IMPROVING BUG LOCALIZATION LEVERAGING LARGE LANGUAGE MODELS' REASONING WITH INFORMATION RETRIEVAL

by

Asif M Samir

Submitted in partial fulfillment of the requirements
for the degree of PhD of Computer Science

at

Dalhousie University
Halifax, Nova Scotia
April 2025

I dedicate this thesis to my parents, whose inspiration and unwavering support have guided me through every step of my life.

Table of Contents

List of Tables

List of Figures

Abstract

Despite decades of research, locating software bugs remains a challenging task, as suggested by recent surveys. Practitioners spend nearly 50% of their time dealing with software bugs and failures. Many existing techniques use traditional methods such as Information Retrieval (IR) to localize software bugs by leveraging textual and semantic features from bug reports and source code. However, these techniques often fail to bridge critical gaps between bug reports and source code, which require an in-depth, comprehensive understanding of software bugs. In other words, these techniques struggle when the localization of a bug requires going beyond simple textual or semantic matching. To address these gaps in the literature, we conduct two independent but complementary studies that attempt to enhance IR-based bug localization by leveraging the reasoning capabilities of Large Language Models (LLM). In our first study, we fine-tune a pre-trained language model (e.g., CodeBERT) using buggy and bug-free source code and leverage its reasoning capability to enhance IR-based bug localization. Unlike existing works, we incorporate our model's reasoning about a bug into query construction and document reranking during bug localization. Our evaluation using three performance metrics and ≈7.5K bug reports shows that our technique achieves improvements of 60.49% in MAP, 64.58% in MRR, and 100.90% in HIT@K compared to baseline methods. In our second study, we advance IR-based bug localization by leveraging expert-like feedback from LLMs to enhance search queries. In particular, we capture Intelligent Relevance Feedback (IRF) from LLMs (e.g., Mistral) against each query and leverage it to reformulate the query and thus improve the result ranks. Our experiments using ≈4.7K bug reports and three performance metrics show that our technique improves bug localization by 87.6% in MAP, 89.5% in MRR, and 48.8% in HIT@K against the baseline techniques. Furthermore, it can localize ≈52% of bugs that the baseline techniques cannot localize due to the poor quality of the corresponding bug reports. Given the above evidence, our work has strong potential to advance IR-based bug localization leveraging the reasoning capabilities of LLMs.

List of Abbreviations

AI	Artificial Intelligence
AST	Abstract Syntax Tree
BLEU	Bi-Lingual Evaluation of Understudy
BPE	Byte-Pair Encoding
BPTT	Backpropagation Through Time
CE	Cross Encoder
CFG	Control Flow Graph
CNN	Convolutional Neural Network
DFG	Data Flow Graph
DL	Deep Learning
EB	Expected Behavior
Es	Elasticsearch
FN	False Negative
FP	False Positive
FPR	False-Positive Rate
GRU	Gated Recurrent Unit
HF	Hugging Face
IR	Information Retrieval
IRBL	Information Retrieval based Bug Localization
IRF	Intelligent Relevance Feedback
ITS	Issue Tracking System
LLM	Large Language Model
LM	Language Model
LSTM	Long Short-term Memory
ML	Machine Learning
NLM	Neural Language Modeling
NL	Natural Language
NLP	Natural Language Processing

NMT	Neural Machine Translation
OB	Observed Behavior
PDG	Program Dependency Graph
PE	Program Elements
PR	Pull Request
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
RPE	Relative Positional Embedding
ST	Stack Trace
S2R	Steps to Reproduce
T5	Text-To-Text Transfer Transformer
TN	True Negative
TNR	True-Negative Rate
TP	True Positive
TPR	True-Positive Rate

Acknowledgements

First, I thank the Almighty, the most gracious and the most merciful, who granted me the capability to carry out this work. Then I would like to express my heartiest gratitude to my supervisor, Dr. Masud Rahman, for his constant guidance, advice, encouragement, and extraordinary patience during this research. Without his constant support, this work would have been impossible.

I would also like to express my sincere gratitude to Dr. Evangelos E. Milios, Dr. Tushar Sharma and Dr. Ga Wu for their invaluable advisement and meticulous evaluation of my RAD report. Their scholarly perspectives and thoughtful input have elevated the quality and rigor of my research, and I am truly appreciative of their time and expertise.

Thanks to all of the members of the Intelligent Automation in Software EngineeRing (RAISE) Lab, with whom I have had the opportunity to grow as a researcher. In particular, I would like to thank Sigma Jahan, Mehil Shah, Usmi Mukherjee, and Riasat Mahbub, as well as past members of the lab— Parvez Mahbub and Ohiduzzaman Shuvo.

I am grateful to Dalhousie University and its Computer Science Department for their generous financial support through scholarships, awards, and bursaries that helped me to concentrate more deeply on my thesis work. In particular, I would like to thank Dr. Michael McAllister, Megan Baker and Vidhya Ramamoorthy.

I would like to express my deepest gratitude to my parents, my sister, and my better half, Jannat, for their unwavering support and love throughout my academic journey. Their constant encouragement, wise guidance, and countless sacrifices have been instrumental in shaping my path.

Lastly, I extend my heartfelt thanks to all those who have supported me in various ways throughout this research endeavor. Your encouragement, advice, and assistance have been invaluable, and I am deeply grateful for your presence in my life.

Chapter 1

Introduction

1.1 Motivation

Software bugs are flaws or errors in a software system that produce incorrect results or unexpected behavior [77]. The impact of software bugs can be severe, leading to substantial financial losses, data breaches, security vulnerabilities, and operational disruptions [41, 56]. In 2022 alone, software maintenance cost the U.S. economy $2.4 trillion, an increase of approximately 84% over the previous two years [4]. More recently, a software bug in Microsoft-owned CrowdStrike's systems caused several hours of disruption in the U.S. airline industry, incurring over $10 billion in damages [154, 170, 174]. Addressing such bugs requires considerable effort, with developers spending 35-50% of their time on bug resolution [25, 115]. A recent survey by Zou et al. [187] involving 327 software practitioners (e.g., developers, project managers, testers) from major IT companies (e.g., Google, Meta, Microsoft, Amazon) suggests that 82.4% of respondents considered bug finding (a.k.a., bug localization) as an important or a highly important task.

Bug localization refers to a process of identifying the specific location (e.g., file, module, or line of code) in a software system where a bug (error) might be present [127, 136]. Software bugs are typically submitted to bug-tracking systems (e.g., Bugzilla, JIRA) as bug reports, which might capture crucial hints for their resolution. Developers often rely on these reports to localize bugs in the code. Given the challenges in localizing bugs, the existing literature has witnessed the emergence of two major types of automated methods: Spectrum based fault localization (SBFL) and Information Retrieval based bug localization (IRBL). Spectrum-based fault localization [94, 110, 162] employs test cases and captures the execution traces of a software system for localizing software bugs or faults. By comparing the execution traces between passing and failing tests, SFBL determines a program element's likelihood of being faulty [94]. However, the execution traces are not always readily

accessible, making these methods less scalable [94]. On the other hand, Information Retrieval (IR)-based methods rely on the keyword overlaps between bug reports and source code to localize bugs [91, 94, 114, 163, 167]. Although IR-based methods are lightweight and scalable, they may not always deliver satisfactory results due to sporadic term matching between bug reports and source code.

1.2 Problem Statement

Although IR-based techniques for bug localization are lightweight, they suffer from *vocabulary mismatch problems* [60]. They rely on textual similarity and often do not consider the context and semantics of the source code when matching terms from bug reports. Over the last few decades, many approaches have been proposed to mitigate these issues. Researchers have incorporated historical data from past bug reports, code change history, past bug fixes, and author history into bug localization [164, 181]. Even though they provide additional contexts for a software bug, these approaches primarily rely on statistical chances, which might not be always high. For instance, many bugs might not have enough historical information to glean insights from, which could limit their effectiveness in bug localization. A recent study [96] also suggests that these approaches, despite their added contexts from historical data, do not significantly outperform the previous approaches for bug localization.

Recent IR-based techniques for bug localization attempt to improve their queries by capturing syntactic, co-occurrence, and hierarchical relationships among the words in bug reports [30, 127, 129, 143]. However, these methods only use terms found in bug reports, which could be poorly written or deficient [127]. As a result, they might also fail to bridge the gap between natural language from bug reports and source code from a project when locating software bugs. To address this issue, several techniques attempt to enhance their queries with relevant terms captured from source code documents through relevance feedback mechanisms [61, 68, 85, 98, 129, 161, 182]. However, the majority of these techniques naively consider the top few documents (based on textual similarity) as relevant, overlooking the need for a comprehensive understanding of the code. As a result, they may not always capture meaningful terms from source code for their search queries. [30, 85]. Thus, the existing IR-based techniques for bug localization suffer from two major challenges as follows.

(a) Textual and semantic similarity might not be sufficient: Bug reports contain not only natural language texts but also technical jargons, commit diffs, stack traces, and program elements [129]. Although these structured artifacts hold important clues and symptoms of encountered bugs, the majority of bug reports still contain natural language texts [30]. Since natural language is loosely structured, it can introduce ambiguity by expressing the same idea in various ways [60]. On the other hand, programming languages are more structured but still allow syntactically diverse expressions (e.g., iterative vs. functional approaches) and arbitrary naming conventions [14, 43, 144]. Such variability in text or code can result in textual mismatches, where keywords or key phrases in the bug report (e.g., "*download failed*") do not directly match the identifiers in the code (e.g., `fetchResource`). At the same time, semantic mismatches can arise when a problem discussed in the bug report does not correspond to the programming task implemented in the code. For example, the reported problem – "*download failed*" – might not align well with the programming task – "*HTTP/FTP operation task and get packets*" if the word-level semantics are considered only. It requires an understanding of the relationship between network operations and file downloading to establish their connection. In other words, to localize such bugs, automated tools or methods need to go beyond surface-level similarity and comprehensively understand the context of an encountered problem as well as the functionality of the corresponding source code. They might also need to capitalize on the strengths of multiple types of methods to design a comprehensive solution. To the best of our knowledge, existing work on bug localization overlooks such aspects, indicating a major gap in the literature.

(b) Relevance feedback against search queries might not be always relevant: Gay et al. [68] proposed a manual, iterative approach that leverages relevance feedback from developers and constructs queries to search for buggy source documents. In contrast, Sisman et al. [144] and Kim et al. [85] select the top few source documents as relevant (a.k.a. pseudo-relevance feedback) and leverage the feedback to improve their search queries. However, these techniques rely heavily on textually similar documents, which may not be always relevant, especially when bug reports are used as queries. Thus, a deeper understanding of both bug reports and source code is warranted to improve the relevance feedback mechanism and the subsequent

steps of Information Retrieval (e.g., query reformulation, retrieval).

1.3 Our Contribution

In this RAD report, we conduct two independent but complementary studies to fill the above gaps. In these studies, we propose two novel techniques for bug localization leveraging the reasoning capabilities of the Large Language Models (LLMs) as follows.

In our first study, we propose – *IQLoc* – a novel hybrid approach that capitalizes on the strengths of both Information Retrieval (IR) and LLM-based program understanding to support bug localization. The sole reliance on textual relevance during bug localization could lead to a large search space. Our technique narrows it down by incorporating a deeper understanding of *program semantics* [47, 73, 175] into IR leveraging the transformer-based models. First, IQLoc retrieves the top K source documents relevant to a bug report using a widely used IR-based approach (e.g., BM25 algorithm [133]). Second, it constructs appropriate search queries and retrieves relevant source documents leveraging the program semantics understanding of transformer-based models (e.g., cross-encoders). Finally, our technique reranks the retrieved source documents based on their overall relevance and perceived semantics by the LLM.

We refined and extended an existing benchmark dataset – Bench4BL [96] – for our experiments. To refine the dataset, we selected the bug reports that include version information and have corresponding relevant documents in their respective GitHub repositories. We also expanded it by incorporating ≈30% recent bug reports submitted between 2018 and 2024, resulting in a final set of 7,483 bug reports. To evaluate our proposed technique, we employed three appropriate and widely used metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. We compare our technique with four appropriate baselines from the literature – BLUiR [136], Blizzard [129], DNNLoc [92] and RLocator [29]. Across various measures, IQLoc consistently outperformed these techniques, with improvements of up to 58.52% and 60.59% in MAP, 61.49% and 64.58% in MRR, and 69.88% and 100.90% in HIT@K for the test bug reports with random and time-wise splits, respectively. Furthermore, IQLoc improves MAP by 91.67% for bug reports with stack

traces, 72.73% for those with code elements, and 65.38% for those with natural language texts only. These results underscore the benefits and technical superiority of our proposed approach.

While our first technique effectively integrates IR and transformer-based language models for bug localization, it struggles to detect bugs for natural language-only bug reports. In our second study, we propose – *BRaIn* – a novel transformer-based technique that captures expert-like feedback from Large Language Models (LLMs) for a given query (i.e., bug report) to enhance bug localization. Unlike existing approaches, BRaIn leverages Intelligent Relevance Feedback (IRF) from LLMs to refine its queries and rerank the source documents using a feedback-driven scoring mechanism. First, BRaIn identifies potentially buggy documents from a codebase by analyzing their contextual relevance (i.e., IRF) to a reported bug using transformer models (e.g., Mistral [151]). Second, it then extracts appropriate terms from these documents and further expands the original query. Finally, BRaIn reranks the documents by executing the expanded query and applying the relevance feedback, providing a refined list of suspicious source documents.

We used the curated dataset, Bench4BL [96], for our experiments. Since our focus was on capturing relevance feedback, we target the bug reports containing only natural language texts without any program artifacts (e.g., stack traces), which left us with 4,683 bug reports. We evaluated the performance of our approach using three commonly used metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. Our approach is compared with six suitable baselines from the literature – BLUiR [136], Blizzard [129], Sysman-SCP [68], DNNLOC [92], NextBug [180] and RLocator [29]. BRaIn consistently outperformed these existing techniques, showing 19.3% and 87.6% higher MAP scores than that of traditional and Machine Learning (ML)-based approaches, respectively. Similar gains were observed in MRR (17.5% and 89.5%) and HIT@10 (12.2% and 48.8%). Additionally, we evaluated BRaIn's ability to localize bugs affecting multiple documents and observed improvements of 7.0-10.6% in MAP, 7.9-10.6% in MRR, and 6-6.9% in HIT@10. BRaIn is also capable of addressing poor-quality bug reports and successfully localizing ≈52% of bugs that baseline techniques cannot handle. All these results demonstrate the effectiveness and superiority of our proposed technique in software bug localization.

1.4 Related Publications

Several parts of this work have been accepted at and submitted to different conferences and journals. We provide a list of related publications here. In each of these papers, I am the primary author who conducted all the studies under the supervision of Dr. Masud Rahman. While I wrote these papers, the co-author took part in ideation, advising, editing, and reviewing the papers.

- *Asif Mohammed Samir* and Mohammad Masudur Rahman. *Improved IR-based Bug Localization with Intelligent Relevance Feedback.* In Proceedings of the International Conference on Program Comprehension (ICPC 2025), Ottawa, Canada, 2025. (In Press)

Apart from the aforementioned paper, our another paper has been submitted to a major software engineering journal.

- *Asif Mohammed Samir* and Mohammad Masudur Rahman. *Improving IR-Based Bug Localization through Program Semantics-Driven Intelligent Query Reformulation.* Journal of Systems and Software (JSS), 2025.

1.5 Outline of the Report

The document contains five chapters in total. To localize bug effectively and efficiently, we conduct two independent but complementary studies, and this section outlines different chapters as follows.

- Chapter 1 motivates the research problems, outlines our research contributions, and discusses related publications.
- Chapter 2 discusses several background concepts (e.g., bug localization, embeddings, cross-encoder) that will be required to follow the rest of the document.
- Chapter 3 introduces IQLoc, a novel hybrid technique that capitalizes on the strengths of both Information Retrieval and LLM-based code understanding to improve bug localization.

- Chapter 4 presents BRaIn, that advances bug localization by leveraging Intelligent Relevance Feedback (IRF).
- Chapter 5 concludes the RAD report with a list of directions for future works.

Chapter 2

Background

In this chapter, we introduce the required terminologies and concepts to follow the rest of the report. Section 2.1 provides an overview of the program semantics. Section 2.2 and Section 2.3 introduce basic, widely used concepts such as Information Retrieval and Embeddings. Section 2.4 covers sequence modeling with RNN, LSTM, GRU, attention mechanisms, and transformers, which are popular neural architectures for handling sequential data. Section 2.5 explains Neural Language Modeling (NLM), a deep learning approach for learning the probability distribution of a textual corpus. Section 2.6 presents Cross-Encoders, a lightweight Transformer-based model used for document reranking. Finally, Section 2.7 provides a summary of the chapter.

