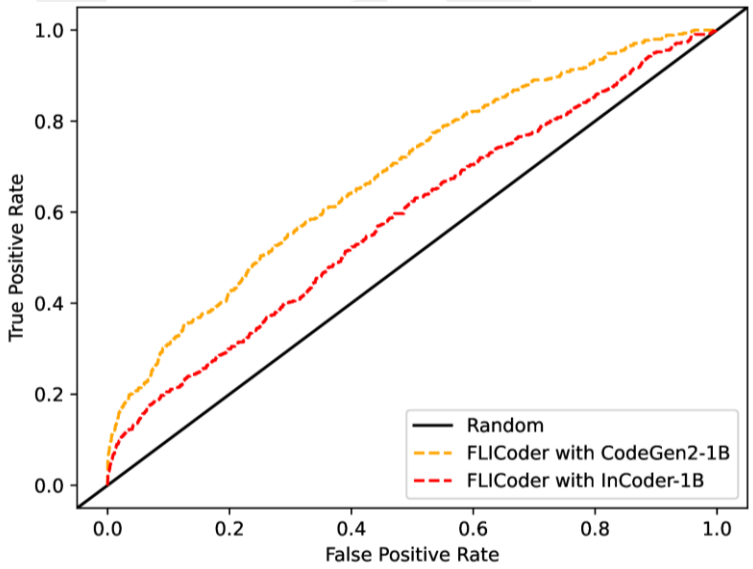

Figure 4.2 ROC Curve – InCoder-1B vs CodeGen2-1B on Defects4J

4.2 RQ2: How does the size of the used CLM affect the performance of the FLICoder models?

For this RQ, we compare FLICoder with CodeGen2-1B versus CodeGen2-3.7B and FLICoder with InCoder-1B versus InCoder-6B on Defects4J and adapter dimensions of 512. Table 4.3 and Table 4.4 show these comparisons respectively. Similarly, FLICoder with InCoder also shows very similar performance with both sizes. However, FLICoder with InCoder-6B exhibits a slightly higher precision. Therefore, it has a higher F1 score.

Figure 4.3 and Figure 4.4 show the ROC curve for FLICoder with different sizes of CodeGen2 and InCoder trained on Defects4J, respectively. Although the curves are quite identical, we can see a slightly higher AUC with larger CLMs in both cases.

Table 4.4In the case of CodeGen2, FLICoder performs very similarly in terms of precision and accuracy with both sizes. However, with CodeGen2-3.7B, it is able to recognize more bugs and shows a slightly higher recall (~3%) and F1 score (~4%).

Table 4.3 FLICoder Performance with Different Sizes of CodeGen2 on Defects4J

	FLICoder with CodeGen2-*1B*	**FLICoder with CodeGen2-*3.7B***
Precision	77.3%	77.0%
Recall	11.3%	14.1%
F1 Score	19.7%	23.8%
Accuracy	97.0%	96.8%

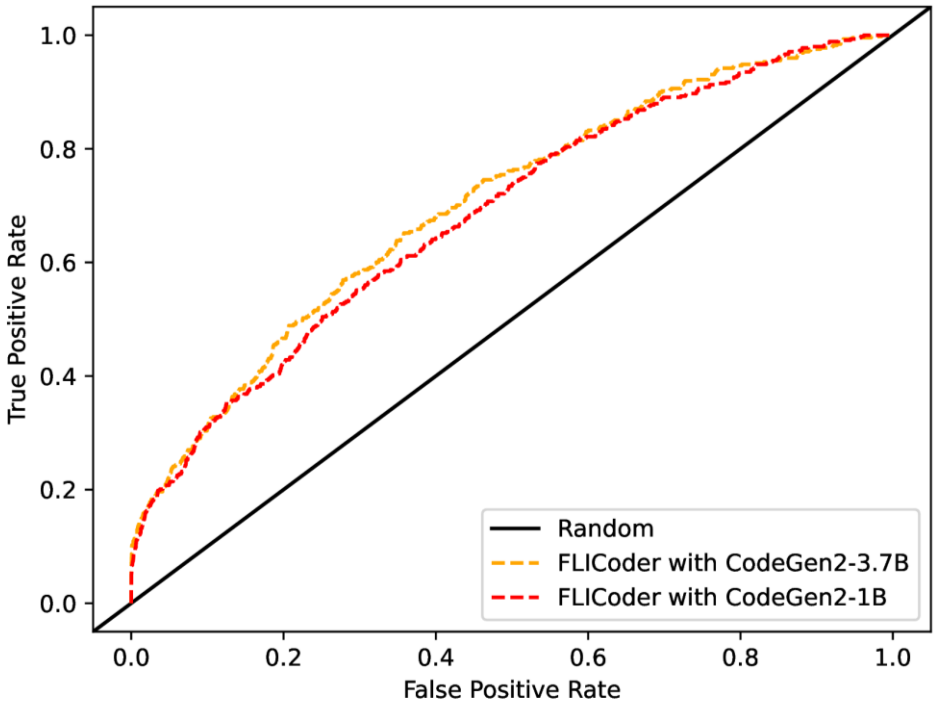

Figure 4.3 ROC Curve – CodeGen2-1B vs CodeGen2-3.7B

Similarly, FLICoder with InCoder also shows very similar performance with both sizes. However, FLICoder with InCoder-6B exhibits a slightly higher precision. Therefore, it has a higher F1 score.

Figure 4.3 and Figure 4.4 show the ROC curve for FLICoder with different sizes of CodeGen2 and InCoder trained on Defects4J, respectively. Although the curves are quite identical, we can see a slightly higher AUC with larger CLMs in both cases.

Table 4.4 FLICoder Performance with Different Sizes of InCoder on Defects4J

	FLICoder with InCoder-*1B*	**FLICoder with InCoder-*6B***
Precision	30.9%	37.4%
Recall	20.5%	20.2%
F1 Score	24.7%	26.2%
Accuracy	96.4%	96.9%

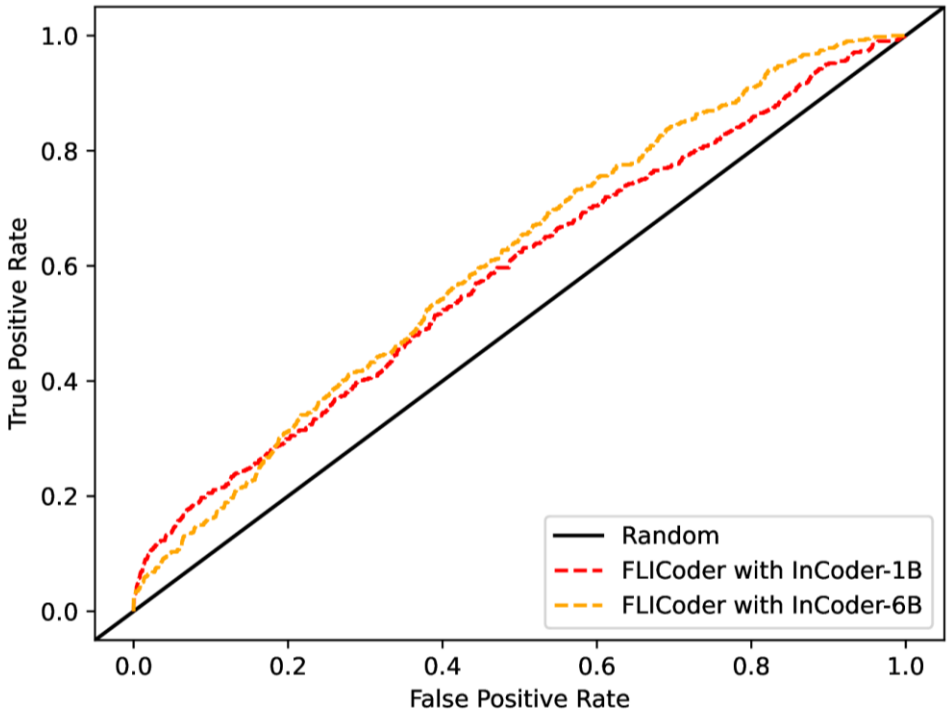

Figure 4.4 ROC Curve – InCoder-1B vs InCoder-6B with Defects4J

4.2.1 RQ2 Summary

FLICoder performs overall quite similarly with the tested different sizes of CodeGen2 (1B and 3.7B) and InCoder (1B and 6B). However, the F1 scores and ROC curves show a slightly better performance with larger versions of the CLMs.

One of the reasons for very similar performance with both small and large-size models can be due to the dimensionality reduction step in our architecture. Although we are getting a denser hidden representation with larger models (4096 as compared to 2048 with small models), we are still suppressing it to a dimension of 512. Now, as we had limited data we had to keep the model small in order for it to be tunable with the available amount of data. The effect could be explored in the future with larger datasets and dimensions.

4.3 RQ3: How does using different sizes and dimensions of the adapter model affect the FLICoder's performance?

As discussed in section 3.3.3, we reduce the dimensions of the hidden states we receive from the CLM as the first layer in the FLICoder's adapter model. As default, we kept these dimensions to 512 for all other experiments. However, we also explore the effect of this dimension size on the FLICoder's performance. For this, we test the dimensions of 256, 512, and 1024.

Table 4.5 FLICoder Performance with Different Target Dimensions of the Adapter Model with CodeGen2-3.7B on Defects4J

	FLICoder with CodeGen2-3.7B-*256*	**FLICoder with CodeGen2-3.7B-*512***	**FLICoder with CodeGen2-3.7B-*1024***
Precision	55.4%	77.0%	76.6%
Recall	13.6%	14.1%	5.4%
F1 Score	21.8%	23.8%	10.1%
Accuracy	97.0%	96.8%	97.1%

We train different FLICoder models with CodeGen2-3.7B and InCoder-6B while varying the above-mentioned input dimensions of the adapter model. We use the Defects4J dataset for these experiments.

Table 4.5 and Table 4.6 show the comparison of these models with CodeGen2-3.7B and InCoder-6B, respectively. In addition, Figure 4.5 and Figure 4.6 show a comparison of the models' ROC curves.

In these experiments, we see a bit of varied behavior depending on the CLM used. With CodeGen2, we see the optimal performance with a dimension of 512. It delivers the highest F1 score compared to the others. In terms of precision, both 512 and 1024-dimension-based FLICoder models perform similarly with a precision of about 77%. While a dimension of 256 delivers considerably less precision of about 55.4%.

On the other hand, with the InCoder-based FLICoder models, we see quite a comparable performance with the dimensions of 512 and 1024. In contrast, with a dimension of 256, we have a similar trend as with the CodeGen2-based FLICoder models; It delivers a comparatively lower overall performance compared to the others.

Table 4.6 FLICoder Performance with Different Target Dimensions of the Adapter Model with InCoder-6B on Defects4J

	FLICoder with InCoder-6B-*256*	**FLICoder with InCoder-6B-*512***	**FLICoder with InCoder-6B-*1024***
Precision	30.0%	37.4%	38.4%
Recall	18.4%	20.2%	21.9%
F1 Score	22.8%	26.2%	27.9%
Accuracy	96.3%	96.9%	96.5%

Now, if we look at the ROC plots, we can see overall a very similar performance here as well, depicting quite a matching strength of the models despite the change in the dimensions.