2.1 Program Semantics and Context

Program semantics refer to the meaning or behavior of a computer program, explaining how it processes inputs, performs computations, and produces outputs [73]. They provide a structured way to reason about a program's functionality, correctness, and execution behavior [47]. However, program semantics cannot be fully understood in isolation — they require context. Variables, functions, and control flows must be interpreted by capturing their surrounding program elements. Existing literature [49] suggests that context is not a static property but rather something that emerges dynamically from the interaction and relationships among program elements. To localize software bugs, we need to analyze not only the source code but also its semantics within its context. In particular, we leverage the Large Language Models' ability to understand program semantics in localizing software bugs.

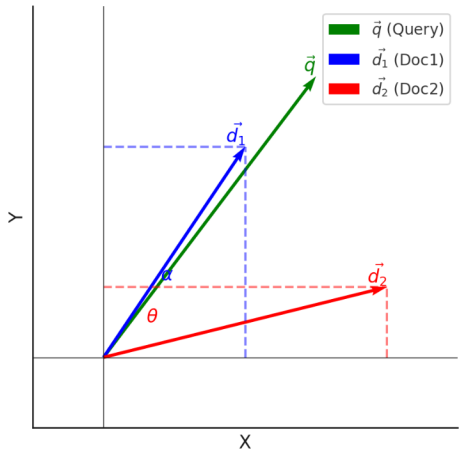


Figure 2.1: Vector Space Model

2.2 Information Retrieval

Information Retrieval (IR) [103] is a technique for retrieving relevant documents from a large collection of documents based on user queries. We discuss several key concepts related to IR as follows.

2.2.1 Vector Space Model (VSM)

The Vector Space Model (VSM) [95] is a foundational concept in Information Retrieval and Natural Language Processing. It represents texts documents as vectors in a continuous vector space (Fig. 2.1), enabling mathematical operations on those vectors. The primary idea is to convert text documents into numerical vectors, where each dimension corresponds to a unique term (usually words) from the entire corpus, and the value in each dimension reflects the importance (e.g., frequency) of that term in the document.

Formally, let a document and all unique terms in the corpus be denoted as D and T respectively. The vector representation of the document D in a $|T|$-dimensional space is given by:

$$D = (w_1, w_2, \ldots, w_{|T|}),$$

where w_i represents the weight of a term in the document. The weights w_i can be calculated using several schemes, such as term frequency (TF), term frequency-inverse document frequency (TF-IDF), or more advanced alternatives like word embeddings.

This vector representation facilitates many tasks, including document similarity computation, clustering, and retrieval, by enabling mathematical operations in the vector space.

2.2.2 Indexing

The first step of information retrieval is to construct an index of the documents from a collection (a.k.a., corpus). It involves parsing the documents, extracting important terms, and storing them in a data structure that allows for efficient retrieval. The most widely used tools for indexing text documents are Lucene [57] and Elasticsearch [6]. They create an inverted index against a corpus [103], which maps terms to their documents and thus optimizes search performance.

To build an efficient index, each document is preprocessed by splitting its content into individual tokens and removing stop words or punctuation marks. Stop words add only little value to the semantics of a document [103, 163]. Stemming or lemmatization is often applied to reduce words to their base forms, ensuring that the variations of a word are treated as identical. All these preprocessing steps can enhance the effectiveness of the index, making searches faster and more accurate. In both of our studies, we used Elasticsearch to index source code, leveraging its ability to handle a large collection of documents.

2.2.3 Document Retrieval

When a user submits a query, the IR-system processes it to understand the information need. This may involve tokenization, stemming, and removal of stop words. The system then matches the processed query against the index to detect relevant documents.

In traditional Vector Space Models (VSM), documents and queries are represented as vectors in a multi-dimensional space, where each dimension corresponds to a unique term in the corpus. To determine relevance, the terms in the documents are often weighted using a scheme like TF-IDF [97], which helps capture the importance of each term. The relevance between the query and the document is then determined by computing the similarity (e.g., cosine similarity) between their corresponding vectors. This approach allows the system to rank documents by how similar they are to the

query, with higher scores showing greater relevance.

Modern IR systems like Elasticsearch [6] typically do not use cosine similarity ranking, given their limitations (e.g., bias towards large documents) [103, 132]. Instead, they rely on more advanced scoring models such as BM25 [133]. We discuss several concepts related document retrieval as follows.

- **TF-IDF**: Term Frequency-Inverse Document Frequency (TF-IDF) is a statistical measure to determine the importance of a term in a document. The *Term Frequency (TF)* measures how often a term appears in a document relative to all the appearances of terms in the document. It is calculated as follows-

$$TF(t,d) = \frac{f_{t,d}}{\sum_{t' \in d} f_{t',d}}$$

Here, $f_{t,d}$ represents the count of term t in document d. The denominator, $\sum_{t' \in d} f_{t',d}$, sums up all the occurrences of terms in the document d. A higher TF value indicates that the term appears frequently in the document, making it more significant for the document.

The *Inverse Document Frequency (IDF)* measures how important a term is within the corpus. It is calculated as follows-

$$IDF(t,D) = \log \frac{|D|}{|\{d \in D : t \in d\}|}$$

Where, $|D|$ is the total number of documents, and $|\{d \in D : t \in d\}|$ counts the documents containing term t. If a term appears in many documents, its IDF value is low, indicating it is common and less informative. Conversely, rare terms have higher IDF values, emphasizing their uniqueness and relevance.

The *TF-IDF score* is obtained by multiplying TF and IDF:

$$\text{TF-IDF}(t,d) = \text{TF}(t,d) \times \text{IDF}(t)$$

Thus, the TF-IDF score assigns greater importance to terms that are frequent in a document but rare across the entire corpus, making them reliable representatives of document content.

- **BM25**: BM25 is a probabilistic retrieval model that ranks documents according to their relevance to a query. Unlike cosine-based ranking, it can address the challenge of large documents. The BM25 score is calculated as follows:

$$\text{BM25}(d, q) = \sum_{t \in q} \text{IDF}(t) \cdot \frac{\text{TF}(t, d) \cdot (k_1 + 1)}{\text{TF}(t, d) + k_1 \cdot (1 - b + b \cdot \frac{|d|}{\text{avgdl}})}$$

  where k_1 and b are parameters, $|d|$ is the length of document d, and $avgdl$ is the average document length in the corpus. The parameter k_1 controls term frequency saturation, limiting the impact of repeated terms. The equation $(1 - b + b \cdot \frac{|d|}{\text{avgdl}})$ scales term frequency based on document length $|d|$ relative to $avgdl$, preventing longer documents from being unfairly favored.

In both of our studies, we use Elasticsearch with its default scoring model– BM25 to retrieve potentially buggy source documents against bug reports (a.k.a., queries) with default $k_1 = 1.2$ and $b = 0.75$ values.

2.3 Word Embedding

Embedding [15] is a mathematical representation of discrete objects in a continuous vector space. This representation allows capturing semantic relationships and similarities between the objects. Formally, an embedding can be defined as a function $f : X \rightarrow \mathbb{R}^d$, where X is a set of discrete objects (e.g., words, sentences) and $\mathbb{R}^d$ is a d-dimensional real-valued vector space. The goal of this mapping is to position similar objects close to each other in the embedding space, thereby preserving their semantic relationships.

Word embedding [108] is a distributed representation of words in a vector space, where semantically similar words are positioned close to each other. This representation captures both syntactic and semantic relationships between words, enabling various natural language processing tasks. A word embedding model learns a mapping w_i for each word $w \in V$, where V is the vocabulary. Words with similar meanings will have vectors that are close together in the embedding space. To find how close two words are, the cosine similarity between two word vectors $\mathbf{v}_i$ and $\mathbf{v}_j$ can be computed as:

$$\text{cosine-similarity}(\mathbf{v}_i, \mathbf{v}_j) = \frac{\mathbf{v}_i \cdot \mathbf{v}_j}{\|\mathbf{v}_i\|\|\mathbf{v}_j\|}$$

where $\cdot$ denotes the dot product and $\|\mathbf{v}\|$ represents the Euclidean norm of each vector. This similarity measure indicates the semantic proximity between two words in the embedding space.

Traditional techniques like Word2Vec [107] and GLoVE [119] use large text corpora to learn these embeddings, leveraging the distributional hypothesis [72]. Similarly, more advanced techniques like BERT [45] capture word semantics by leveraging the self-attention mechanism. Such embeddings enhance the performance of various downstream applications in NLP (e.g., information retrieval). In our first study, we employ an algorithm to reformulate queries from bug reports based on CodeT5 [168] embeddings for bug localization.

2.4 Sequence Modeling

Many real-world tasks (e.g., text completion, summarization) require the analysis of sequential data, where the order of information matters. Traditional feed-forward neural networks often fall short in handling sequential data since they lack memory mechanisms. To overcome such a challenge, several sequence modeling techniques have been introduced over the years. In this section, we will discuss several of them in detail.

2.4.1 Recurrent Neural Network

Recurrent Neural Networks (RNNs) [63, 150] are a type of neural network designed to handle sequential data. Unlike traditional feed-forward networks, RNNs can capture patterns over time, making them useful for tasks that require sequence modeling. These networks are based on the idea of a dynamic system that can capture neuron interactions [134, 173] and maintain an internal memory to process sequences effectively [150].

An RNN model processes an input sequence by targeting one element from the sequence at a time and maintains an internal state (or memory) that retains information about the history of the sequence. The model's output at each step is dependent not

only on the current input of the sequence but also on the previous state, enabling the network to exhibit temporal dynamic behavior. The network is a recursive structure, where the output at the current time step is a function of the state at the previous time step and the current input. The technique of unrolling or unfolding an RNN for a finite number of steps reveals how the state signal at any given point incorporates contributions from all preceding inputs and states within the unrolled window.

RNNs are particularly valuable for text-based tasks because language follows a sequential structure, where the meaning of a word often depends on its context [63]. Their ability to maintain memory over time makes them well-suited for natural language processing (NLP) applications such as language modeling, machine translation, and speech-to-text transcription [142].

RNNs provide a way to model sequential data, but they struggle with vanishing and exploding gradients [118]. [118]. These challenges arise during training using "Back Propagation Through Time" (BPTT) [63], which is an adaptation of the back-propagation algorithm for sequential data. When training on long sequences, the gradients of the loss function with respect to the parameters associated with earlier time steps can become either exponentially small (a.k.a., vanishing gradients), hindering the network's ability to learn long-range dependencies, or exponentially large (a.k.a., exploding gradients), leading to unstable training. To address such challenges, subsequent networks (e.g., LSTM) have been proposed.

2.4.2 Long Short-term Memory and Gated Recurrent Unit Models

Long Short-Term Memory (LSTM) [35, 157] networks and Gated Recurrent Units (GRU) [36] are advanced architectures designed to address the limitations of traditional Recurrent Neural Networks (RNNs), particularly the vanishing and exploding gradient problems that hinder the learning of long-term dependencies.

LSTMs [157] address the above challenge by using a more complex structure that includes memory cells and three types of gates: the input gate, the forget gate, and the output gate. These gates work together to control what information is added to memory, what is discarded, and what is passed on to the next layer. This mechanism helps LSTMs retain important information over long periods while filtering out irrelevant data, effectively addressing the problem of vanishing gradients.

GRUs [36] simplify the above process by combining the input and forget gates into a single update gate and introducing a reset gate. This streamlined approach reduces the model's complexity while still allowing it to capture long-term dependencies effectively. As a result, GRUs often train faster than LSTMs and perform comparably in many tasks, making them a more efficient option in certain scenarios.

Despite their advantages, both LSTMs and GRUs are not without challenges. They can be computationally intensive, requiring significant resources for training, especially on large datasets. Additionally, while they are better at handling long-term dependencies than traditional RNNs, they can still struggle with very long sequences or when the relevant information is far back in the input history.

2.4.3 Attention Mechanism

The attention mechanism [23, 158] overcomes key limitations of LSTM and GRU models, particularly in handling long sequences and dependencies. Although LSTMs and GRUs are designed to retain information across multiple time steps, they often struggle with very long-term dependencies, as information can become diluted or lost. Moreover, their sequential nature restricts parallel computation, increasing training time and reducing efficiency. The attention mechanism addresses these issues by enabling the model to focus dynamically on relevant inputs and facilitating parallel processing during attention score calculation. By leveraging query, key, and value vectors, the mechanism computes attention scores that capture dependencies across the entire input sequence. Its comprehensive contextual representation allows the model to maintain and utilize relevant information for the current output, thereby enhancing its ability to process long and complex sequences. The attention mechanism has different variants [23] to address specific challenges in sequence modeling. One key variant is self-attention, which allows a model to weigh the importance of different elements within a single input sequence. This allows it to capture intra-sequence relationships between words or tokens important for contextual understanding. Multi-head attention extends self-attention by using multiple independent attention heads in parallel, enabling the model to capture diverse relationships and features more efficiently. This parallelism improves performance by employing a model's ability to focus on different aspects of an input sequence simultaneously. In contrast, cross-attention computes

attention between two different input sequences, using queries from one sequence and keys and values from another. This method is particularly useful in tasks where interactions between different modalities or sequences (e.g., machine translation) play a critical role. In our studies, we take advantage of the self-attention mechanism to determine the contextual relationship between bug reports and source documents.

2.4.4 Transformers

The Transformer model, introduced by Vaswani et al. [158] in 2017, represents a significant advancement in deep learning, particularly for natural language processing (NLP) tasks. Unlike traditional sequential models (e.g., RNN, LSTM, GRU), the Transformer employs a self-attention mechanism that processes all elements of an input sequence simultaneously. Self-attention computes the relevance of each token in the sequence to every other token, enabling the model to dynamically focus on the most important parts of the input for a given context. This mechanism enhances the model's ability to capture long-range dependencies and understand contextual relationships within the data. The Transformer architecture consists of an encoder-decoder structure, where the encoder transforms the input sequence into a continuous representation, and the decoder generates the output sequence based on this representation. Each encoder and decoder layer incorporates multi-head attention— a more advanced form of self-attention—and feed-forward networks, facilitating rich interactions between different parts of the input. Positional encodings further allow the model to retain information about the order of tokens in the sequence, which is critical for understanding context. Our work employs Transformer models, and leverage their strengths and contextual understanding.

2.5 Natural Language Modeling

Language modeling (LM) [105], a cornerstone of natural language processing (NLP), has evolved significantly from its early stage. Initially, statistical language models (SLMs) focused on predicting the next word in a sequence using fixed context lengths and probabilistic techniques (e.g., Hidden Markov Model) [105]. While effective for basic tasks (e.g., Part-of-Speech tagging), these models faced challenges due to high-dimensional data, limiting their ability to model complex patterns in language. The

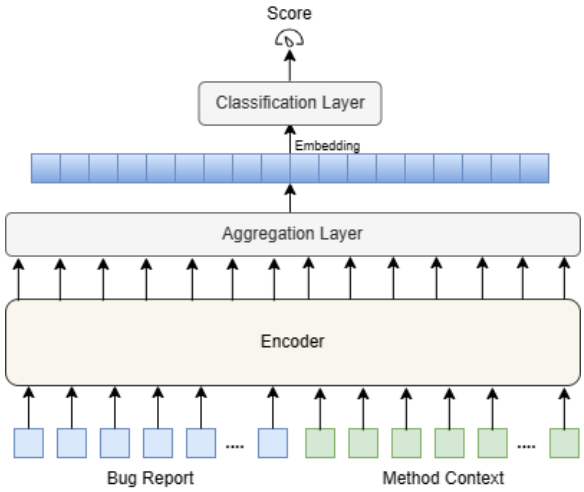


Figure 2.2: Cross-Encoder's Architecture

advent of neural language models (NLMs) [19] addressed these limitations by leveraging deep learning and distributed word representations, enabling machines to capture richer, more nuanced linguistic relationships.

The NLP field advanced further with the introduction of pre-trained language models like BERT [45] and GPT [124], which learn contextual representations from vast datasets. The NLP field advanced further with the introduction of pre-trained language models like BERT [45] and GPT [124], which learn contextual representations from vast datasets. As models grew in size and complexity, they started to exhibit special abilities (e.g., in-context learning) that are not observed in small scale models [184]. These abilities not only enhance traditional tasks (e.g., text classification and translation) but also in more complex reasoning and problem-solving scenarios [184]. They also allow models to tackle challenges across diverse domains, such as programming assistance and data analysis. Both of our works leverage the capability of language models (e.g., CodeBERT, Mistral) to improve bug localization.

2.6 Cross-Encoders

The cross-encoder [75] is a transformer-based approach used for reranking documents by enabling deep interactions between the input context and the candidate. This approach processes the context and the candidate jointly within a single transformer model. The inputs are concatenated with a special separator token (e.g., `[SEP]`)

and passed through the transformer, allowing self-attention to capture fine-grained dependencies between them (Fig. 2.2).

Given an input context `ctxt` and a candidate `cand`, the transformer T produces a joint representation, typically extracted from the first token of the output sequence:

$$y_{\text{ctxt,cand}} = h_1 = \text{first}(T(\text{ctxt}, \text{cand}))$$

During encoding, the candidate label can focus on specific features within the input context, enabling the model to identify the most relevant input features for each candidate. This property makes the cross-encoder particularly effective for reranking tasks where fine-grained relevance is warranted.