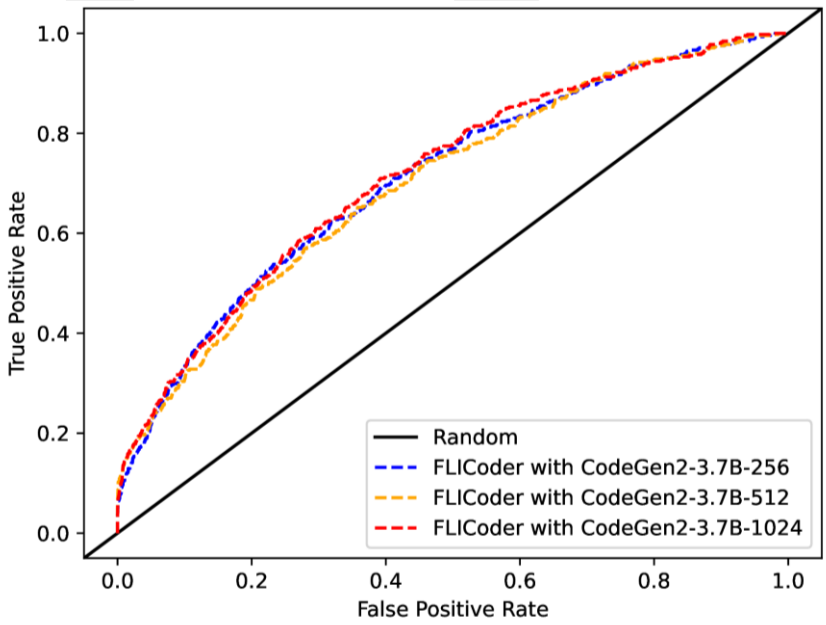

Figure 4.5 ROC Curve – CodeGen2-3.7B with Different Adapter Dimensions

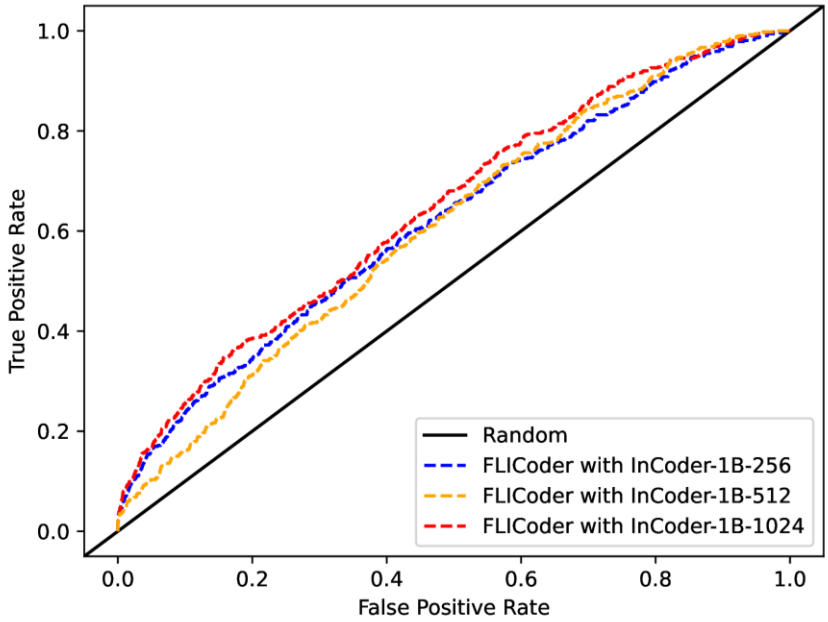

Figure 4.6 ROC Curve – InCoder6B with Different Adapter Dimensions

4.3.1 RQ3 Summary

With an increment in adapter input dimensions, we generally see an improvement in the FLICoder's performance, specially going from 256 to 512. However, in the case of 512 and 1024, we see a changed response with the different CLMs. With CodeGen2, the performance decreases with 1024, and the model's recall falls to 10.1% from 23.8% compared to 512. On the other hand, with InCoder, the FLICoder delivers a very similar performance with both 512 and 1024.

In addition, the ROC curves show a very similar strength of all the models with the same CLM. This shows us that, there is not a very strong relation between the model's input dimension and the performance. We get comparable results as long as the input dimension is appropriate enough to capture the input information and still fits well with the model's overall intended size. The intended model size depends on different factors, like the suitability of the available amount of training data in our case.

4.4 RQ4: How well do the FLICoder models perform on different datasets of different programming languages, namely Java and Python?

In this experiment, we trained and tested FLICoder on two different datasets in the same settings. We use FLICoder with CodeGen2-3.7B and use an adapter input dimension of 512. We use Defects4J, a Java bug-fixing dataset, and BugsInPy, a Python bug-fixing dataset.

It is to be noted that as mentioned in section 3.6.4, a batch size of 8 was found suitable for training with BugsInPy. Therefore, in this experiment, we use the same batch size for Defects4J as well to support the fairness of comparison. Although it decreased the overall performance of FLICoder with Defects4J compared to the results discussed above, it still works to facilitate the comparison with BugsInPy.

Table 4.7 shows the results of this test. With BugsInPy, we do not get a much higher performance. FLICoder on BugsInPy could only achieve a precision of 15.8%, a recall of 3.3%, and an F1 score of 7.9%. However, we see a high accuracy of about 97.6%, which indicates a possible imbalance in the dataset. It should also be noted that BugsInPy is a smaller dataset with less than half the total samples (lines) than Defects4J. It can also be a reason for poor performance with this dataset.

Table 4.7 FLICoder with CodeGen2-3.7B Performance on Defects4J vs BugsInPy

	Defects4J	**BugsInPy**
Number of Lines	168,960	76,672
Precision	21.3%	15.8%
Recall	17.2%	3.3%
F1 Score	18.9%	7.9%
Accuracy	95.8%	97.6%

Furthermore, we draw ROC curves for these tests as well, as shown in Figure 4.7. It can clearly be seen that Defects4J results in a much smoother ROC with a higher AUC.

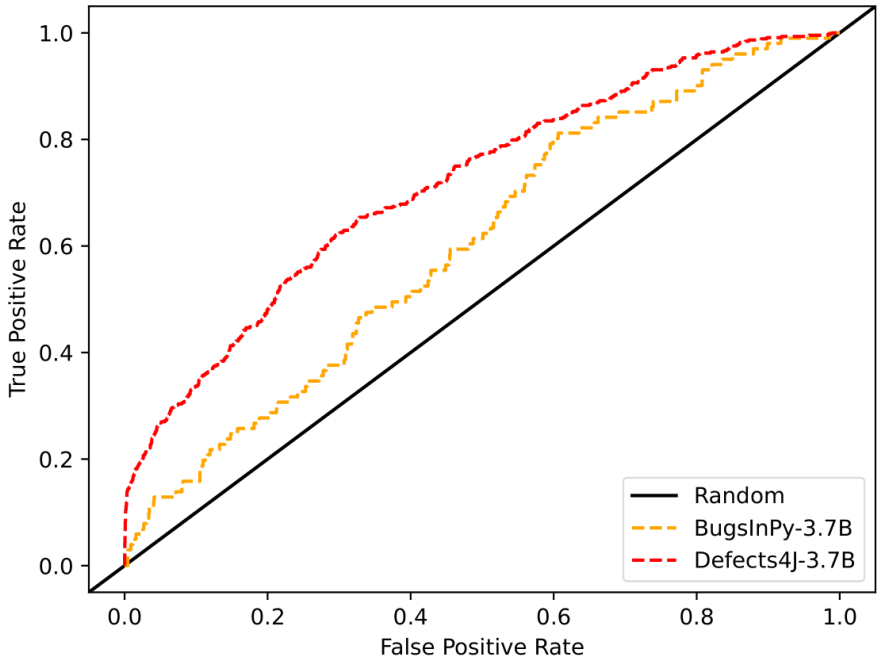

Figure 4.7 ROC Curve – FLICoder with CodeGen2-3.7B on Defects4J vs BugsInBy

4.4.1 RQ4 Summary

The dataset used to train FLICoder has a very strong impact on its performance. A balanced dataset with a higher number of samples leads to better results. FLICoder achieves better results with Defects4J as compared to BugsInPy.

4.5 RQ5: How does using a bidirectional attention-based CLM affect the FLICoder's performance compared to the prior approaches that used a prefix-only CLM?

In this experiment, we compare the results delivered with our model's architecture compared to LLMAO, the prior technique studied by Yang et al., [53]. LLMAO serves

as the baseline for our study. In LLMAO, Yang et al. worked with the older version of CodeGen that has a prefix-only attention mechanism.

Table 4.8 shows a comparison of FLICoder with CodeGen2 versus LLMAO with CodeGen evaluated using Top-N on Defects4J. One can see that the proposed approach in this study significantly outperforms LLMAO in all categories. Our largest model, FLICoder with CodeGen2-3.7B, even though smaller than LLMAO with CodeGen-6B, outperforms it by 30.7%, 45.1%, and 52.4%, in Top-1, Top-3, and Top-5, respectively.