Prior studies [44, 117] have demonstrated the cross-encoder's advantages in refining retrieved results by analyzing relationships between queries and documents. We used a cross-encoder in our first study, IQLoc, to rerank source documents based on program semantic relevance between source code and bug reports.

2.7 Summary

In this chapter, we introduced key terminologies and background concepts essential for following the remainder of this report. We began with an overview of program semantics. Then we discuss Information Retrieval (IR) in the context of textual retrieval, followed by a discussion on word embeddings, which numerically represent textual data. We then explored RNNs, LSTMs, GRUs, and Transformers, the widely used architectures for processing sequential data. Next, we traced the evolution of Neural Language Modeling, highlighting its progression toward modern Large Language Models (LLMs). Finally, we introduced the cross-encoder, a document reranking method based on Transformer architecture.

Chapter 3

Improving IR-based Bug Localization with Semantics-Driven Query Reduction

Software practitioners consider bug resolution as one of the major challenges, consuming up to 50% of development time and accounting for 40% of maintenance costs. In this chapter, we introduce a novel bug localization technique – IQLoc – that capitalizes on the strengths of Information Retrieval and LLM-based program understanding to localize software bugs. Previous approaches rely primarily on statistical chances rather than a comprehensive understanding of the program's semantics, limiting their effectiveness in bug localization. In this chapter, we address this gap by introducing an approach that leverages program semantics to improve bug localization.

The rest of this chapter is organized as follows: Section 3.1 introduces IQLoc and highlights the novelty of our contribution. Section 3.2 presents a motivating example to illustrate the usefulness of our approach. Section 3.3 describes our proposed approach for localizing bugs. Section 3.4 discusses our experimental design including dataset construction, evaluation metrics, and results. Section 3.5 discusses related studies in the field of bug localization. Section 3.6 identifies potential threats to the validity of our work. Finally, Section 3.7 summarizes this study.

3.1 Introduction

Software maintenance claimed $2.4 trillion in 2022 from the US economy, a ≈84% increase over the previous two years [4]. A significant factor contributing to this increase in costs is software bug, which prevents a software system from working correctly [77]. These bugs not only hinder the functionality of software but also can lead to severe system failures (e.g., AT&T Mobility outage) [8, 52, 120, 140]. Consequently, software developers dedicate 35-50% of their time tackling these bugs [25, 115]. Zou et al. [187] recently conduct a practitioner survey where they select ten different tasks related to software bugs and collect responses from 327 software practitioners (e.g., developers,

project managers, testers) affiliated with leading IT companies (e.g., Google, Meta, Microsoft, Amazon). According to their survey, 82.4% of the practitioners consider bug localization as the most important or an important task. The task involves identifying the specific locations in the software code responsible for a software bug or failure.

During bug localization, developers often rely on bug reports to find the locations of problematic code. However, understanding bugs from their description in the reports and locating them in the software code is challenging and time-consuming, even for seasoned developers [127]. Bug reports contain natural language text and program artifacts such as stack traces, program elements, and code diffs [129]. The presence of these diverse contents in bug reports and the inherent ambiguities in natural language text [60] can significantly complicate the process of bug localization. To address these challenges, researchers have been actively pursuing automated solutions for bug localization over the last 50 years [84, 92, 94, 114, 121, 136, 163, 164].

Over the last few decades, many methods have been employed to automatically localize software bugs. One frequently used method is Information Retrieval (IR). Existing IR-based approaches [94, 114, 121, 163] often use pre-processed texts from bug reports as search queries and attempt to match them with source code, overlooking the context or semantics of the code. This can lead to spurious matching between bug reports and source code, given the variability of natural language text describing the software bug [31, 60]. Several existing approaches incorporate code change history, version control history, or even code authoring history to improve bug localization [164, 181]. Although additional contexts of a software bug are captured, these approaches rely primarily on statistical chances rather than a comprehensive understanding of the program semantics, limiting their effectiveness in bug localization. A recent work [96] shows that these techniques perform comparably despite incorporating additional contextual information. Furthermore, another recent work [127] suggests that the performance of existing IR-based techniques might be significantly affected by their selected queries from the bug reports.

Recently, several approaches have focused on making appropriate search queries based on bug reports to improve bug localization. Rahman and Roy [126] leverage both co-occurrences and syntactic dependencies among words from a bug report to

capture meaningful search queries. In a recent work [129], the authors also show the use of static and hierarchical dependencies among program elements to construct queries. Other techniques [30, 143] make queries from a bug report through keyword expansion or noise removal. Although effective in making queries, these techniques could be limited by the vocabularies of bug reports, which are of varying quality [127]. This limitation restricts their capacity to comprehend contextual intricacies and locate defective source code documents. In contrast, deep learning-based techniques show promise in understanding the conceptual relationship between code and text [26,177,185]. Being trained on corpora beyond bug reports, these techniques can draw inferences from a larger knowledge base. However, they fail to leverage the benefits that the existing methods (e.g., IR-based) have to offer during bug localization. Moreover, they require extensive data and computational power, hindering their large-scale, sustainable adoption.

To design a comprehensive solution capitalizing on the strengths of both approaches above, we propose *IQLoc*, a hybrid approach combining Information Retrieval (IR) with transformer-based models to localize software bugs. Textual relevance alone may broaden a search space, potentially delivering unrelated results. Our approach narrows down the search space by leveraging program semantics understanding of the pre-trained language models during bug localization. First, IQLoc retrieves the top K source documents textually relevant to a bug report (a.k.a., query) using an IR-based approach (e.g., the BM25 algorithm [133]). Second, it trains a transformer-based, cross-encoder model to assess their relevance (based on their program semantics) to the bug report and narrows down the search space. Finally, IQLoc reduces the search query leveraging the above documents, reranks the documents using the query, and returns the suspicious source code documents.

We selected an existing benchmark dataset – Bench4BL [96] – for our experiments and extended it with recent bug reports (i.e., submitted until September 2024). To refine the dataset, we selected the bug reports that include version information and have corresponding relevant documents in their respective GitHub repositories. We further expanded it with ≈30% more recent bug reports, resulting in a final set of 7,483 bug reports. To evaluate our proposed technique, we employed three appropriate and widely used metrics: Mean Average Precision (MAP), Mean Reciprocal

Rank (MRR), and HIT@K. We compare our technique with four appropriate baselines from literature – BLUiR [136], Blizzard [129], DNNLoc [92] and RLocator [29]. Across various measures, IQLoc consistently outperformed these techniques, with improvements of up to 58.52% and 60.59% in MAP, 61.49% and 64.58% in MRR, and 69.88% and 100.90% in HIT@K for the test bug reports with random and time-wise splits, respectively. Additionally, IQLoc improves bug localization over baselines in terms of MAP and MRR, achieving 91.67% and 86.49% for bug reports with stack traces, 72.73% and 66.67% for those containing program artifacts, and 65.38% and 64.29% for natural language-only bug reports.

Thus, this research makes the following contributions:

- A novel hybrid bug localization technique, *IQLoc*, that capitalizes on the strengths of both traditional (e.g., query reformulation, IR) and deep learning-based approaches (e.g., transformer models) and leverages textual relevance, program semantics relevance, and language models in bug localization.
- A refined and extended Bench4BL dataset incorporating recent bug reports, resulting in ≈7.5K bug reports.
- A comprehensive evaluation of IQLoc using three popular metrics, two different splits of the dataset, and comparison with four baseline techniques [29, 92, 129, 136] from the literature.
- A replication bundle[1] comprising a prototype, a carefully curated experimental dataset, configuration details, and trained and pre-trained models for third-party use and replication.

3.2 Motivational Example

In this section, we demonstrate the effectiveness of our approach –IQLoc– in bug localization using a motivating example. We present an example where our approach outperforms the baseline methods. Table 3.1 shows an example bug report from *Spring Workflow (SFW)* and Fig. 3.1 shows the associated buggy code. When the

[1]https://github.com/asifsamir/IQLoc

```
@@ -118,6 +118,10 @@ public class DefaultFlowExecutionRepository
public void putFlowExecution(FlowExecution flowExecution) {
        assertKeySet(flowExecution);
+       if (maxSnapshots == 0) {
+               return;
+       }
+
        if (logger.isDebugEnabled()) {
                logger.debug("Putting flow execution '" + flowExecution + "' into repository");
        }
        FlowExecutionKey key = flowExecution.getKey();
        Conversation conversation = getConversation(key);
        FlowExecutionSnapshotGroup snapshotGroup = getSnapshotGroup(conversation);
        FlowExecutionSnapshot snapshot = snapshot(flowExecution);
        if (logger.isDebugEnabled()) {
                logger.debug("Adding snapshot to group with id " + getSnapshotId(key));
        }
        snapshotGroup.addSnapshot(getSnapshotId(key), snapshot);
        putConversationScope(flowExecution, conversation);
}
```

Figure 3.1: Buggy Code and Method Context

Table 3.1: An Example of Bug Report and Search Queries

Bug ID # 1416 (Spring Webflow)		Rank
Bug Title	Turn off snapshot creation when max-snapshots=0	72
Bug Description	Creating snapshots involves serializing and compressing the flow execution object, which can lead to issues with some PersistenceContext providers. Having the ability to turn off snapshot creation will provide an option when such issues occur at the cost of losing back button support.	57
Baseline query	Bug Title + Bug Description	53
IQLoc	persistencecontext snapshot involves creation losing support flow object turn execution ability providers serializing	1

pre-processed version of the bug report is executed as a query by an IR-based search engine (e.g., Elasticsearch), the first buggy document is found at the 53rd position, which is not an ideal result. As shown in Table 3.1, the pre-processed version of the title or description fields also does not make a good query.

The example bug report discusses an issue involving snapshot creation, which could affect several classes including `PersistentContext`. Interestingly, the root cause of the bug involves the serialization/compression of the `FlowExecution` object. However, traditional techniques such as BLUiR and Blizzard consistently select `PersistenceContext` as a query keyword since it is found in the bug report. This leads to poor retrieval performance and they retrieve their first buggy documents at the 13th and 18th positions, respectively. In contrast, the seminal work on deep

learning-based bug localization –DNNLoc– considers multiple features (e.g., textual relevance, change history, collaborative filtering), but performs poorly by retrieving the buggy document at the 28th position.

On the other hand, IQLoc attempts to understand the context of a reported bug from the source code. In particular, its cross-encoder module learns to connect bug reports to their corresponding buggy code through transformer-based, contextual learning. The cross-encoder module of our technique was able to rank the buggy code at the 11th position. Our module achieves that by capturing the context of the bug from the bug report and making a connection with the intent of the buggy code (e.g., snapshot creation from `FlowExecution`). More interestingly, upon reformulating the query (a.k.a., preprocessed bug report) leveraging the IR-based and cross-encoder-based components, our technique delivers the buggy source document (e.g., Fig 3.1) at the 1st position, which is ideal and highly promising.

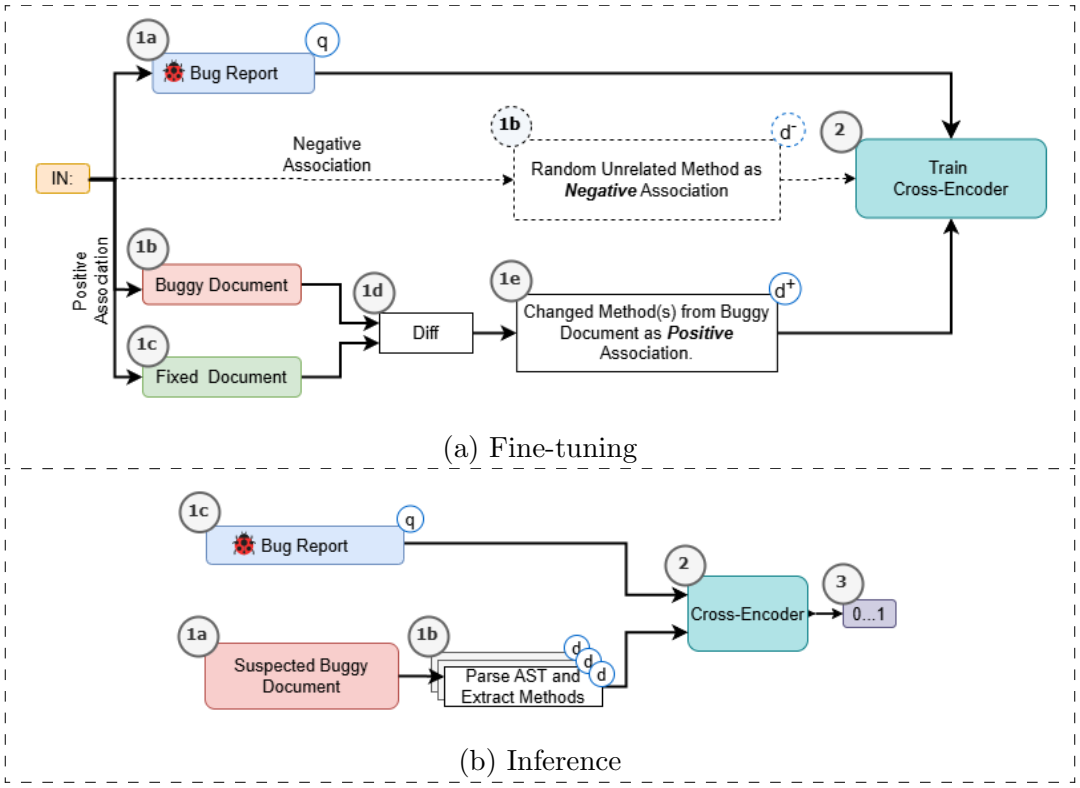


Figure 3.2: Fine-tuning and Prediction of Cross-Encoder Model

3.3 Methodology

In this section, we present the methodology of our proposed technique –IQLoc– for software bug localization. First, we describe the fine-tuning process of the Cross-Encoder model, shown in Fig. 3.2, which assesses the relevance between bug reports and source documents. Then, we explain how the fine-tuned model is integrated into the overall workflow, as illustrated in Fig. 3.3.

3.3.1 Fine-tune Cross-Encoder Model

In IQLoc, we employ a cross-encoder model to determine relevance between a bug report and a code segment (Step 3, Fig. 3.3). First, we fine-tune the base model of CodeBERT, a transformer-based encoder model [54], using bug-fix changes and enable it to differentiate between buggy and non-buggy instances during inference (Fig. 3.2a). In particular, we append feed-forward network to the pre-trained CodeBERT model to achieve classification. Due to its pre-training on a large amount of natural language text and source code, the model is well-suited for our task involving bug reports and source code.

To fine-tune the model, we first parse each source code document from the training corpus and extract its buggy methods by leveraging the bug-fix diffs from corresponding version control history (Step 1c, Fig. 3.2a). We detailed this method extraction process in Section 3.4.1. We also establish connection between bug reports and corresponding buggy code using appropriate heuristics [96]. Then we treat bug reports (Step 1a) and buggy code pairs (Step 1e) as *positive* samples in our training dataset. These pairs are fed into the cross-encoder model for training with the goal of establishing *positive* contextual associations between buggy methods and their respective bug reports (Step 2, Fig. 3.2a).

To train the model to differentiate between buggy and non-buggy samples, we also add negative pairs where each pair consists of a bug report (Step 1a, Fig. 3.2a) and a randomly selected method body (Step 1b-dashed) to distinguish buggy from non-buggy code by leveraging program semantics. The capability stems from CodeBERT's pre-training on a large bimodal corpus, which equips it with rich semantic representations of code structure and intent [54], as evidenced by its performance on

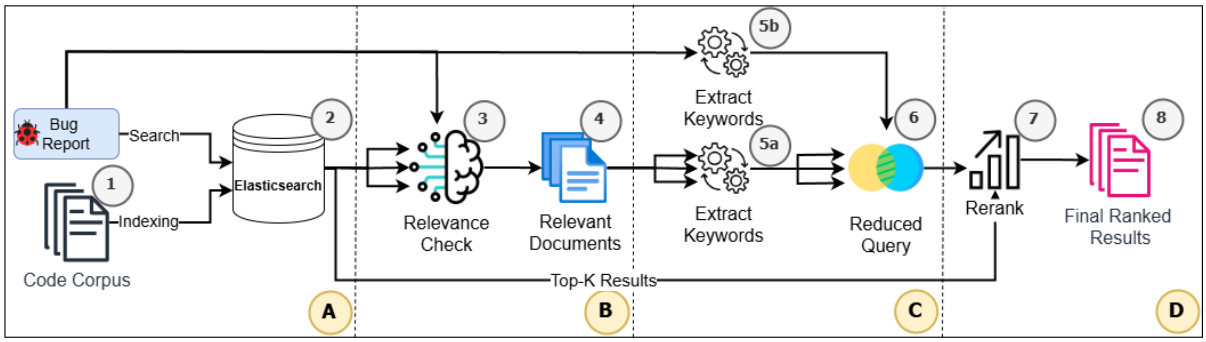


Figure 3.3: Schematic Diagram of *IQLoc*: *(A)* Indexing & Retrieval of Documents, *(B)* Check Relevancy, *(C)* Query Reformulation, *(D)* Bug-Localization

downstream tasks (e.g., code summarization, clone detection [81, 101, 146]). Fine-tuning further adapts these representations by reinforcing links between real-world bug reports and their corresponding buggy code, enabling the model to capture subtle semantic cues associated with buggy behavior.

3.3.2 Corpus Indexing

As we aim to localize software bugs using an Information Retrieval (IR)-based solution, it is essential to index all the source documents of a code base (a.k.a., corpus). In our approach, we chose Elasticsearch [6] due to its efficient handling of diverse data types, seamless integration with research tools, and robustness in document indexing. Further details on corpus indexing can be found in Section 2. We index all source code (Step 1, Fig. 3.3) from the 1,578 buggy versions of 42 subject-systems from the Bench4BL dataset.

3.3.3 Retrieval of Potentially Buggy Source Documents

We use Elasticsearch to retrieve potentially buggy source documents by executing queries selected from bug reports (Step 2, Fig. 3.3). Elasticsearch employs its default Standard Analyzer to preprocess queries and retrieve the top-K results (e.g., K = 100) from the corpus. During this retrieval process, we use the default scoring function of Elasticsearch, Okapi BM25 [133], to assess the relevance of the search results. When retrieving suspicious buggy documents, we specify the particular project and version of the bug report where the bug occurred. Unlike other bug localization techniques [84, 129, 136], which consider only the most recent version of a project and

may retrieve unrelated documents, our approach ensures that we localize bugs within the correct historical context. This allows us to better mimic real-world scenarios by retrieving documents from the appropriate project and version. At the end of this step, we get a list of potential buggy source documents based on their textual relevance to the bug report.

3.3.4 Relevance Estimation using Cross-Encoder

After obtaining results from Elasticsearch, we employ our fine-tuned cross-encoder model to estimate their program semantic relevance to the bug report (Step 3, Fig. 3.3). Given that the cross-encoder model was trained to differentiate between buggy and bug-free code (Fig. 3.2), its reasoning against the candidate methods should be useful. While source code vocabulary is important, we thus also analyze code-level contexts leveraging the cross-encoder to support bug localization.

We analyze each retrieved source code document by parsing its Abstract Syntax Tree (AST) and extracting its individual methods (Step 1a-1b, 3.2b). Each method is then paired with the bug report (Step 1c, 3.2b) and passed into the fine-tuned cross-encoder model (Step 2, 3.2b), which generates generate a relevance score. This score, ranging from 0 to 1, estimates the likelihood of a method being buggy. This step refines the retrieved documents, enhancing the relevance between the bug report and source code (Step 4, Fig. 3.3).

3.3.5 Query Reformulation

Although the above steps narrow down our search space, we further refine the relevance estimation between bug reports and source code segments by reformulating our search queries. In the following section, we discuss how we reformulate our queries by leveraging the LLM's reasoning capabilities as follows.

Pre-training of Large Language Models for Software Bugs

We extract salient keywords from a bug report and relevant code segments (Steps 5a, 5b, Fig. 3.3), using a Transformer-based model. By leveraging the self-attention mechanism [158], Transformer models can focus on the most relevant parts of the

Algorithm 1 Extract Keywords

Require: Document Doc, number of top keywords N, trade-off parameter λ ($0 \leq \lambda \leq 1$)

procedure ExtractKeywords(Doc, N, λ)
  $D \leftarrow \{\}$ ▷ Initialize an empty dictionary
  $T \leftarrow$ Preprocess(Doc) ▷ Get tokens
  for each $t \in T$ **do**
    $E \leftarrow \text{Embed}(t)$
    $D[t] \leftarrow E$ ▷ Add embedding for a token
  end for
  $B \leftarrow \text{Embed}(Doc)$ ▷ Compute document embedding
  $K \leftarrow \emptyset$ ▷ Initialize selected keyword set
  $C \leftarrow D.keys()$ ▷ Candidate tokens
  while $|K| < N$ and $C \neq \emptyset$ **do**
    $Scores \leftarrow \{\}$
    for each $t \in C \setminus K$ **do**
      $s_d \leftarrow$ CosineSimilarity($D[t], B$) ▷ Similarity with document
      if $K \neq \emptyset$ **then**
        $s_k \leftarrow \max_{k \in K}$ CosineSimilarity($D[t], D[k]$)
      else
        $s_k \leftarrow 0$ ▷ No diversity penalty if no keywords selected
      end if
      $MMR \leftarrow \lambda \cdot s_d - (1 - \lambda) \cdot s_k$ ▷ Compute MMR score
      $Scores[t] \leftarrow MMR$
    end for
    $t^* \leftarrow \arg\max_{t \in C \setminus K} Scores[t]$ ▷ Select token with highest MMR score
    $K \leftarrow K \cup \{t^*\}$
    $C \leftarrow C \setminus \{t^*\}$ ▷ Remove selected token from candidates
  end while
  return K ▷ Return top-N keywords
end procedure

text, understand the context, and represent the text comprehensively using high-dimensional vectors. Their representations can be further adapted to different application domains through pre-training using domain specific unstructured data [149].

To help capture salient keywords from bug reports and source code, we pre-trained a transformer model, CodeT5, using bug reports with masking. The process of collecting the pre-training dataset is explained in Section 3.4.1. We chose the CodeT5 model for embedding generation since it has been trained on both natural language texts and source code, which is ideal for bug reports containing various elements including text, code snippets, and program elements. Masked pre-training [45] involves randomly masking a portion of the input tokens during training, encouraging the model to learn contextual representations by predicting the masked tokens. Bug reports contain valuable information about software issues, including descriptions of bugs, code snippets, and stack traces, etc. By incorporating the masking technique to learn from bug reports, we aimed to enhance CodeT5's general understanding of the domain-specific language, including its software engineering jargon and bug-related linguistic nuances.

During the pre-training phase, we implemented a masking mechanism where 15% of the tokens in each input sequence (i.e., bug reports) were randomly selected and substituted with a special `<extra_id_XX>` token. This approach aligns with the original T5 [125] model's convention, where XX serves as a unique identifier assigned to each masked token. The identifier follows a continuous numbering system, ensuring that each masked token in the input sequence receives a distinct number in sequential order (e.g., `<extra_id_0>, <extra_id_1>, <extra_id_2>`, and so forth). These input sequences with the masked tokens are then fed into the model to predict the original tokens based on contextual cues. By training the model to reconstruct the original tokens from the masked input, it acquires the ability to capture the general contextual understanding of bugs from bug reports.

Keywords from Bug Reports:

Keyword extraction involves selecting a subset of words or phrases that capture the main theme of a text document, which can support various subsequent tasks (e.g., Information Retrieval [21]). For extracting keywords from bug reports, we employ

EmbedRank [20] due to its simplicity and compatibility with Transformer models using the KeyBERT [66] library. The Algorithm 1 outlines the keyword extraction process. We begin by pre-processing a bug report using standard text pre-processing techniques, such as removing stop words and punctuation marks. Next, we employ our pre-trained CodeT5 model to embed each token of the bug report. Similarly, we apply the technique to embed the entire bug report. Then we calculate cosine similarity [70] to measure the semantic proximity of each token to the bug report. Based on these scores, we select the top-N most similar keywords (Step 5b, Fig. 3.3) for the subsequent steps. To maximize diversity in the chosen keywords, we employ the Maximal Marginal Relevance (MMR) [27] algorithm and set the MMR parameter $\lambda = 0.5$, following the authors' recommendation. Since the number of keywords affects retrieval, we also determined the optimal value of N through a controlled experiment, as discussed in RQ_2.

Keywords from Code Segments:

We follow a similar approach to extract keywords from source code segments (Step 5a, Fig. 3.3). However, instead of processing entire source documents, we consider only the relevant code segments identified by the cross-encoder with a confidence score above a predefined threshold (e.g., 0.5). If multiple code segments within a document are deemed relevant, we concatenate them into a single code block before applying Algorithm 1 for keyword extraction. We detail the selection of this threshold through a controlled experiment in RQ_3.

Reformulating the Query:

After extracting keywords from the bug report and relevant code segments, we leverage them to reformulate our queries and to enhance the retrieval (Step 6, Fig. 3.3). We first measure the semantic similarity between the bug report keywords and the code keywords using cosine similarity, leveraging embeddings from the CodeT5 model. Our goal was to detect the top relevant documents and capture their overlapping tokens to enhance the keywords from the bug report. This reformulated query based on such an enhancement is then used in the subsequent steps.

3.3.6 Bug Localization

Once we have the reformulated query, we rerank the top-K results (e.g., 100) initially retrieved from the Elasticsearch (Step 7, Fig. 3.3). Similar to the initial retrieval, we use the BM25 algorithm to rerank with the query constructed in the previous step. As a result, it provides a more refined set of final results (Step 8 of Fig. 3.3) for bug localization. Our goal was to place the buggy documents at the top positions within the ranked list through the reranking. Then the developers will encounter the buggy documents earlier and spend less time analyzing the false-positives.

3.4 Experiment

We evaluate our approach using the Bench4BL benchmark dataset and three appropriate performance metrics. We conduct our experiments on a cluster computing system equipped with an NVIDIA GPU with 16 GB of vRAM and compare our technique, IQLoc, against four baseline techniques from the literature. Using our experiments, we attempt to answer four research questions as follows:

- $\boldsymbol{RQ}_1$: How does IQLoc perform in bug localization in terms of evaluation metrics?

- $\boldsymbol{RQ}_2$: *(a)* How does query length affect IQLoc's performance in bug localization? *(b)* What is the rationale for choosing the CodeT5 model for reformulating query? *(c)* Does the use of pre-trained models enhance the performance of these queries?

- $\boldsymbol{RQ}_3$: How does the cross-encoder model perform in identifying relevant buggy source code based on program semantics?

- $\boldsymbol{RQ}_4$: *(a)* Can IQLoc outperform the existing baseline techniques in bug localization? *(b)* How does it perform in localizing different types of bug reports compared to the baseline techniques?

Table 3.2: Bench4BL Dataset Summary

Project	Subject-systems	Bug reports
Apache	13	4,503
Jbosss	8	1,505
Spring	25	3,451
Old Subject	5	558
Total	51	10,017

3.4.1 Dataset Construction

We use Bench4BL [96], a benchmark dataset, for our experiments. Table 3.2 summarizes the Bench4BL dataset. Since the dataset is relatively old, we refine and extend it with more recent bug reports. Table 3.3 shows our refined and expanded dataset. In total, we spent over 80 hours refining and expanding the dataset. The following sections detail our refinement process, the collection of new bug reports, and the dataset's split for training and testing in bug localization.

Refinement of Bench4BL Dataset:

The Bench4BL [96] dataset contains curated bug reports from various Java-based community projects, including Apache, JBoss, and Spring. It captures 10,017 bug reports from 51 subject systems along with their buggy/fixed versions of code and other relevant metadata. Table 3.2 provides a summary of the benchmark dataset.

From our initial analysis of the benchmark dataset, we noticed that many projects or bug reports lacked appropriate versioning information, making them unsuitable for our experiment. We thus dropped the bug reports without any version details and ensured that the remaining reports could be traced back to both buggy and bug-free versions of the code. During this process, we encountered several challenges, including cases where either the buggy or fixed version was missing in the code repository. We discarded the bug reports with such discrepancies, ultimately retaining a dataset of 5,753 bug reports from the three large community projects (i.e., Apache, JBoss, and Spring).

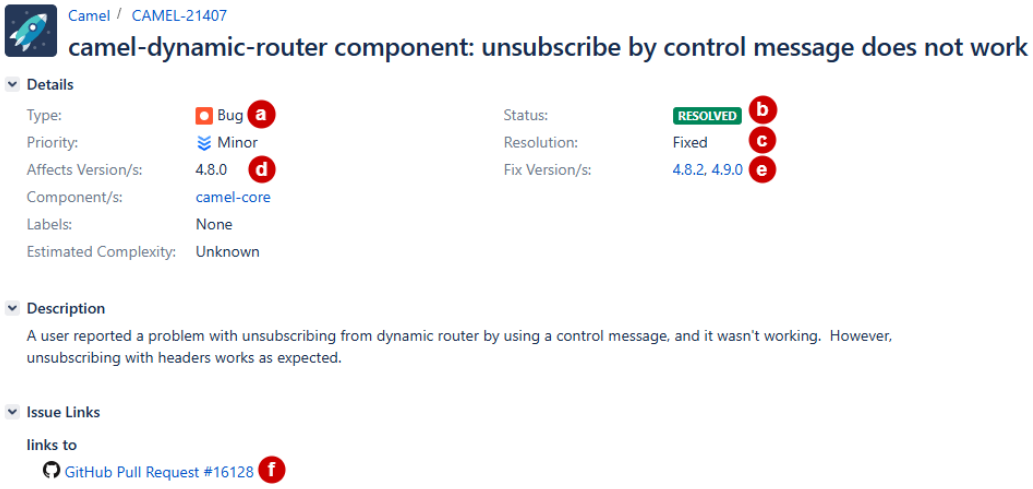


Figure 3.4: An Example Bug Report from JIRA

Expanding the Bench4BL Dataset:

After refining the original Bench4BL dataset, we expanded it by adding more recent bug reports for our experiments. In particular, we collected bug reports that were submitted by September 2024. Our expansion is limited to bug reports from the same projects and subject systems as the original dataset. These systems are managed by GitHub (e.g., Spring) and JIRA (e.g., JBoss, Apache). To collect new bug reports, we employed two distinct approaches tailored to these issue-tracking systems.

To collect bug reports from JIRA-based issue tracking systems, we used the JIRA API [40]. We also filtered issues that were of type ⓐ 'Bug,' had a status of ⓑ'Resolved,' a resolution of ⓒ 'Fixed,' and included both ⓓ a buggy version and ⓔ a fixed version (Fig. 3.4). Additionally, we considered bug reports containing explicit ⓕ Git pull request links to ensure a proper mapping between the buggy and fixed versions, reducing false positives. Once we collected the bug reports from JIRA, we used the GitHub API to extract the corresponding buggy and fixed files from the linked pull requests. Since a pull request can contain multiple commits, we considered only accepted and verified commits. When multiple commits were associated with a pull request, we included all files across these commits as a part of the fix for that bug report. This approach accounts for bug fixes that evolve over time, often involving multiple developers addressing different aspects of the same bug. Additionally, it captures cases where bug reports were closed and later reopened for further

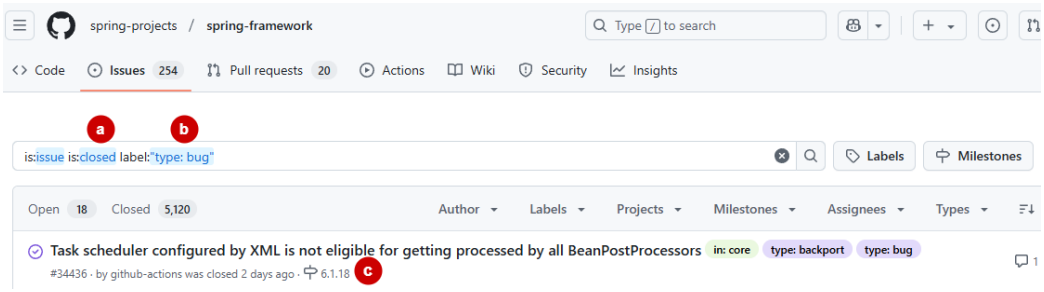


Figure 3.5: GitHub Issue Selection

fixes. By including all relevant files, we ensure a comprehensive representation of the bug-fix process. Finally, we ensured that the versions specified in JIRA issues (e.g., semantic version, 4.8.2) matched the version tags in GitHub (e.g., camel-4.8.2) using regex-based validation, as the version strings in JIRA could differ from those in GitHub.