Table 4.8 FLICoder Performance with CodeGen2 (Bidirectional Attention) vs LLMAO with CodeGen (Prefix-only Attention); Evaluated on Top-N on 395 Bugs from Defects4J

Model	Top-1	Top-3	Top-5
LLMAO* with *CodeGen-350M	82 (20.8%)	106 (26.8%)	126 (31.9%)
LLMAO* with *CodeGen-6B	85 (21.5%)	115 (29.1%)	160 (40.5%)
FLICoder* with *CodeGen2-1B	183 (46.3%)	218 (55.2%)	262 (66.3%)
FLICoder* with *CodeGen2-3.7B	206(52.2%)	293(74.2%)	367(92.9%)

4.5.1 RQ5 Summary

FLICoder models, using CodeGen2 (with a bidirectional attention mechanism), outperform the baseline LLMAO models that use CodeGen (with a prefix-only attention mechanism). The bidirectional attention in CodeGen2 improves FLICoder's comprehension of the entire code context, leading to more effective bug identification.

4.6 Results Conclusion

By looking at the results and discussion presented above, we can see that FLICoder is able to perform fault localization by using CLMs and adapter tuning. However, its performance is influenced by various factors.

While FLICoder surpasses the baseline LLMAO models in Top-N metrics, it's important to note that its overall performance, as measured by the F1 score, is not exceptionally high. In all the experiments conducted, the highest F1 score FLICoder could achieve was 27.9%. Conversely, FLICoder consistently achieved high accuracies, approximately 97%, across all experiments. This suggests that the datasets used, namely Defects4J and BugsInPy, may not be entirely suitable for fine-tuning and may contain imbalanced data.

As Defects4J and BugsInPy are real-world bug datasets, they have only a few buggy lines in each code sample and the rest of the code lines are non-faulty. It mirrors the fact that in reality as well, we have a small ratio of buggy lines in any carefully written code. However, this leads to an imbalance in the dataset, with a disproportionate number of faulty and non-faulty elements, which hinders the model's ability to equally recognize both classes.

Therefore, there is a need for balanced bug-fixing datasets to more effectively train bug-fixing/fault-localizing models. We are confident that FLICoder's performance could significantly improve when trained on such better datasets.

CHAPTER 5

CONCLUSION

Fault localization is a challenging field of code debugging and there have been continuous efforts by scientists and engineers to automate and optimize it as much as possible. In this work, we contribute towards the same efforts by exploring an LLM-based FL technique and developing the FLICoder models that can perform line-level fault localization.

The variance in FLICoder's performance is explored by varying different aspects of its architecture. With the experiments, we found out that the pre-trained CLM used in FLICoder has a large impact. With CodeGen2 FLICoder achieves a higher precision while with InCoder it achieves a better recall and F1 score. The size of the CLM also contributes to an improvement in FLICoder's performance, although not to a very high extent. We see an improvement of 2% - 4% in the F1 score when switching from the small version of a CLM to a larger version.

The adapter tuning approach is adopted to obtain line-level faultiness scores of the input code. The code is passed through a pre-trained CLM, yielding a hidden states vector. This vector is dimensionally reduced for compatibility with the FLICoder's adapter. It is found that increasing the size of this vector improves performance until it reaches a saturation point, dependent on the adapter size and available training data. Performance improved by 7%-22% when vector size increased from 256 to 512, but did not show significant change between 512 and 1024. In the case of FLICoder with CodeGen2, the recall of the model even decreased by 13.7% when switching from 512 to 1024.

The effect of the dataset used to train the adapter model is also investigated. It is understood that training data plays a very important role in FLICoder's performance.

Further, we compare our technique, FLICoder, which uses a bidirectional attention-based CLM, with the baseline technique, LLMAO, which uses a prefix-only attention-based CLM. We find that FLICoder outperforms LLMAO by 25.5% - 52.4% on the Top-N metric and is able to identify 101 – 207 more bugs in Defects4J out of the total of 395 bugs.

Lastly, as we inspect the results of the experiment, we notice that although FLICoder can perform fault localization up to some extent, we do not get exceptionally high precision, recall, and F1 scores. While we get consistent accuracies of about 98%. This points to possibly an insufficiency of the data for enough fine-tuning and the possibility of an imbalance in the training data. This imbalance is an expected trait in a real-world bug dataset, with a predominantly higher number of non-faulty and a few faulty lines in the code samples. Therefore, we identify the need for a larger and balanced bug dataset to enable more effective training of line-level fault localizing models. We express confidence in the possibility of improved performance of FLICoder when trained on better datasets.

5.1 Future Work

Based on the research outcomes discussed above, we identify two major aspects that can be explored in future work. Firstly, as we saw changes in FLICoder's performance while working with different CLMs, it will be insightful to test it with more CLMs and observe how FLICoder's performance evolves. Secondly, training and testing FLICoder with other large and improved bug datasets is also a promising study that can lead to better results.

REFERENCES

[1] A. Alaboudi and T. D. LaToza, "What constitutes debugging? An exploratory study of debugging episodes," *Empir. Softw. Eng.*, vol. 28, no. 5, p. 117, Sep. 2023.

[2] D. Yang, Y. Qi, X. Mao, and Y. Lei, "Evaluating the usage of fault localization in automated program repair: an empirical study," *Front. Comput. Sci.*, vol. 15, no. 1, p. 151202, Sep. 2020.

[3] Q. Zhu *et al.*, "A syntax-guided edit decoder for neural program repair," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 341–353.

[4] M. Chen *et al.*, "Evaluating Large Language Models Trained on Code," arXiv, Jul. 14, 2021. Accessed: Jun. 04, 2024. [Online]. Available: http://arxiv.org/abs/2107.03374

[5] N. Jiang, K. Liu, T. Lutellier, and L. Tan, "Impact of Code Language Models on Automated Program Repair." arXiv, Apr. 16, 2023. Accessed: Apr. 06, 2024. [Online]. Available: http://arxiv.org/abs/2302.05020

[6] W. J. Gonzalez, "The Relevance of Language for Scientific Research," in *Language and Scientific Research*, W. J. Gonzalez, Ed., Cham: Springer International Publishing, 2021, pp. 1–37.

[7] C. Montemayor, "Language and Intelligence," *Minds Mach.*, vol. 31, no. 4, pp. 471–486, Dec. 2021.

[8] A. L. Guzman and S. C. Lewis, "Artificial intelligence and communication: A Human–Machine Communication research agenda," *New Media Soc.*, vol. 22, no. 1, pp. 70–86, Jan. 2020.

[9] T. Winograd, "Understanding natural language," *Cognit. Psychol.*, vol. 3, no. 1, pp. 1–191, Jan. 1972.

[10] V. Rajaraman, “From ELIZA to ChatGPT,” *Resonance*, vol. 28, no. 6, pp. 889–905, Jun. 2023.