Table 3.3: Refined and Expanded Dataset

Project	Subject-systems	Major versions	Bug reports
Apache	11	428	3,145
Jbosss	6	303	1,416
Spring	25	847	2,922
Total	42	1,578	7,483

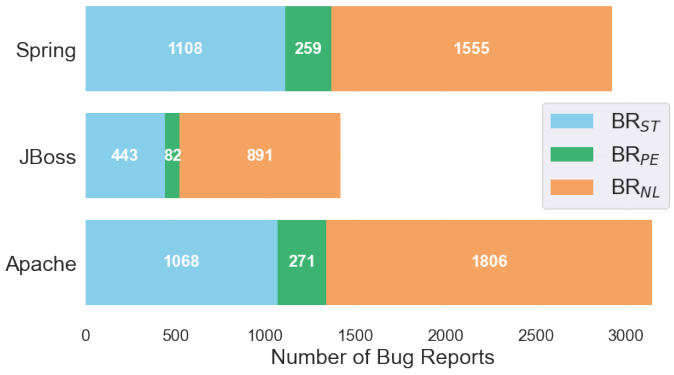


Figure 3.6: Classification of Bug Reports

On the other hand, when collecting bug reports from GitHub, we considered issues that are ⓐ 'closed' and labeled as ⓑ 'type: bug' (Fig. 3.5). To determine the version in which a bug was fixed and reduce false positives, we collected bug reports that had explicit ⓒ milestones attached, indicating the version(s) in which the bug was

Table 3.4: Train, Validation and Test-sets

Split Type	Training	Validation	Test	Total
Random Split	5,238	748	1,497	7,483
Time-wise Split	5,236	746	1,501	7,483

resolved. Since GitHub issue reports do not explicitly define the buggy version, we identified the immediate previous version tag associated with the first bug-fix commit associated with the issue in the branch tree. While this may not always correspond to the exact commit or version where the bug was introduced, it allows us to determine the latest version in which the bug exists. After that, for resolving the buggy files, we applied the same approach as used for JIRA bug reports.

Once we collected the bug reports and resolved the buggy files, we dropped the bug reports related to configuration bugs (i.e., IML, XML). Following this procedure, we complemented the original Bench4BL dataset with 1,730 recent bug reports. In total we curated 7,485 bug reports from 42 subject-systems across 1,578 buggy/fixed versions.

To gain further insight, we classify bug reports in our experimental dataset based on their content, as was done by existing literature [129]:

- BR_{ST} (Bug Reports with Stack Traces): Includes stack traces along with text or program elements. Queries from these reports are generally noisy.

- BR_{PE} (Bug Reports with Program Elements): Contains program elements (e.g., method invocations, package names) but no stack traces. Queries from these reports are considered rich.

- BR_{NL} (Bug Reports with Natural Language Only): Lacks both program elements and stack traces. Queries from these reports are generally less informative.

To classify them, we use regular expressions adapted from the work of Rahman et al. [129]. Fig. 3.6 shows the classification of our curated dataset.

(a) Random Split (b) Time-wise Split

Figure 3.7: Distribution of Bug Reports in Different Dataset-Splits

Train and Test Set:

For our experiments, we split the dataset into training, validation, and test sets using a 70:10:20 ratio, following the standard machine learning practice for data partitioning [64, 79]. In our dataset splitting, we adopted two strategies, as was done by earlier studies [111, 145]. The first approach utilized random selection with shuffling, where the dataset was randomly divided into training and test sets. To ensure an unbiased random split, we conducted this process five times, generating five randomly shuffled datasets to facilitate separate experiments. This process delivered 5,238 bug reports for training, 1,497 for testing and 748 for validation on random trials in each of the five sets. The second strategy was a time-wise split. This type of splitting divides data into training and test sets based on their chronological order, simulating real-world scenarios where models are trained on past data and evaluated on future instances. For each subject system, we sorted the bug reports by their submission dates, split them individually into train, validation and test sets, and then combined them with similar splits from other systems. This process resulted in 5,236 bug reports for training, 1,501 for testing and 746 bug reports for validation. Table 3.4 summarizes our curated dataset for the experiments. Also, Fig. 3.7 shows how classified bug reports (i.e., BR_{ST}, BR_{PE}, BR_{NL}) are distributed across our training, validation and test dataset. Note, for random-split, the distribution shows the average of five random-splits.

Table 3.5: Cross-Encoder Dataset (Random Split)

Dataset Type	Bug Reports	q, d^+ Pairs (Avg.)	q, d^- Pairs (Avg.)	Total (Avg.)
Training	5,238	12,694	50,776	63,470
Test	1,497	3,715	14,860	18,575
Validation	748	1,829	7,316	9,145

Table 3.6: Cross-Encoder Dataset (Time-wise Split)

Dataset Type	Bug Reports	q, d^+ Pairs	q, d^- Pairs	Total
Training	5,236	12,727	50,908	63,635
Test	1,501	3,820	15,280	19,100
Validation	746	1,691	6,764	8,455

Expanding Training Set for Cross-Encoder:

To fine-tune our cross-encoder model, we extract both the buggy and bug-free versions of the code associated with each bug report from the GitHub repository of the respective subject system. Using a Diff tool [5], we compare between buggy and bug-free code and identify the method bodies in the buggy document that were changed to correct the bug. These altered method bodies are selected as positive instances (q, d^+) for our model training against each bug report, with a label of 1 assigned to them. This process is illustrated in Fig. 3.2.

In our dataset, we observed that each bug resulted in changes to a minimum of 1 and a maximum of 7 source code documents. To construct the training set for our cross-encoder, which determines the semantic relevance between a bug report and a document, we process all documents individually and generate multiple instances of positive associations. For example, if a bug report requires changes to three documents, we create three training instances, linking the altered method bodies to the bug report as positive instances for training. Our approach ensures that each training instance has a limited number of tokens, ensuring compatibility with the transformer model, which has a token limit of 512.

To generate negative samples (q, d^-), we randomly select method bodies associated with different bug reports for a given report at hand. We only consider source code from different subject systems for these negative cases. This process is reiterated four times to yield sufficient training instances for the model, and they are labeled

as 0. The choice of four negative instances against each positive instance is based on a study conducted by Huang et al. [74]. They found no notable distinctions across different numbers of negative samples and also endorsed the 4:1 ratio.

For the validation and test data, we followed the same approach to generate positive and negative samples. Tables 3.5 and Table 3.6 summarize the training, validation, and test sets for the cross-encoder across two types of experiments.

Dataset for Pre-training Model with Bug Reports:

To pre-train an existing baseline language model (e.g., CodeT5) with domain-specific data, we collected thousands of bug reports from GitHub repositories hosting Java-based projects using the GitHub API. Our selection process involved choosing the top 100 repositories based on their star count and thus ensuring that they remained active up to the date of selection. To maintain data integrity, we targeted the issue reports labeled as 'bugs' and collected reports submitted before April 2024. Additionally, we confirmed that none of these repositories were already included in the Bench4BL dataset to prevent any bias. After collecting the bug reports, we meticulously cleaned them to retain only their textual content using Beautiful Soup [1] and excluded repositories with fewer than five bug reports. Finally, all these steps resulted in a dataset of 70,884 bug reports from 74 repositories. Then, these bug reports were used to pre-train the CodeT5 model.

3.4.2 Evaluation Metrics

To evaluate IQLoc in bug localization, we use three widely used metrics- MAP, MRR, and HIT@K. These metrics have been frequently used by the relevant literature on IR-based bug localization [127,129], and thus are highly relevant to our approach. To perform ablation study involving our cross-encoder model, we used four commonly used metrics – Accuracy, Precision, Recall, and F1 score.

Mean Average Precision (MAP)

Precision@K refers to precision for each occurrence of the buggy source document in the ranked list. Average Precision calculates the average precision@K for all buggy

documents against a search query. Therefore, Mean Average Precision (MAP) is computed by averaging the AP values across all queries (Q) from a dataset.

$$P_k = \frac{\textit{No. of Relevant Items in Top-k}}{k}$$

$$AP@K = \frac{1}{|D|} \sum_{k=1}^{K} P_k \times B_k$$

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{Q} AP@K_q$$

Here, P_k calculates the precision for the k^{th} element of the top K items in the ranked list returned by a query. $AP@K$ computes the average precision for a list of K results against a query, utilizing B_k to determine if the k^{th} document is buggy or not. B_k outputs 1 for a match with the ground truth and 0 otherwise. D represents the set of relevant instances that match the ground truth documents against a query. Finally, MAP calculates the mean of the average precision ($AP@K$) across all the individual queries q in the set Q.

Mean Reciprocal Rank (MRR)

Reciprocal Rank (RR) is associated to the rank of the first relevant result retrieved by a technique. It calculates the reciprocal of the rank of the first relevant source document within the ranked list returned by each query.

$$RR_q = \frac{1}{\textit{Rank of First Relevant Item}}$$

$$MRR = \frac{1}{|Q|} \sum_{q=1}^{|Q|} RR_q$$

Here, RR_q represents the Reciprocal Rank for a specific query q. Thus, the Mean Reciprocal Rank (MRR) is calculated as the mean of the Reciprocal Ranks (RR_q) for all individual queries q in the set Q.

HIT@K

HIT@K or Recall@Top-K [136] refers to the proportion of queries for which a technique returns at least one relevant document among the top K retrieved results. The higher the HIT@K values, the better the performance of a bug localization technique.

$$HIT@K = \frac{1}{|\mathcal{Q}|} \sum_{q=1}^{|Q|} \begin{cases} 1, & r_q \in \mathcal{G} \\ 0, & \text{otherwise} \end{cases}$$

Here, Q is the set of all queries and r_q is a binary indicator function that returns 1 if the query q has at least one ground truth item $r_q \in G$ within the top-K results, and 0 otherwise.

Accuracy

Accuracy is a metric that determines the proportion of correctly predicted instances out of the total instances evaluated.

$$\text{Accuracy} = \frac{TP + TN}{TP + FP + TN + FN}$$

Here, TP (True Positive) and TN (True Negative) represent instances that are *correctly predicted* as positive and negative, respectively. Conversely, FP (False Positive) and FN (False Negative) represent instances that are *incorrectly predicted* as positive and negative, while in reality, they are negative and positive instances, respectively. In the context of bug localization, positive instances denote buggy documents, while negative instances denote non-buggy ones.

Precision

Precision measures the proportion of correctly *predicted* positive instances out of all positive *predictions* made by a model.

$$\text{Precision} = \frac{TP}{TP + FP}$$

Recall

Recall, also known as sensitivity, measures the proportion of correctly *predicted* positive instances out of all *actual* positive instances in the dataset.

$$\text{Recall} = \frac{TP}{TP + FN}$$

F1 Score

The F1 Score provides a single measure that balances between precision and recall. It is calculated as the harmonic mean of precision and recall. This balanced measure is particularly valuable in scenarios where false positives and false negatives have different implications, ensuring a robust assessment of a model's performance.

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

Table 3.7: Performance of IQLoc

Split Type	MAP	MRR	HIT@1	HIT@5	HIT@10
Random Split	0.493	0.520	0.423	0.647	0.721
Time-wise Split	0.520	0.553	0.466	0.669	0.735

Table 3.8: Impact of the Selection of Top-K Results from Elasticsearch

Split Type	HIT@1	HIT@5	HIT@10	HIT@50	HIT@100	HIT@200
Random Split	0.389	0.619	0.703	0.843	0.889	0.929
Time-wise Split	0.396	0.618	0.822	0.904	0.937	0.961

3.4.3 Evaluting IQLoc

Answering RQ_1: Performance of IQLoc

Table 3.7 presents the performance of IQLoc in terms of Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and accuracy at different top-K results (HIT@1, HIT@5, HIT@10). For the randomly split datasets, we run the experiments five times, and the table shows the mean performance from five separate runs. On the other hand, we run our experiment once for the time-wise split dataset.

In the random split, IQLoc achieves a MAP of 0.493, indicating that, on average, the relevant documents (a.k.a., buggy source code) rank higher than the irrelevant ones. Similarly, the MRR is 0.520, suggesting that the first relevant document is, on average, found within the top 2 positions. IQLoc achieves a HIT@1 of 0.423,

indicating that 42.3% of the bug reports have their relevant documents (a.k.a., buggy source code) retrieved as the top-ranked result. It also achieves a HIT@5 of 0.647, indicating that ≈65% of the bug reports have returned at least one buggy document within the top 5 positions, whereas the HIT@10 measure is 0.721.

In the time-wise split, IQLoc demonstrates higher performance, achieving a MAP of 0.520 and an MRR of 0.553. Besides, a HIT@1 of 0.466 indicates a substantial performance improvement over its counterpart above. However, the improvement is more noticeable for HIT@5 and HIT@10 metrics, where nearly 67% of the bug reports that have at least one buggy document retrieved within the top 5 positions and 73.5% within the top 10 positions, respectively. These metrics suggest that IQLoc consistently performs well across different data splits. Moreover, its higher performance across all metrics in the time-wise split implies that IQLoc potentially captures temporal trends from past bug reports (i.e., time-wise split) and source code versions to identify recent bugs within the code.

In our approach, we capture the top 100 results retrieved by the Elasticsearch module for subsequent reranking (Step 2, Fig. 3.3). Previous studies in information retrieval have also used a subset of results to rerank [83, 122, 156]. Our decision was made after carefully analyzing different top-K results. As demonstrated in Table 3.8, for the random split test set, we observed a HIT@100 of 0.889 and a HIT@200 of 0.929. That is, for ≈89% of the bug reports, a relevant result can be found within the top 100 search results, and for ≈93% of the reports within the top 200 search results. This is a slight 4.5% increase by considering an additional 100 results. For the time-wise split, this difference is even smaller, only 2.6%. Besides, our reranking step relies on a Transformer-based cross-encoder model, which demands significant computing power. By considering only top 100 results, we thus strike a balance between the relevance of the results and the management of our computational resources.

We also evaluate IQLoc's performance in localizing different types of bug reports: bug reports containing Stack Trace (ST), Program Element (PE), and Natural Language (NL), as discussed in Section 3.4.1. In this case, we consider the Elasticsearch as a traditional baseline adapted from Apache Lucene [57]. Table 3.9 presents the results. In the time-wise split test set (Table 3.9a), IQLoc outperforms Elasticsearch by 15.77% and 17.15% in MAP and MRR, respectively for bug reports containing

Table 3.9: Performance of IQLoc for Different Classes of Bug Reports

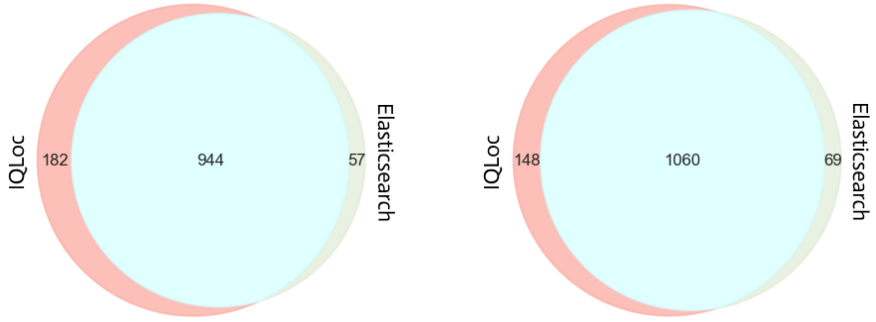
(a) Time-wise Split

Model	MAP	MRR	HIT@1	HIT@5	HIT@10
ST					
Baseline	0.488	0.513	0.443	0.594	0.686
IQLoc	0.565	0.601	0.535	0.698	0.730
PE					
Baseline	0.599	0.621	0.505	0.768	0.874
IQLoc	0.689	0.737	0.674	0.811	0.863
NL					
Baseline	0.442	0.467	0.350	0.612	0.709
IQLoc	0.460	0.487	0.382	0.626	0.720

(b) Random Split

Model	MAP	MRR	HIT@1	HIT@5	HIT@10
ST					
Baseline	0.492	0.513	0.421	0.637	0.721
IQLoc	0.561	0.592	0.503	0.708	0.755
PE					
Baseline	0.582	0.632	0.542	0.759	0.819
IQLoc	0.655	0.687	0.602	0.771	0.843
NL					
Baseline	0.423	0.454	0.353	0.592	0.680
IQLoc	0.432	0.456	0.350	0.596	0.686

stack traces (ST). It also improves such localization by detecting at least one buggy document in the top-10 positions, with gains of 6.41%–20% in HIT@1, HIT@5, and HIT@10. For bug reports with program elements (PE), performance improvements range from 5.6% to 33.46% in the time-wise split, with a 1.2% drop in HIT@10. In contrast, our techniques improves in all metrics for the bug reports containing only natural language (NL), but the gains are smaller. IQLoc achieves only a 4.07% increase in MAP and 4.28% in MRR. HIT@K improvements range from 1.57% to 9.14%, which is lower than the gains for ST and PE. For the random split dataset, we observe a similar trend in localizing bug reports containing stack traces (ST) and program elements (PE). For ST, IQLoc outperforms Elasticsearch by 4.72%–14.02% across all metrics. For PE, the improvement ranges from 1.58% to 12.54%. However, similar to the time-wise split, performance gains for NL bug reports remain minimal,



(a) Random Split (b) Time-wise Split

Figure 3.8: Impact of Query Reduction on Retrieval Performance

ranging from 0.44% to 2.13% across all metrics except HIT@1.

We also demonstrate how our reduced queries improve bug localization through document reranking in IQLoc. To illustrate this, we present two analyses from two different perspectives. Figure 3.8 highlights the benefit of incorporating our reformulated queries during the reranking step by comparing two methods: baseline Elasticsearch and IQLoc, both evaluated using the top-10 results. In our analysis of randomly split test sets, we found that both techniques retrieved buggy source documents for 944 bug reports. However, IQLoc localizes at least one buggy document for 182 more bug reports that Elasticsearch could not. A similar pattern can be observed in the time-wise split dataset, where IQLoc localized 148 more bugs for which Elasticsearch could not succeed. It should be noted that baseline Elasticsearch also localized some unique bugs, but they were much lower in number.