[11] L. Siddharth, L. Blessing, and J. Luo, “Natural language processing in-and-for design research,” *Des. Sci.*, vol. 8, p. e21, Jan. 2022.

[12] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” arXiv, Oct. 11, 2018. Accessed: Apr. 12, 2024. [Online]. Available: https://arxiv.org/abs/1810.04805

[13] H. Naveed *et al.*, “A Comprehensive Overview of Large Language Models,” arXive, Jul. 12, 2023. Accessed: Apr. 12, 2024. [Online]. Available: https://arxiv.org/abs/2307.06435

[14] T. B. Brown *et al.*, “Language Models are Few-Shot Learners,” arXiv, May. 28, 2020. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2005.14165

[15] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, “Language Models are Unsupervised Multitask Learners,” 2019, [Online]. Available: https://d4mucfpksywv.cloudfront.net/better-language-models/language_models_are_unsupervised_multitask_learners.pdf

[16] “Introducing ChatGPT.” Accessed: Apr. 12, 2024. [Online]. Available: https://openai.com/blog/chatgpt

[17] A. Chernyavskiy, D. Ilvovsky, and P. Nakov, “Transformers: ‘The End of History’ for Natural Language Processing?,” in *Machine Learning and Knowledge Discovery in Databases. Research Track: European Conference, ECML PKDD 2021, Bilbao, Spain, September 13–17, 2021, Proceedings, Part III*, Berlin, Heidelberg: Springer-Verlag, Sep. 2021, pp. 677–693.

[18] A. Wang *et al.*, “SuperGLUE: A Stickier Benchmark for General-Purpose Language Understanding Systems,” in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019. Accessed: Apr. 12, 2024. [Online]. Available: https://papers.nips.cc/paper_files/paper/2019/hash/4496bf24afe7fab6f046bf4923da8de6-Abstract.html

[19] D. Adiwardana *et al.*, “Towards a Human-like Open-Domain Chatbot,” arXiv, Jan. 27, 2020. Accessed: Apr. 12, 2024. [Online]. Available: https://arxiv.org/abs/2001.09977

[20] M. Lewis *et al.*, “BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, Online: Association for Computational Linguistics, 2020, pp. 7871–7880.

[21] M. E. Peters *et al.*, “Deep Contextualized Word Representations,” in *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long Papers)*, M. Walker, H. Ji, and A. Stent, Eds., New Orleans, Louisiana: Association for Computational Linguistics, Jun. 2018, pp. 2227–2237.

[22] Z. Zhang *et al.*, “Unifying the Perspectives of NLP and Software Engineering: A Survey on Language Models for Code.” arXiv, Nov. 14, 2023. Accessed: Apr. 13, 2024. [Online]. Available: http://arxiv.org/abs/2311.07989

[23] T. Ahmed, D. Yu, C. Huang, C. Wang, P. Devanbu, and K. Sagae, “Towards Understanding What Code Language Models Learned,” arXiv, Jun. 20, 2023. Accessed: Apr. 13, 2024. [Online]. Available: https://arxiv.org/abs/2306.11943

[24] Y. Pruksachatkun *et al.*, “Intermediate-Task Transfer Learning with Pretrained Language Models: When and Why Does It Work?,” in *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*, D. Jurafsky, J. Chai, N. Schluter, and J. Tetreault, Eds., Online: Association for Computational Linguistics, Jul. 2020, pp. 5231–5247.

[25] R. Wang, S. Si, G. Wang, L. Zhang, L. Carin, and R. Henao, “Integrating Task Specific Information into Pretrained Language Models for Low Resource Fine Tuning,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational Linguistics, Nov. 2020, pp. 3181–3186.

[26] P. Bhattacharya *et al.*, “Exploring Large Language Models for Code Explanation.” arXiv, Oct. 25, 2023. Accessed: Apr. 13, 2024. [Online]. Available: http://arxiv.org/abs/2310.16673

[27] F. F. Xu, U. Alon, G. Neubig, and V. J. Hellendoorn, “A systematic evaluation of large language models of code,” in *Proceedings of the 6th ACM SIGPLAN*

International Symposium on Machine Programming, San Diego CA USA: ACM, Jun. 2022, pp. 1–10.

[28] Y. Wan, W. Zhao, H. Zhang, Y. Sui, G. Xu, and H. Jin, "What do they capture?: a structural analysis of pre-trained language models for source code," in *Proceedings of the 44th International Conference on Software Engineering*, Pittsburgh Pennsylvania: ACM, May 2022, pp. 2377–2388.

[29] A. Soliman, S. Shaheen, and M. Hadhoud, "Leveraging pre-trained language models for code generation," *Complex Intell. Syst.*, Feb. 2024.

[30] N. Chirkova and S. Troshin, "Empirical study of transformers for source code," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 703–715.

[31] E. Nijkamp *et al.*, "CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis." arXiv, Feb. 27, 2023. Accessed: Apr. 06, 2024. [Online]. Available: http://arxiv.org/abs/2203.13474

[32] D. Fried *et al.*, "InCoder: A Generative Model for Code Infilling and Synthesis," arXiv, Apr. 12, 2022. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2204.05999

[33] A. Vaswani *et al.*, "Attention Is All You Need." arXiv, Jun. 12, 2017. Accessed: Apr. 06, 2024. [Online]. Available: http://arxiv.org/abs/1706.03762

[34] J. J. Webster and C. Kit, "Tokenization as the Initial Phase in NLP," in *COLING 1992 Volume 4: The 14th International Conference on Computational Linguistics*, 1992. Accessed: Apr. 13, 2024. [Online]. Available: https://aclanthology.org/C92-4173

[35] R. Sennrich, B. Haddow, and A. Birch, "Neural Machine Translation of Rare Words with Subword Units," in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds., Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 1715–1725.

[36] T. Kudo, "Subword Regularization: Improving Neural Network Translation Models with Multiple Subword Candidates," in *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long*

Papers), I. Gurevych and Y. Miyao, Eds., Melbourne, Australia: Association for Computational Linguistics, Jul. 2018, pp. 66–75.

[37] S. J. Mielke *et al.*, "Between words and characters: A Brief History of Open-Vocabulary Modeling and Tokenization in NLP." arXiv, Dec. 20, 2021. Accessed: Apr. 13, 2024. [Online]. Available: http://arxiv.org/abs/2112.10508

[38] A. Kazemnejad, I. Padhi, K. N. Ramamurthy, P. Das, and S. Reddy, "The Impact of Positional Encoding on Length Generalization in Transformers." arXiv, Nov. 06, 2023. Accessed: Apr. 13, 2024. [Online]. Available: http://arxiv.org/abs/2305.19466

[39] K. Clark, M.-T. Luong, Q. V. Le, and C. D. Manning, "ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators." arXiv, Mar. 23, 2020. Accessed: Apr. 14, 2024. [Online]. Available: http://arxiv.org/abs/2003.10555

[40] L. Dong *et al.*, "Unified Language Model Pre-training for Natural Language Understanding and Generation," in *Advances in Neural Information Processing Systems*, Curran Associates, Inc., 2019. Accessed: Apr. 14, 2024. [Online]. Available: https://proceedings.neurips.cc/paper/2019/hash/c20bb2d9a50d5ac1f713f8b34d9aac5a-Abstract.html

[41] L. Zhuang, L. Wayne, S. Ya, and Z. Jun, "A Robustly Optimized BERT Pre-training Approach with Post-training," in *Proceedings of the 20th Chinese National Conference on Computational Linguistics*, S. Li, M. Sun, Y. Liu, H. Wu, K. Liu, W. Che, S. He, and G. Rao, Eds., Huhhot, China: Chinese Information Processing Society of China, Aug. 2021, pp. 1218–1227. Accessed: Apr. 14, 2024. [Online]. Available: https://aclanthology.org/2021.ccl-1.108

[42] Y. Liu *et al.*, "RoBERTa: A Robustly Optimized BERT Pretraining Approach." arXiv, Jul. 26, 2019. Accessed: Apr. 14, 2024. [Online]. Available: http://arxiv.org/abs/1907.11692

[43] K. Song, X. Tan, T. Qin, J. Lu, and T.-Y. Liu, "MASS: Masked Sequence to Sequence Pre-training for Language Generation," in *Proceedings of the 36th International Conference on Machine Learning*, PMLR, May 2019, pp. 5926–5936. Accessed: Apr. 14, 2024. [Online]. Available: https://proceedings.mlr.press/v97/song19d.html

[44] S. Black, G. Leo, P. Wang, C. Leahy, and S. Biderman, “GPT-Neo: Large Scale Autoregressive Language Modeling with Mesh-Tensorflow.” Zenodo, Mar. 21, 2021. Accessed: Apr. 07, 2024. [Online]. Available: https://zenodo.org/records/5297715

[45] Ben Wang and Aran Komatsuzaki, “GPT-J-6B: A 6 Billion Parameter Autoregressive Language Model.” 2021. [Online]. Available: https://github.com/kingoflolz/mesh-transformer-jax

[46] S. Black *et al.*, “GPT-NeoX-20B: An Open-Source Autoregressive Language Model,” in *Proceedings of BigScience Episode #5 -- Workshop on Challenges & Perspectives in Creating Large Language Models*, virtual+Dublin: Association for Computational Linguistics, 2022, pp. 95–136.

[47] Z. Feng *et al.*, “CodeBERT: A Pre-Trained Model for Programming and Natural Languages,” in *Findings of the Association for Computational Linguistics: EMNLP 2020*, T. Cohn, Y. He, and Y. Liu, Eds., Online: Association for Computational Linguistics, Nov. 2020, pp. 1536–1547.

[48] A. Kanade, P. Maniatis, G. Balakrishnan, and K. Shi, “Learning and evaluating contextual embedding of source code,” in *Proceedings of the 37th International Conference on Machine Learning*, in ICML’20, vol. 119. JMLR.org, Jul. 2020, pp. 5110–5121.

[49] OpenAI *et al.*, “GPT-4 Technical Report,” 2023.

[50] S. Kang, B. Chen, S. Yoo, and J.-G. Lou, “Explainable Automated Debugging via Large Language Model-driven Scientific Debugging.” arXiv, Apr. 04, 2023. Accessed: Apr. 06, 2024. [Online]. Available: http://arxiv.org/abs/2304.02195

[51] C. Raffel *et al.*, “Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer,” *J. Mach. Learn. Res.*, no. 21, pp. 1–67, 2020.

[52] L. Xue *et al.*, “mT5: A Massively Multilingual Pre-trained Text-to-Text Transformer,” in *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, K. Toutanova, A. Rumshisky, L. Zettlemoyer, D. Hakkani-Tur, I. Beltagy, S. Bethard, R. Cotterell, T. Chakraborty, and Y. Zhou, Eds., Online: Association for Computational Linguistics, Jun. 2021, pp. 483–498.

[53] A. Z. H. Yang, C. Le Goues, R. Martins, and V. Hellendoorn, “Large Language Models for Test-Free Fault Localization,” in *Proceedings of the 46th IEEE/ACM*

International Conference on Software Engineering, Lisbon Portugal: ACM, Feb. 2024, pp. 1–12.

[54] R. Abreu, P. Zoeteweij, and A. J. C. van Gemund, "On the Accuracy of Spectrum-based Fault Localization," in *Testing: Academic and Industrial Conference Practice and Research Techniques - MUTATION (TAICPART-MUTATION 2007)*, Sep. 2007, pp. 89–98.

[55] X. Li, W. Li, Y. Zhang, and L. Zhang, "DeepFL: integrating multiple fault diagnosis dimensions for deep fault localization," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, in ISSTA 2019. New York, NY, USA: Association for Computing Machinery, Jul. 2019, pp. 169–180.

[56] Y. Lou *et al.*, "Boosting coverage-based fault localization via graph-based representation learning," in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2021. New York, NY, USA: Association for Computing Machinery, Aug. 2021, pp. 664–676.

[57] Y. Li, S. Wang, and T. N. Nguyen, "Fault localization to detect co-change fixing locations," in *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, in ESEC/FSE 2022. New York, NY, USA: Association for Computing Machinery, Nov. 2022, pp. 659–671.

[58] J. A. Jones and M. J. Harrold, "Empirical evaluation of the tarantula automatic fault-localization technique," in *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering*, in ASE '05. New York, NY, USA: Association for Computing Machinery, Nov. 2005, pp. 273–282.

[59] R. Abreu, P. Zoeteweij, and A. J. c. Van Gemund, "An Evaluation of Similarity Coefficients for Software Fault Localization," in *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Dec. 2006, pp. 39–46.

[60] L. Zhang, L. Zhang, and S. Khurshid, "Injecting mechanical faults to localize developer faults for evolving software," in *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications*, in OOPSLA '13. New York, NY, USA: Association for Computing Machinery, Oct. 2013, pp. 765–784.

[61] S. Moon, Y. Kim, M. Kim, and S. Yoo, “Ask the Mutants: Mutating Faulty Programs for Fault Localization,” in *Verification and Validation 2014 IEEE Seventh International Conference on Software Testing*, Mar. 2014, pp. 153–162.

[62] T. T. Chekam, M. Papadakis, and Y. L. Traon, “Assessing and Comparing Mutation-based Fault Localization Techniques.” arXiv, Jul. 19, 2016.

[63] Y. Li, S. Wang, and T. N. Nguyen, “Fault Localization with Code Coverage Representation Learning,” in *Proceedings of the 43rd International Conference on Software Engineering*, in ICSE ’21. Madrid, Spain: IEEE Press, Nov. 2021, pp. 661–673.