These analysis above led us to investigate the types of bug reports for which IQLoc might excel or fail during localization. Figure 3.9 presents our findings for both time-wise and random splits. In the randomly split test dataset, IQLoc successfully localized the majority of bug reports containing stack traces (ST), accounting for 96.0% of the localized cases, compared to 71.1% for those containing program elements (PE) and 32.9% for bug reports classified as natural language (NL). A similar pattern is observed in the time-wise split, where 95.7% of bug reports with stack traces were localized, followed by 63.5% with PE and 40.8% with NL.

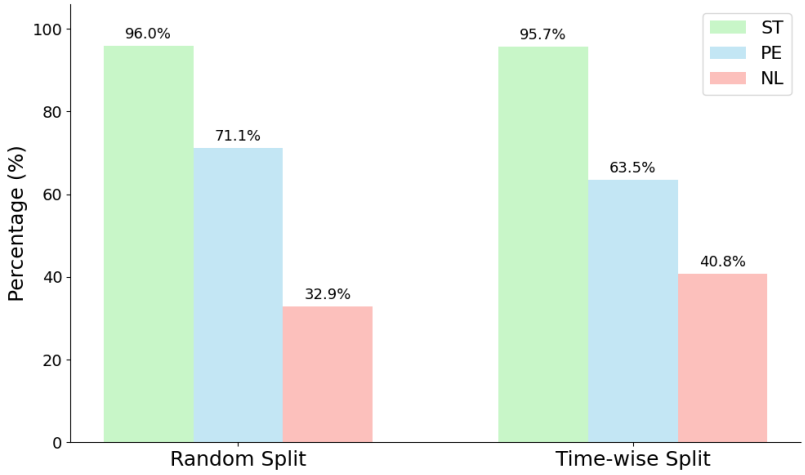
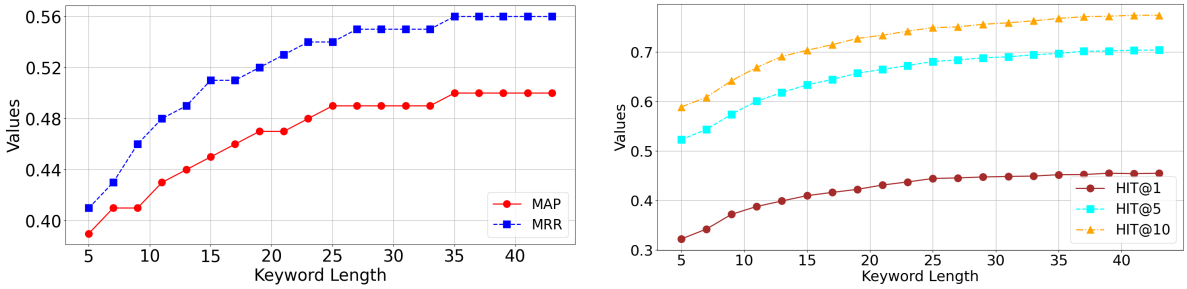


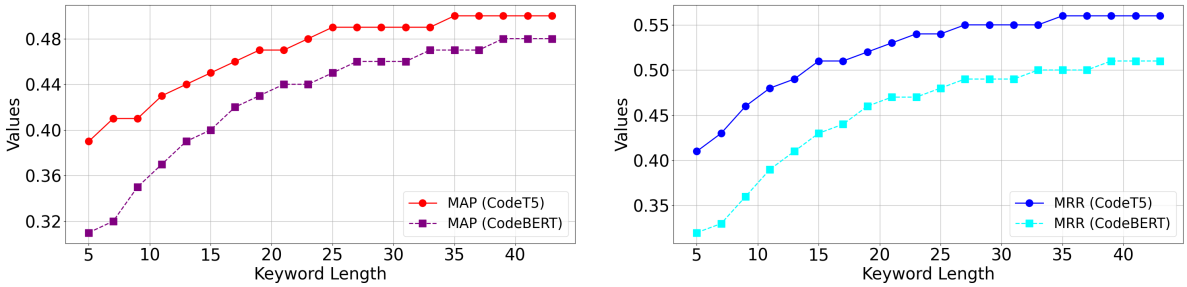
Figure 3.9: IQLoc's Performance for Different Types of Bug Reports

RQ1 Summary: IQLoc demonstrates promising performance in localizing bugs, achieving a MAP score of up to 0.520— a 10.43% improvement over baseline Elasticsearch. This improvement is driven by our query reduction strategy, which uses Transformer-based reasoning of the buggy code to better match between a query and the code. Additionally, IQLoc excels at handling various types of bug reports, successfully localizing up to 96.0% of those containing stack traces.



(a) MAP & MRR (b) HIT@K

Figure 3.10: Impact of Query Length on Bug Localization



(a) MAP : CodeT5 vs. CodeBERT (b) MRR : CodeT5 vs. CodeBERT

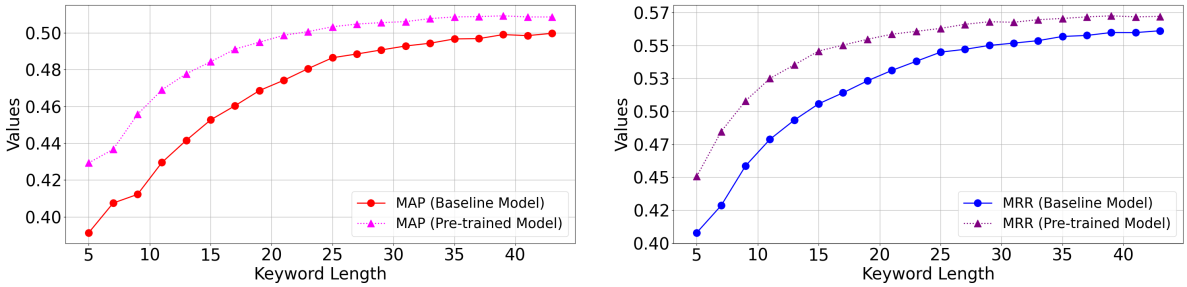
Figure 3.11: Choice of Pre-trained Models for Query Reformulation

Answering RQ$_2$: Impact of Query Length and Embedding Models on IQLoc's Performance

Fig. 3.10 shows how the length of our search queries (N) influences the performance of IQLoc. Selection of search keywords is a crucial aspect of our technique since the keywords can narrow down the search scope and position the relevant documents at higher ranks during bug localization. However, determining the optimal number of keywords in a query without compromising bug localization performance poses a challenge.

To determine an optimal length for search queries (N), we use a fixed pre-trained model, the baseline CodeT5 [168] model, during document reranking step while varying the length of queries. From Fig. 3.10 we see that the HIT@1 increases from 0.32 with 5 keywords to a maximum of 0.455 with a keyword length of 43, representing a ≈42% increase. Such a trend is consistent across other metrics, with improvements of 27.64%, 37.61%, 34.61%, and 31.52% for MAP, MRR, HIT@5, and HIT@10, respectively. However, we observe diminishing returns in performance improvement for HIT@K as query length increases, reaching a plateau around the length of 15, where performance improvement slows down. To balance between performance and query specificity, we thus chose a maximum query length of 15 for our experiments.

We also investigate the role of embedding models in our query reformulation step during bug localization, as the quality of reformulated queries directly impacts localization effectiveness. Fig. 3.11 compares the performance of CodeT5 and CodeBERT in terms of MAP and MRR to determine which model is more effective for generating reformulated queries in our proposed technique. Reformulated queries using



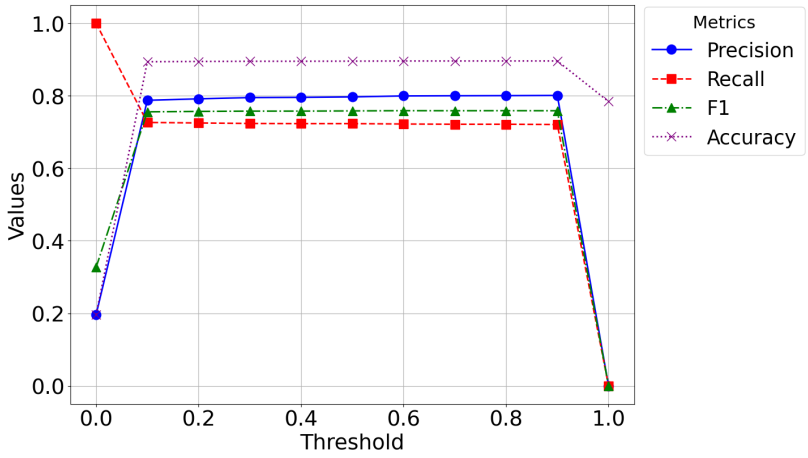
(a) MAP: Baseline vs. Pre-trained (b) MRR: Baseline vs. Pre-trained

Figure 3.12: Choice of Pre-trained, Domain-Specific Embedding Model for Query Reformulation

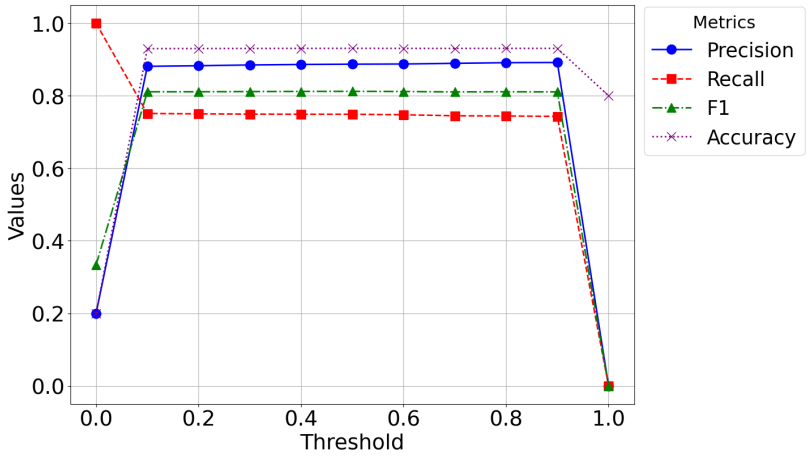
CodeBERT achieve a MAP score of 0.306 for a keyword length of 5, which is 27.90% lower than CodeT5's score of 0.392. Although increasing the query length reduces the performance gap, CodeBERT does not surpass CodeT5. For instance, at a keyword length of 43, CodeBERT still performs 3.71% lower than CodeT5. A similar trend is observed for MRR (Fig. 3.11b), where CodeBERT lags by 27.35% at a keyword length of 5. Based on these performance differences, we selected CodeT5 for embedding generation in our keyword extraction module, as it leads to more effective query reformulations and thus enhances the overall bug localization performance.

We also wanted to investigate if domain-specific embedding helps in constructing queries. Recognizing the importance of domain-specific pre-training to capture nuanced language features [149], we pre-trained the *CodeT5-small* [168] model on a comprehensive dataset containing bug reports (see Section 3.3.5 for details). This pre-training aimed to equip the model with a deeper understanding of software bugs from Java-based systems. Our findings indicate that the pre-trained model consistently outperforms the baseline model across all metrics, achieving superior performance even with fewer keywords. From Fig. 3.12, we see that the MAP improves from 0.39 to 0.42, representing a 9.68% increase over the baseline model using 5 keywords. Similarly, it increases from 0.45 to 0.48 with 15 keywords, indicating a 6.98% improvement. This trend holds true across various query lengths experimented with, ranging from 5 to 43 for all metrics. Therefore, we choose the CodeT5 model pre-trained with bug reports for our technique - IQLoc.

RQ2 Summary: Reformulated queries improved the performance of IQLoc in bug localization (e.g., HIT@1 by ≈42%) over baseline queries (i.e., bug report), with improvement maximizing on 15 keywords in a query. Similarly, a language model pre-trained with bug reports enhances the performance of IQLoc (e.g., 9.68% for MAP) by offering domain-specific embedding, with CodeT5 performing best among the models evaluated.



(a) CE Performance: Random Split



(b) CE Performance: Time-wise Split

Figure 3.13: Cross-Encoder's Performance at Different Relevance Thresholds

Answering RQ_3: Performance of the Cross-Encoder Model in Determining Buggy Source Code Based on Program Semantics

In this section, we investigate the effectiveness of the cross-encoder model (adopted by IQLoc) in identifying buggy code segments based on their program semantics. Transformer-based cross-encoder models are designed to output probabilistic confidence scores ranging from 0 to 1. We fine-tuned our model to classify code segments as either buggy (i.e., 1) or non-buggy (i.e., 0) against on a given bug report. Fig. 3.13 illustrates the performance of our adopted cross-encoder model with various configurations.

For the random split evaluation set (Fig. 3.13a), at threshold 0, the model achieves the lowest accuracy of 19.5%, with the lowest precision and F1 scores and the highest recall. This occurs because, at this threshold, the model classifies all instances as positive (a.k.a., buggy), capturing all true positives (TP) but also misclassifying all negative cases as false positives (FP). Since recall is calculated as $TP/(TP + FN)$ and $FN = 0$ in this scenario, recall remains at 1 despite poor precision or accuracy. Once the threshold increases above 0 (e.g., 0.1), performance stabilizes, reaching an accuracy of 91.4%, with precision, recall, and F1 scores of 81.3%, 73.8%, and 75.82%, respectively. However, at threshold 1, the opposite effect occurs— the model classifies all instances as negative, correctly identifying all negative cases (TN) but misclassifying all positive cases (TP), leading to a recall of 0. Since our evaluation dataset consists of four times more negative cases than positive ones (discussed in Section 3.4.1), the accuracy at threshold 1 is approximately 80%. It is noteworthy that these cross-encoder results for the random split evaluation set are averaged over five independent runs.

In the time-wise split scenario (Fig. 3.13b), our cross-encoder model exhibits similar trends for thresholds 0 and 1. However, once the threshold surpasses 0.1, the model stabilizes and demonstrates slightly improved performance across all metrics. Starting with accuracy, the model shows a consistent upward trend, reaching ≈93%, which is a 1.9% improvement over the random split dataset scenario. Similarly, precision, recall, and F1 scores remain relatively stable, hovering around 88.7%, 74.9%, and 81.2%, respectively.

The precision of our cross-encoder models at both very high and low thresholds

suggests that it makes predictions closer to 1 and 0 for positive (contextually relevant) and negative (contextually not relevant) outcomes, respectively, while maintaining accuracy. Given our focus on the correctness of positive predictions (i.e., precision) while maintaining overall correctness (i.e., accuracy), we set the threshold to 0.5 for our experiments, which achieves the best balance between precision and recall, as reflected in the F1 score.

RQ3 Summary: Our fine-tuned cross-encoder model can effectively identify buggy and non-buggy source code segments leveraging their program semantics and relevance to given a bug report. A threshold of 0.5 ensures an effective separation between the two types of source code by achieving up to 93% accuracy and 81.1% precision.

Answering RQ_4: Comparison with the Baseline Techniques

In this section, we compare our proposed technique IQLoc against existing bug localization techniques in terms of various evaluation metrics. In particular, we compare IQLoc with five baseline techniques – Baseline Elasticsearch, BLUiR [136], Blizzard [129], DNNLoc [92], and RLocator [29].

To replicate the baseline Elasticsearch technique, we indexed all source documents of a subject system's repository and capture the pre-processed bug reports as queries. Then, we execute the queries with the Elasticsearch engine [6], which retrieves the relevant source documents based on the BM25 algorithm [10] and Boolean query [13]. We use the default values for the parameters - k and b - in the BM25 algorithm.

BLUiR [136] employs a structured Information Retrieval approach where it leverages the structural components of both source code documents and bug reports for effective localization, which helps avoid spurious matching. To achieve this, BLUiR collects four code elements (i.e., class names, method names, variable names, and comments) from each source code document and two textual elements from each bug report (i.e., bug title and description). Then, it conducts eight searches to compute suspiciousness score for each code-text pair and then combines scores to calculate the overall suspiciousness score for a source document. For the experiments, we collected BLUiR's replication package from the Bench4BL repository [96] and adapted it for

version-based replication. It uses Indri [148], a TF-IDF [97]-based search engine, as its backend for experiments, which has become obsolete recently. Thus, we chose to replicate the technique using Apache Lucene [57] with a TF-IDF-based scoring algorithm. It should be noted that we adopted the formula suggested by Saha et al. [136] for calculating the TF and IDF metrics, which were then fed to the TF-IDF-based retrieval algorithm of Apache Lucene.

Blizzard [129] is an IR-based bug localization technique that leverages contextual information from bug reports for query construction. It categorizes bug reports into three types (i.e., NL-Natural Language, PE-Program Element, and ST-Stack Trace), employs three separate graph-based techniques to construct queries from them, and then retrieves buggy source documents from the corpus by executing the queries with the Apache Lucene engine [57]. For the replication, we collected and adapted the replication package hosted at GitHub [2] by the original authors.

DNNLoc [92] is the first technique for bug localization that combines Deep Learning and Information Retrieval. It uses several features – bug report-source code similarity (rVSM score [84]), class name similarity, collaborative filtering, bug report recency, and bug report frequency to train a deep learning model and then uses the model to localize the bugs. To replicate DNNLoc, we trained appropriate models by extracting features from our training sets, following the authors' suggestions. During bug localization, we used these trained models to predict suspiciousness scores of source code documents and to rank them.

RLocator [29] is a recent IR-based technique for bug localization that incorporates reinforcement learning [80], modeling the localization task as a Markov Decision Process (MDP) [171]. Before applying reinforcement learning (RL), it retrieves source documents from Elasticsearch based on the bug report and filters suspicious documents using an XGBoost [34] model. It then employs an RL agent based on an actor-critic framework [88] that attempts to rank relevant source documents at the top positions, with MAP or MRR as reward signals. We collected the replication package of RLocator from Zenodo [11] and adapted it for our study on version-based bug localization while maintaining the original specifications of the authors.

From Table 3.10a (random split dataset), we see that existing techniques such as baseline Elasticsearch achieve a MAP score of 0.457, while BLUiR, Blizzard, and

Table 3.10: Comparison between IQLoc and Baseline Techniques in Bug Localization

(a) Average Performance Metrics for Random Split

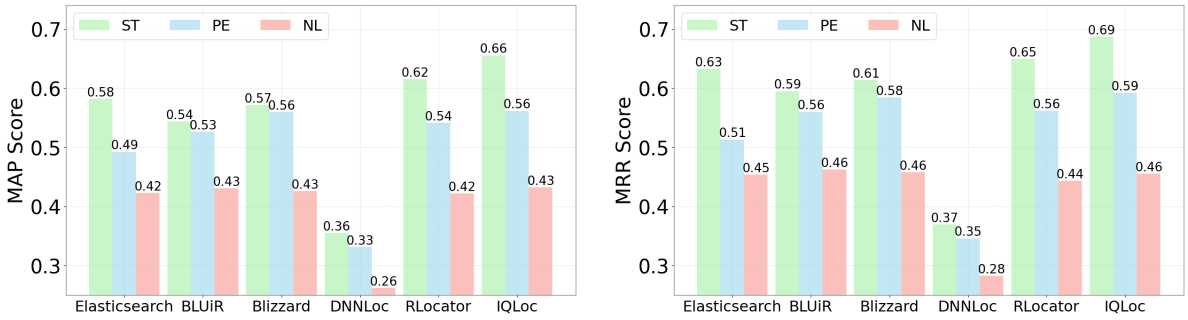
Technique	MAP	MRR	HIT@1	HIT@5	HIT@10
Baseline Elasticsearch	0.457	0.486	0.389	0.618	0.703
BLUiR	0.470	0.504	0.393	0.655	0.745
Blizzard	0.480	0.510	0.410	0.648	0.735
DNNLoc	0.311	0.322	0.249	0.416	0.508
RLocator	0.478	0.511	0.419	0.652	0.726
IQLoc	0.493	0.520	0.423	0.647	0.721

(b) Performance Metrics for Time-wise Split

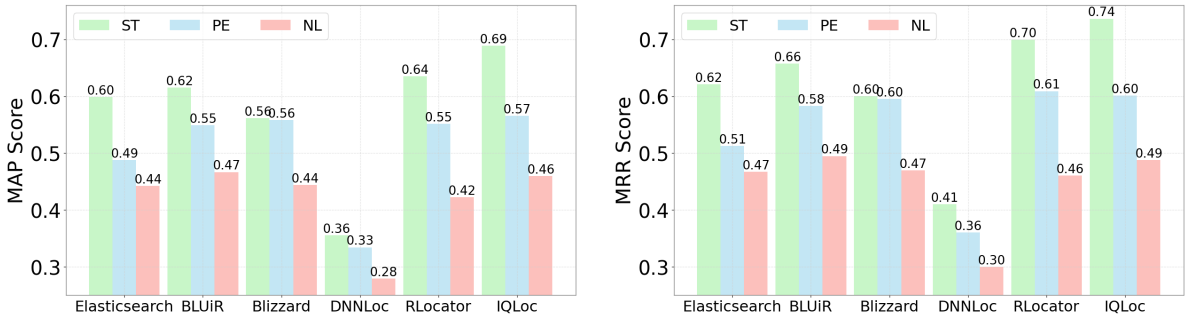
Technique	MAP	MRR	HIT@1	HIT@5	HIT@10
Baseline Elasticsearh	0.471	0.496	0.396	0.618	0.714
BLUiR	0.508	0.539	0.429	0.686	0.786
BLIZZARD	0.494	0.525	0.419	0.670	0.753
DNNLoc	0.324	0.336	0.232	0.477	0.591
RLocator	0.505	0.532	0.439	0.683	0.731
IQLoc	0.520	0.553	0.466	0.669	0.735

RLocator achieve a comparative MAP score from 0.470 to 0.480, indicating ≈2.71%-4.89% improvement. On the other hand, DNNLoc achieves a much lower MAP score of 0.311. IQLoc, on the other hand, has a MAP score of 0.493, which is 2.71% to 58.52% better than the baseline scores. This shows that relevant documents rank higher than irrelevant ones compared to other baseline techniques. Similar improvements are observed for MRR and HIT@1, with increases of up to 61.49% and 69.88%, respectively, indicating promising performance. Although IQLoc exhibits a marginal decrease in HIT@5 and HIT@10 compared to BLUiR, Blizzard, and Rlocator, it demonstrates an improvement of up to 55.53% and 41.9% over the baseline Elasticsearch and DNNLoc measures.

In Table 3.10b (time-wise split dataset), IQLoc's performance also surpasses those of baseline techniques. While Elasticsearch achieves a MAP score of 0.471, BLUiR, Blizzard, and RLocator achieve comparative MAP scores between 0.494 and 0.508; DNNLoc's score is much lower at 0.324. In contrast, IQLoc achieves a MAP score of 0.520, representing improvements ranging from 2.36% to 60.49% over the baseline measures. Similar improvements are observed for MRR and HIT@1, ranging from 2.59%-64.58%, and 6.15%-100.9%, respectively. However, like the randomly split data



(a) Random Split

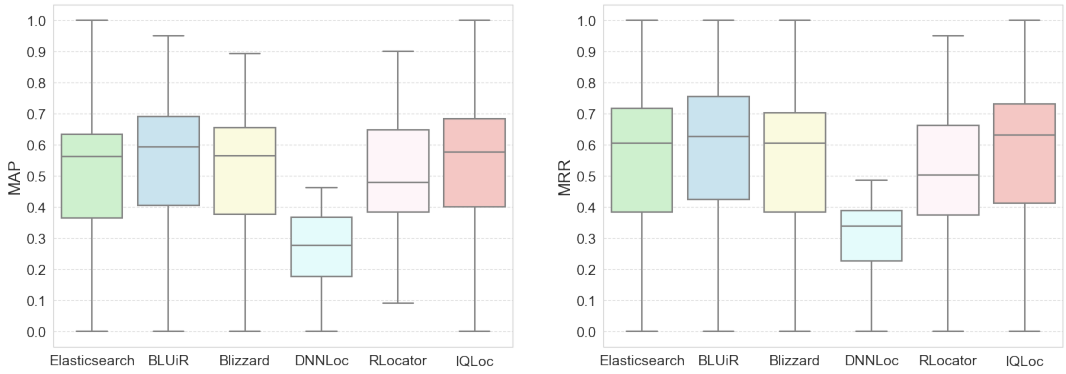


(b) Time-wise Split

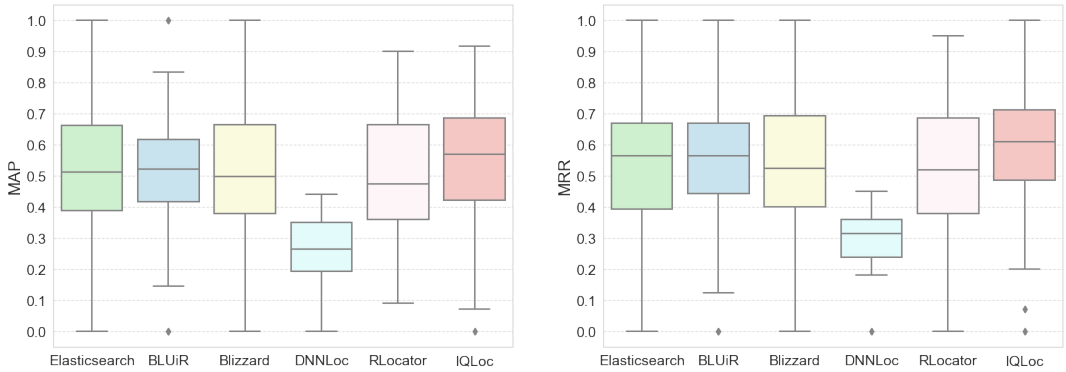
Figure 3.14: Comparison of Baseline Techniques in Localizing Different Types of Bugs

(Table 3.10a), IQLoc experiences a marginal drop in HIT@5 and HIT@10 compared to BLUiR and Blizzard. Nonetheless, it demonstrates improvements of up to 40.25% and 24.37% over the baseline Elasticsearch and DNNLoc measures in these metrics.

We also demonstrate the effectiveness of IQLoc in localizing bugs from different types of bug reports compared to baseline techniques. Fig. 3.14 presents the performance of various methods in localizing bugs from the reports with Stack Trace (ST), Program Elements (PE), and Natural Language (NL) [129]. For the random split test set (Fig. 3.14a), Elasticsearch, BLUiR, and Blizzard show similar MAP scores (0.54–0.58) for ST bug reports, while RLocator performs slightly better at 0.62. IQLoc outperforms all, achieving 0.66 MAP, a 6.42% improvement over RLocator. In the case of bug reports with program elements (PE), most techniques score between 0.53 and 0.56, except Elasticsearch, which lags at 0.49, while IQLoc leads with 0.56 MAP. DNNLoc consistently underperforms, scoring 0.36 for ST and 0.33 for PE. For natural language-only (NL) bug reports, all techniques struggle, with IQLoc performing comparably. A similar trend is observed in MRR, where IQLoc improves



(a) Random Split



(b) Time-wise Split

Figure 3.15: Performance of Different Techniques on Different Subject Systems

by 6.15%–86.49% for ST and 1.72%–126.92% for PE. In the case of natural language bug reports (NL), MRR shows an improvement reaching ≈ 64%.

A similar pattern was observed in the time-wise split test set (Fig. 3.14b), where IQLoc achieves 7.81%–91.67% improvement in ST and 3.63%–72.73% in PE in terms of MAP. For MRR, IQLoc shows up to 80.48% improvement in bug reports with stack traces and 66.67% improvement in those with program elements. However, for bug reports containing only natural language, all techniques perform similarly, except for DNNLoc, which remains the lowest-performing model.

We further analyze each techniques performance across 42 subject systems and compare their MAP and MRR using box plots (Fig. 3.15). For the random split test set (Fig. 3.15a), IQLoc achieves a higher median MAP than that of all competitors except BLUiR. However, our technique outperforms others in terms of median MRR.

For the time-wise split test set (Fig. 3.15b), IQLoc outperforms all techniques with

Table 3.11: Statistical Test: IQLoc vs. Baselines

(a) Time-wise Split (MAP)

IQLoc *vs*	*p-value*	Effect-Size (Cliff's δ)
Elasticsearch	0.0098	Large (0.71)
BLUiR	0.0176	Large (0.58)
Blizzard	0.0209	Medium (0.49)
DNNLoc	0.0063	Very Large (0.78)
RLocator	0.0228	Medium (0.47)

(b) Time-wise Split (HIT@1)

IQLoc *vs*	*p-value*	Effect-Size (Cliff's δ)
Elasticsearch	0.0108	Large (0.52)
BLUiR	0.0257	Medium (0.42)
Blizzard	0.0181	Medium (0.44)
DNNLoc	0.0093	Very Large (0.70)
RLocator	0.0236	Medium (0.42)

a higher median value and a more compact interquartile range (IQR) for both MAP and MRR, suggesting lower variability in performance. While BLUiR also shows a compact IQR, its median value is lower. DNNLoc, despite being less variable, consistently performs poorly across all subject systems, reinforcing its less effectiveness. Overall, IQLoc demonstrates superior performance in localizing bugs across different subject systems with a higher median measure and a reduced variability in performance.

IQLoc's performance improvements over the baseline measures demonstrate its effectiveness across both dataset splits. To further validate this efficacy, we conducted *Mann-Whitney Wilcoxon* (a.k.a., *Mann–Whitney U test*) [18], a non-parametric statistical test. We use the SciPy library [78] to compute statistical significance of MAP and HIT@1. For MAP, we capture all buggy documents that match the ground truths, while for HIT@1, we focus on the first buggy document at top rank in the entire test dataset. Both metrics are computed by comparing the results from IQLoc and the baseline techniques using the test set, which contains queries from 42 subject systems. For each query, the ranked results are converted into binary vectors based on their relevance to the ground truth, to assess their effectiveness in localizing bugs. In the case of the time-wise split dataset, IQLoc demonstrated *p-values* ranging from 0.0063 to 0.0228 when compared against baseline techniques (Table 3.11a) for MAP. These *p-values* are below the significance threshold of 0.05, indicating statistical significance. Moreover, IQLoc exhibited effect sizes ranging from medium to very large, as measured by Cliff's δ [18], further explaining the extent of differences. Similarly, for HIT@1, IQLoc achieved comparable results, with *p-values* varying from 0.093 to 0.257, accompanied by similar effect sizes ranging from medium to very large (Cliff's δ, Table 3.11b). Note that we avoid measuring statistical significance for the

randomly split dataset, as we perform five independent runs and average the results for our experiment. Given the evidence above, the null hypothesis is rejected, and IQLoc's performance is found to be significantly higher than the baseline measures.

RQ4 Summary: IQLoc outperforms baseline techniques significantly, with MAP performance improvements of 58.52% for the random split dataset and 60.49% for the time-wise split dataset. This improvement stems from its ability to localize various types of bugs across different subject systems. Additionally, statistical significance tests confirm the superiority of our technique over baseline techniques with medium to large effect sizes.

3.5 Related Work

3.5.1 IR based Bug Localization

Bug localization has been a key area of research for decades, driven by the significant impacts and challenges of software bugs [187]. There are two primary categories of bug localization: spectra-based and Information Retrieval (IR)-based [163]. Spectra-based methods are known for their complexity and limited scalability [110, 162]. In our work, we primarily focus on Information Retrieval for bug localization.

Traditional IR-based bug localization methods [84,129,136,164,165] typically rely on the vector space model (VSM) [95], which analyzes the token overlap between bug reports and source code to identify buggy documents. Several studies have extended VSM by integrating additional contexts, such as bug report history [135], code change history [172], or version history [143], into the IR process. For instance, Zhou et al. introduced BugLocator [84] that employs a combined score of a modified VSM score and previous bug fix history to localize bugs. The traditional VSM-based scoring [95] method tends to show bias towards longer documents. To address this, they modified the VSM-based scoring and introduced *rVSM* scoring, which enhances the computation of textual relevance between bug reports and code elements. BLUiR [136] captures four types of structural components from the source code (i.e., class names, method names, variable names, and comments) and two components from bug reports (i.e., bug title and bug description). It then forms pairwise combinations of

these components to perform eight separate searches using Indri [148], leveraging a modified TF-IDF approach. The final localization score is computed by summing the scores obtained from these individual searches. AmaLgam [164] integrates BLUiR and BugLocator techniques, along with version history inspired by Google by analyzing historical data from version control systems. These components undergo three independent systems to generate rankings before producing the final ranked lists. AmaLgam+ [165] takes a step further by incorporating stack trace and bug reporter history, alongside AmaLgam's contexts. It ranks files using five components and then returns a final ranked list of source documents for bug localization. Recently, an IR-based technique, PathIdea [32], utilizes bug report logs (e.g., log snippets, stack traces) for bug localization. The authors employ a static analysis tool to construct a file-level call graph and reconstruct system execution paths from the logs. To determine the suspiciousness score for each file, they combine the VSM score, a log score that emphasizes files mentioned in the logs, and a path score that highlights files in the execution path. Some IR-based bug localization methods employ more complex mechanisms, such as Latent Dirichlet Allocation (LDA) [84] and Latent Semantic Index (LSI) [121]. Nonetheless, simpler methods have shown comparable performance while being more cost-efficient [96].

In our work, we employ a BM25-based approach (e.g., Elasticsearch [6]) to collect the candidate source documents. Then, our subsequent modules combine the scalability of textual relevance with Transformer-based code understanding grounded in program semantics to retrieve buggy documents at top-ranked positions.