[64] X. Meng, X. Wang, H. Zhang, H. Sun, and X. Liu, “Improving fault localization and program repair with deep semantic features and transferred knowledge,” in *Proceedings of the 44th International Conference on Software Engineering*, in ICSE ’22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1169–1180.

[65] C. S. Xia, Y. Wei, and L. Zhang, “Practical Program Repair in the Era of Large Pre-trained Language Models,” arXiv, Oct. 25, 2022. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2210.14179

[66] P. Vaithilingam, T. Zhang, and E. L. Glassman, “Expectation vs. Experience: Evaluating the Usability of Code Generation Tools Powered by Large Language Models,” in *Extended Abstracts of the 2022 CHI Conference on Human Factors in Computing Systems*, in CHI EA ’22. New York, NY, USA: Association for Computing Machinery, Apr. 2022, pp. 1–7.

[67] V. Liventsev, A. Grishina, A. Härmä, and L. Moonen, “Fully Autonomous Programming with Large Language Models,” in *Proceedings of the Genetic and Evolutionary Computation Conference*, Lisbon Portugal: ACM, Jul. 2023, pp. 1146–1155.

[68] X. Chen, M. Lin, N. Schärli, and D. Zhou, “Teaching Large Language Models to Self-Debug,” 2023.

[69] J. Kaplan *et al.*, “Scaling Laws for Neural Language Models.” arXiv, Jan. 22, 2020. Accessed: Apr. 15, 2024. [Online]. Available: http://arxiv.org/abs/2001.08361

[70] Y. Li, S. Wang, and T. N. Nguyen, “DEAR: a novel deep learning-based approach for automated program repair,” in *Proceedings of the 44th International*

Conference on Software Engineering, in ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 511–523.

[71] Y. Wu, Z. Li, J. M. Zhang, M. Papadakis, M. Harman, and Y. Liu, "Large Language Models in Fault Localisation," arXiv, Aug. 29, 2023. Accessed: Apr. 09, 2024. [Online]. Available: https://arxiv.org/abs/2308.15276

[72] L. Tunstall, L. von Werra, T. Wolf, and A. Géron, *Natural language processing with transformers: building language applications with hugging face*, Revised edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2022.

[73] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "Unified Pre-training for Program Understanding and Generation," arXiv, Mar. 10, 2021. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2103.06333

[74] Y. Wang, W. Wang, S. Joty, and S. C. H. Hoi, "CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation," arXiv, Sep. 02, 2021. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2109.00859

[75] L. Gao *et al.*, "The Pile: An 800GB Dataset of Diverse Text for Language Modeling," arXiv, Dec. 31, 2021. Accessed: Apr. 08, 2024. [Online]. Available: https://arxiv.org/abs/2101.00027

[76] M. Zeng, Y. Wu, Z. Ye, Y. Xiong, X. Zhang, and L. Zhang, "Fault Localization via Efficient Probabilistic Modeling of Program Semantics," in *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, May 2022, pp. 958–969.

[77] A. Zeller, *Why programs fail: a guide to systematic debugging*, 2nd ed. Amsterdam ; Boston: Elsevier/Morgan Kaufmann, 2009.

[78] L. Ouyang *et al.*, "Training language models to follow instructions with human feedback," arXiv, Mar. 04, 2022. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2203.02155

[79] R. Li *et al.*, "StarCoder: may the source be with you!," arXiv, May. 09, 2023. Accessed: Apr. 07, 2024. [Online]. Available: https://arxiv.org/abs/2305.06161

[80] T. Yu *et al.*, "Spider: A Large-Scale Human-Labeled Dataset for Complex and Cross-Domain Semantic Parsing and Text-to-SQL Task," in *Proceedings of the*

2018 Conference on Empirical Methods in Natural Language Processing, E. Riloff, D. Chiang, J. Hockenmaier, and J. Tsujii, Eds., Brussels, Belgium: Association for Computational Linguistics, Oct. 2018, pp. 3911–3921.

[81] J. Austin *et al.*, "Program Synthesis with Large Language Models." arXiv, Aug. 15, 2021. Accessed: May. 28, 2024. [Online]. Available: http://arxiv.org/abs/2108.07732

[82] M.-A. Lachaux, B. Roziere, L. Chanussot, and G. Lample, "Unsupervised Translation of Programming Languages." arXiv, Sep. 22, 2020. Accessed: May 28, 2024. [Online]. Available: http://arxiv.org/abs/2006.03511

[83] Y. Li *et al.*, "Competition-level code generation with AlphaCode," *Science*, vol. 378, no. 6624, pp. 1092–1097, Dec. 2022.

[84] T. Helmuth and P. Kelly, "PSB2: the second program synthesis benchmark suite," in *Proceedings of the Genetic and Evolutionary Computation Conference*, Lille France: ACM, Jun. 2021, pp. 785–794.

[85] S. Kang, G. An, and S. Yoo, "A Preliminary Evaluation of LLM-Based Fault Localization," arXiv, Aug. 10, 2023. Accessed: Apr. 09, 2024. [Online]. Available: https://arxiv.org/abs/2308.05487

[86] N. Jiang, T. Lutellier, Y. Lou, L. Tan, D. Goldwasser, and X. Zhang, "KNOD: Domain Knowledge Distilled Tree Decoder for Automated Program Repair," in *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, Melbourne, Australia: IEEE, May 2023, pp. 1251–1263.

[87] H. Ye, M. Martinez, and M. Monperrus, "Neural program repair with execution-based backpropagation," in *Proceedings of the 44th International Conference on Software Engineering*, in ICSE '22. New York, NY, USA: Association for Computing Machinery, Jul. 2022, pp. 1506–1518.

[88] N. Jiang, T. Lutellier, and L. Tan, "CURE: Code-Aware Neural Machine Translation for Automatic Program Repair," in *Proceedings of the 43rd International Conference on Software Engineering*, in ICSE '21. Madrid, Spain: IEEE Press, Nov. 2021, pp. 1161–1173.

[89] V. P. Dwivedi and X. Bresson, "A Generalization of Transformer Networks to Graphs," arXiv, Dec. 17, 2020. Accessed: Apr. 08, 2024. [Online]. Available: https://arxiv.org/abs/2012.09699

[90] Z. Hu, X. Ma, Z. Liu, E. Hovy, and E. Xing, “Harnessing Deep Neural Networks with Logic Rules,” in *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, K. Erk and N. A. Smith, Eds., Berlin, Germany: Association for Computational Linguistics, Aug. 2016, pp. 2410–2420.

[91] A. Radford, K. Narasimhan, T. Salimans, and I. Sutskever, “Improving Language Understanding by Generative Pre-Training,” OpenAI Blog, 2018. Accessed: Apr. 08, 2024. [Online]. Available: https://cdn.openai.com/research-covers/language-unsupervised/language_understanding_paper.pdf

[92] S. Hochreiter and J. Schmidhuber, “Long Short-term Memory,” *Neural Comput.*, vol. 9, pp. 1735–80, Dec. 1997.

[93] R. Just, D. Jalali, and M. D. Ernst, “Defects4J: a database of existing faults to enable controlled testing studies for Java programs,” in *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, in ISSTA 2014. New York, NY, USA: Association for Computing Machinery, Jul. 2014, pp. 437–440.

[94] R. Widyasari *et al.*, “BugsInPy: a database of existing bugs in Python programs to enable controlled testing and debugging studies,” in *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, Virtual Event USA: ACM, Nov. 2020, pp. 1556–1560.

[95] D. Lin, J. Koppel, A. Chen, and A. Solar-Lezama, “QuixBugs: a multi-lingual program repair benchmark set based on the quixey challenge,” in *Proceedings Companion of the 2017 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, in SPLASH Companion 2017. New York, NY, USA: Association for Computing Machinery, Oct. 2017, pp. 55–56.

[96] E. Nijkamp, H. Hayashi, C. Xiong, S. Savarese, and Y. Zhou, “CodeGen2: Lessons for Training LLMs on Programming and Natural Languages.” arXiv, Jul. 11, 2023. Accessed: May. 04, 2024. [Online]. Available: http://arxiv.org/abs/2305.02309

[97] J. Hoffmann *et al.*, “Training Compute-Optimal Large Language Models.” arXiv, Mar. 29, 2022. Accessed: May. 09, 2024. [Online]. Available: http://arxiv.org/abs/2203.15556

[98] C. Parnin and A. Orso, “Are automated debugging techniques actually helping programmers?,” in *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, in ISSTA ’11. New York, NY, USA: Association for Computing Machinery, Jul. 2011, pp. 199–209.

[99] P. S. Kochhar, X. Xia, D. Lo, and S. Li, “Practitioners’ expectations on automated fault localization,” in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, in ISSTA 2016. New York, NY, USA: Association for Computing Machinery, Jul. 2016, pp. 165–176.

[100] “colab.google,” colab.google. Accessed: May 10, 2024. [Online]. Available: http://0.0.0.0:8080/

APPENDIX A

CODE USED TO OBTAIN THE PRESENTED RESULTS

1) For Obtaining Precision, Recall, and Accuracy

```
def recall_prec_function(self, predictions, label, mask):
    """
    Computes masked prediction accuracies

    Args:
        predictions (2D float): model predictions
        labels (2D binary float): ground truth labels
        mask (2D binary float): mask

    Returns:
        1D float: recall, precision, & accuracy per batch
    """

    # Calculate the total number of bugs, add a small
value to avoid division by zero
    num_bugs = torch.sum(label * mask) + 1e-6

    # Apply sigmoid function and round off to get binary
predictions
    sigmoid_prediction =
torch.round(torch.sigmoid(predictions))

    # Check where the predictions match the labels
    corrects = torch.eq(label, sigmoid_prediction)

    # Find the true positives: where the prediction is
correct and the mask is true
    true_positive = torch.logical_and(mask, corrects)
    true_positive = true_positive * label

    # Calculate the accuracy: where the prediction is
correct and the mask is true
    accuracies = torch.logical_and(mask, corrects)
```

```
    # Calculate the final accuracy as the mean of the
accuracies
    final_acc = torch.mean(torch.sum(accuracies) /
torch.sum(mask))

    # Calculate recall: ratio of true positives to the
total number of bugs
    recall = torch.sum(true_positive) / num_bugs

    # Calculate precision: ratio of true positives to the
total number of positive predictions
    precision = torch.sum(true_positive) /
torch.sum(sigmoid_prediction * mask + 1e-6)

    # Return recall, precision, and final accuracy as
numpy arrays
    return (
        recall.cpu().detach().numpy(),
        precision.cpu().detach().numpy(),
        final_acc.cpu().detach().numpy(),
    )  # 1D float
```

2) For Calculating Top-N

```
def top_scores(probabilities, labels):
    # Define the number of top scores to consider
    n_tops = [5, 3, 1]

    data_split = 15
    window_split = 14
    prob_cutoffs = [0.05, 0.1, 0.2]

    # Loop over the probability cutoffs
    for i, prob_cutoff in enumerate(prob_cutoffs):
        n_top = n_tops[i]
        label_bug_counter = 0
        predicted_bug_counter = 0
        hit_counter = 0

        # Split the probabilities and labels into
projects
        split_probs_proj = np.array_split(probabilities,
data_split)
        split_labels_proj = np.array_split(labels,
data_split)
```

```
        # Loop over the projects
        for proj_idx in range(len(split_probs_proj)):
            prob_project = split_probs_proj[proj_idx]
            label_project = split_labels_proj[proj_idx]

            # Split the project probabilities and labels
into bugs
            split_probs_bugs =
np.array_split(prob_project, window_split)
            split_labels_bugs =
np.array_split(label_project, window_split)

            # Loop over the bugs
            for bug_idx in range(len(split_probs_bugs)):
                prob_bug = split_probs_bugs[bug_idx]
                label_bug = split_labels_bugs[bug_idx]

                # Count the number of bugs
                label_bug_counter += sum(label_bug)

                # Apply the probability cutoff
                prob_bug = list(
                    map(lambda x: 1.0 if x > prob_cutoff
else 0.0, prob_bug)
                )

                # Count the number of predicted bugs
                predicted_bug_counter += sum(prob_bug)

                # Find the intersection of predicted and
actual bugs
                intersection = [
                    prob_bug.index(n)
                    for m, n in zip(prob_bug, label_bug)
                    if (n == m and n == 1.0)
                ]

                # Count the number of correct predictions
                correct_preds = len(intersection)

                # If there are any correct predictions,
increment the hit counter
                if correct_preds > 0:
                    hit_counter += 1

        # Calculate the average number of bugs
        label_bug_counter = round(label_bug_counter /
window_split)
```

```
        # Assign the hit counter to the appropriate top
score
        if n_top == 5:
            top_5 = hit_counter
        elif n_top == 3:
            top_3 = hit_counter
        elif n_top == 1:
            top_1 = hit_counter

        # Reset the bug counter
        label_bug_counter = 0

    # Return the top scores
    return top_5, top_3, top_1
```

APPENDIX B

EFFECT OF BATCH SIZE ON FLICODER PERFORMANCE

In the following table, a comparison of the FLICoder (with CodeGen2-3.7B) performance is presented by using different batch sizes for training with Defects4J and BugsInPy. Here an adapter dimension size of 512 was used.

	Defects4J		**BugsInPy**	
Batch Size	**32**	**8**	**32**	**8**
Precision	77.0%	21.3%	49.9%	15.8%
Recall	14.1%	17.2%	0.9%	3.3%
F1 Score	23.8%	18.9%	3.6%	7.9%
Accuracy	96.8%	95.8%	98.4%	97.6%