3.5.2 Query Reformulation

Studies by Mills et al. [109] and Rahman et al. [127] suggest that poor queries from bug reports may adversely affect the performance of a VSM-based search. Consequently, numerous approaches propose to construct queries to assist developers in their tasks [69, 86, 129, 130]. These studies for query construction fall into two categories: frequency-based and graph-based keyword selection methods [127].

In frequency-based methods, researchers have utilized TF-IDF [97] and its variants to extract meaningful keywords from both bug reports and source code for use as queries. For instance, Gay et al. [62] employed relevance feedback (RF) from users

to update the query. They implemented Rocchio's expansion [42] method, enhancing query performance by adding terms from relevant documents with increased weights and suppressing or removing terms from irrelevant documents. Haiduc et al. [69] proposed a technique that employs a machine learning model trained on 28 query properties to recommend the best reformulation strategy for a given query. These strategies include query reduction, Rocchio expansion, RSV expansion, or Dice expansion, selected based on the query's properties and performance.

Graph-based methods analyze semantic and syntactic dependencies among words to determine their importance. Rahman and Roy [129] applied the PageRank algorithm [24] on constructed graphs to suggest search keywords from various sources. Subsequently, they explored genetic algorithms [93] for near-optimal search query construction, leveraging the vocabulary in bug reports for effective query building [127]. However, their genetic algorithm-based approach is costly and relies solely on the textual relevance between bug reports and source documents.

These techniques typically use statistics and correlations to generate queries from bug reports. However, they often fail to capture the contextual relevance between bug reports and code which limits the effectiveness of query reformulations. In our approach, we address this by leveraging the broader understanding of natural language and program texts from a pre-trained language model and identifying the most salient terms from both bug reports and code for query construction.

3.5.3 Deep Learning for Bug Localization

Recent advancements of deep learning in various domains (e.g., natural language processing, and machine translation) have encouraged its application to bug localization. DNNLOC, an early and influential work in the field of localization, is designed to identify potentially buggy documents by learning from multiple features (i.e., rVSM score, class name similarity, collaborative filtering, bug report recency, and bug report frequency) of bug reports and source code. However, its reliance on certain features like bug fixing recency and frequency is available for only 20-40% of the bugs [84]. So, the unavailability of these features might affect the performance of the model. A recent technique, FBL-BERT [37], incorporates the ColBERT model [82] for changeset-based bug localization. ColBERT uses a late interaction architecture,

independently encoding bug reports and changesets with BERT, followed by efficient vector similarity calculations for relevance estimation. FBL-BERT enhances this by considering different levels of changeset granularity, enabling offline pre-computation of changeset embeddings, and employing a two-stage process of retrieving and reranking for bug localization. Nonetheless, the model may not be as effective in large-scale environments, where big software projects produces thousands of commits a day. Another recent bug localization technique, RLocator [29], is a first-of-its-kind reinforcement learning (RL) technique for bug localization. The technique attempts to optimize the ranking of a set of documents by employing RL agents based on an actor-critic [88] framework with entropy regularization, using a reward signal derived from ranking metrics (e.g., MAP and MRR) for localizing bugs. Other approaches, such as DeepLoator [178] proposed bug localization using Convolutional Neural Networks (CNN). DreamLoc [123] addresses the problem of bug localization by proposing a deep and wide architecture. The deep component applies attention mechanisms to capture textual relevance, while the wide component incorporates domain knowledge through a linear model using features such as bug-fixing history and code complexity. However, in addition to technical challenges, these techniques may encounter inefficiencies when dealing with large numbers of documents for bug localization.

In contrast, our hybrid approach addresses these challenges by focusing on a limited set of documents using Transformer models for scalability and performance. Leveraging the self-attention mechanism of the Transformer model, we process both source code and bug reports simultaneously and analyze their semantics, which helps detect their contextual relevance. This methodology offers a promising avenue for overcoming the limitations faced by existing bug localization techniques.

3.6 Threats to Validity

Threats to *internal validity* relate to experimental errors and biases. Re-implementation of the existing baseline techniques could pose a threat. For BLUiR, Blizzard and RLocator, we collected the replication packages from Bench4BL, Rahman et al.'s GitHub repositories and from Zenodo [3, 11, 136]. However, since the Indri [148] library has become obsolete, we replaced that with Apache Lucene [57] in the BLUiR replication. For DNNLOC, as the replication package is not available by the original

authors, we had to replicate it ourselves. While we acknowledge the potential for implementation errors, we addressed this concern by adhering to the settings and parameters of the original authors through extensive testing. We also repeat our experiments on two different datasets [84, 129] and compare the performance with baselines to mitigate any bias due to random trials.

Threats to *external validity* relate to the generalizability of our work. Even though IQLoc is evaluated using only Java code, the underlying models can be adapted to different programming languages through appropriate fine-tuning [186].

Threats to construct validity relate to the evaluation metrics for our work. We use several metrics such as Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K, which are widely adopted in recommendation systems [129,131, 136] and Information Retrieval [28,71]. This confirms no or little threats to construct validity.

Finally, we adapted Bench4BL [96] dataset for our experiments, which might contain biases [87] and data quality issues (e.g., misclassified bugs, erroneously labeled buggy files). During the training of the Cross-Encoder model, we accepted method bodies containing buggy lines from Java classes as context. However, the impact of different context sizes was not well tested, which we consider as a scope of future work.

3.7 Summary

To summarize, in this study, we introduce IQLoc, a hybrid approach that capitalizes on the strengths of both Information Retrieval (IR) and LLM-based program understanding to support bug localization. Our approach enhances IR-based bug localization with reformulated queries, derived from the program understanding of Transformer models. By going beyond surface-level semantic relevance, IQLoc identifies buggy code more accurately. In our evaluation, we compared IQLoc against several baselines using three key metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. The results show that IQLoc consistently outperforms the baselines, with improvements of up to 60.49% in MAP and 64.58% in MRR.

All these findings highlight the potential of IQLoc as a robust technique for bug

localization that seamlessly integrates the textual relevance of traditional techniques with the program semantic understanding of modern Transformer-based models, setting a new benchmark for performance and reliability in addressing software bugs. However, while our technique significantly improves bug localization performance, it might fall short for certain types of bug reports that warrant deeper contextual understanding and more effective search queries. To address this, Chapter 4 introduces a novel technique that introduces Intelligent Relevance Feedback leveraging LLM to further advance bug localization using Information Retrieval.

Chapter 4

Improved IR-based Bug Localization with Intelligent Relevance Feedback

Our first study in Chapter 3 incorporates program understanding of pre-trained language models (e.g., CodeBERT, CodeT5) into Information Retrieval methods for localizing bugs. However, it struggles when localizing bugs for the bug reports primarily written in natural language. In other words, despite being empowered by program semantic understanding, it suffers from less informative queries from bug reports. To address this limitation, we incorporate Intelligent Relevance Feedback (IRF) into Information Retrieval leveraging the reasoning capabilities of Large Language Models and support bug localization. Our evaluation using three performance metrics demonstrates that our proposed solution effectively localizes software bugs and outperforms multiple baseline techniques from the literature.

The rest of this chapter is structured as follows: Section 4.1 introduces BRaIn and outlines the key contributions of this study. Section 4.2 presents a motivating example to demonstrate the effectiveness of our approach. Section 4.3 presents the methodology of our proposed approach. Section 4.4 details the dataset construction process, metrics and presents evaluation results. Section 4.5 reviews existing relevant studies in the domain of bug localization. Section 4.6 examines potential threats to the validity of our findings. Finally, Section 4.7 concludes the chapter with a summary of our study.

4.1 Introduction

Software bugs can cause major financial losses and lead to data breaches, security vulnerabilities, and operational disruptions [41, 56]. A recent software bug from Microsoft-owned CrowdStrike caused several hours of disruption in the U.S. airline industry, nearly halting operations and resulting in over $10 billion in damages [170, 174]. Developers at the major IT companies, such as Microsoft and Google,

have reported bug resolution as a top concern [187]. According to existing studies up to 50% of the programming time is spent by developers on finding, understanding, and fixing software issues [25, 46, 115]. Thus, any automated support to tackle these challenges can greatly benefit the developers.

Software bugs are submitted to bug-tracking systems (e.g., Bugzilla, JIRA) as bug reports, which might capture crucial hints for resolving software-related issues. Developers often rely on these reports to trace the origin of bugs in the code. However, the content and quality of bug reports can vary significantly based on their submitters' level of expertise and articulation skills. In particular, there might be variations in word choice and presence of technical terms [60, 129]. Such variations pose challenges for developers when pinpointing the root cause of defects, even for seasoned practitioners [127]. To address these challenges, there has been significant research targeting the detection or localization of software bugs over the last few decades.

Researchers have presented two major categories of methods to automatically localize software bugs: program spectrum analysis and Information Retrieval. First, spectrum-based methods rely on program execution traces for fault localization. However, the execution traces are not always readily accessible, which makes these methods less scalable [110, 162]. On the other hand, Information Retrieval (IR)-based methods use overlapping terms or keywords between bug reports and source code to localize bugs [91, 94, 114, 163, 167]. They are lightweight and scalable. However, they also struggle with the *vocabulary mismatch problems* [60] and may not always deliver satisfactory results due to sporadic term matching. Researchers have also incorporated historical data from past bug reports, code change history, past bug fixes, and bug recurrences [164, 181]. Although these enhancements have been reported to improve the performance of the IR-based methods in localizing bugs, a recent study [96] suggests that they do not significantly outperform the previous methods.

Recent IR-based techniques focus on search queries and attempt to improve their queries by capturing syntactic, co-occurrence, and hierarchical dependencies among the words in bug reports [30, 127, 129, 143]. However, these methods only use terms found in bug reports, which could be poorly written or insufficient [127]. As a result, they frequently fail to bridge the gap between natural language from bug reports and

programming code from a project when searching for software bugs. To address this issue, several techniques attempt to enhance queries with relevant terms extracted from source documents through relevance feedback mechanisms [61,68,85,98,129,161,182]. However, the majority of these techniques naively consider the top few documents (based on textual similarity) as relevant, overlooking the need for a comprehensive understanding of the code. As a result, they may not always capture the most meaningful terms from source code for their search queries. [30, 85]. Thus, the existing IR-based techniques for bug localization suffer from two major challenges as follows.

(a) Relevance feedback against search queries might not be always relevant: Gay et al. [68] proposed a manual, iterative approach that leverages relevance feedback from developers and constructs queries to search for buggy source documents. In contrast, Sisman et al. [144] and Kim et al. [85] select the top few documents as relevant (a.k.a., pseudo relevance feedback) and leverage the feedback to improve their search queries. However, these techniques rely heavily on textually similar documents, which may not be always relevant, especially when dealing with source code and bug reports. Thus, a deeper understanding of both bug reports and source code is warranted to improve the relevance feedback mechanism and the subsequent steps of Information Retrieval (e.g., query reformulation, retrieval).

(b) Textual and semantic relevance might not be sufficient: Bug reports contain not only natural language texts but also technical jargons, commit diffs, stack traces, and program elements [129]. These artifacts describe the context and symptoms of encountered bugs [30]. Since natural language is loosely structured, it can introduce ambiguity by expressing the same idea in various ways [60]. Similarly, programming languages are more structured yet allow syntactically diverse expressions (e.g., iterative vs. functional approaches) and arbitrary naming conventions [14, 43, 144]. This flexibility can result in textual mismatches, where keywords or key phrases in the bug report (e.g., "*download failed*") do not directly match the identifiers in the code (e.g., `fetchResource`). At the same time, semantic mismatches can arise when a problem encountered in the bug report does not correspond to the programming task implemented in the code. For example, the encountered problem – "*download failed*" – might not align well with the task – "*HTTP/FTP operation task and get packets*" if the word-level semantics are considered only. It requires

an understanding of the relationship between network operations and file downloading to establish their connection. In other words, to localize such bugs, automated tools or methods need to go beyond surface-level matching and comprehensively understand the context of an encountered problem as well as the functionality of the corresponding source code.

In this paper, we present a novel technique – *BRaIn* – to support bug localization using Information Retrieval (IR) and Intelligent Relevance Feedback (IRF). Our approach overcomes the challenge of contextual understanding of software bugs using Transformer models [159] and localizes the bugs leveraging such understanding. First, BRaIn collects potentially buggy documents from a codebase using an IR method (e.g., BM25) and analyzes their contextual relevance to a bug by employing large language models (e.g., Mistral [151]). That is, unlike the existing methods, our method captures more expert-like feedback to a query (a.k.a., Intelligent Relevance Feedback). Second, it extracts appropriate terms from these documents and expands the original query (i.e., preprocessed bug report) by further leveraging the captured feedback. Finally, BRaIn reranks the source documents by executing the expanded query and employing the relevance feedback, providing a refined list of suspicious source documents.

We conducted experiments using 4,683 bug reports from a benchmark dataset– Bench4BL [96]. We evaluated the performance of our approach using three commonly used metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. Our approach is compared with six suitable baselines from the literature [29, 92, 129, 136, 144, 180]. BRaIn consistently outperformed existing techniques, showing 19.3% and 87.6% higher MAP scores than that of traditional and Machine Learning (ML)-based approaches, respectively. Similar gains were observed in MRR (17.5% and 89.5%) and HIT@10 (12.2% and 48.8%). These results underscore the effectiveness and superiority of our proposed technique in software bug localization.

Thus, this research make following contributions-

- A novel relevance feedback mechanism, namely *Intelligent Relevance Feedback (IRF)*, that leverages the code understanding and reasoning capability of the LLM to offer useful feedback to a search query. It is neither naive like pseudo-relevance feedback nor costly like human feedback.

- A novel approach – *BRaIn* – that localizes software bugs using effective search queries and retrieval, supported by the Intelligent Relevance Feedback mechanism.
- An extensive evaluation of *BRaIn* using three commonly used metrics and ≈4.7K bug reports and comparison with six baselines from three areas of the literature.
- A replication package [1] with a prototype, a curated dataset, and configuration details for third-party replication and reuse.

4.2 Motivational Example

In this section, we present a motivating example to demonstrate the benefits of our proposed technique for bug localization. Let us consider the example bug report in Table 4.1 that discusses access problems to an LDAP server. The bug manifests as a failure in the authentication process, where the system returns an HTTP code of 403 (a.k.a., forbidden) instead of prompting for necessary credentials. This behavior results in a denial of access to the LDAP services and hinders any migration to a newer version of the services.

Fig. 4.1 presents the source code triggering the bug. The root cause of this bug is a subtle omission in the code handling authentication process. We see that the switch statement in the buggy version of code fails to account for the BASIC authentication type. Instead, it handles the PLAIN authentication type, which is semantically closer but not equivalent. On the other hand, the bug report mentions "BASIC" HTTP authentication, which is not present in the target code. This terminology mismatch creates a disconnect between the high-level system behavior described in the bug report and the code level implementation, making the detection of bugs challenging. As a result, traditional text-based search methods perform poorly and retrieve the buggy code at 72^{nd}, 13^{th}, and 24^{th} positions when title, description, or their combination are used as queries, respectively (Table 4.1). Even after employing embedding-based semantic relevance, NextBug [180] struggles to link the code to the bug, placing it at 25^{th} position.

[1] https://github.com/asifsamir/BRaIn

Table 4.1: An Example of Bug Report and Search Techniques

Bug ID# 2013 (Wildfly CORE)		Rank
Title	Unable to access HTTP management interface secured by legacy LDAP realm.	72
Description	When the HTTP management interface is secured with a legacy security realm using LDAP, the user is not prompted to provide credentials as should be in the case of BASIC HTTP authentication mechanism. Instead, a 403 HTTP status is returned directly. Users won't be able to migrate their current (6.4, 7.0) configuration to 7.1 without change.	13
Baseline Query	Bug Title + Bug Description	24
NextBug [180]	Cosine Similarity (Embedding + TF-IDF) between Bug Report and Source Documents	25
BRaIn	Intelligent Relevance Feedback + Query Expansion + Scoring	1

```
@@ -236,7 +236,7 @@ public class SecurityRealmService
private AuthMechanism toAuthMechanism
    (String mechanismType, String mechanismName) {
        switch (mechanismType) {
            case "SASL": ...
            case "HTTP":
                switch (mechanismName) {
                    case "DIGEST":
                        return AuthMechanism.DIGEST;
-                   case "PLAIN":
+                   case "BASIC":
                        return AuthMechanism.PLAIN;
                }
                break;
        }
        return null;
    }
```

Figure 4.1: Buggy Code with Diff

The above evidence suggests that an in-depth analysis involving contextual relevance is essential. A seasoned developer would recognize the missing clause of BASIC HTTP authentication in the code, although it is not explicitly stated in the bug report. By probing deeper, they would infer that the missing BASIC authentication is likely the root cause of the reported issue.

Large language models (e.g., Mistral) are exposed to a vast amount of data, including text and code. As a result, they can identify patterns (as humans do) and infer missing details, making them adept at handling tasks that require deep contextual understanding. As shown in Table 4.1, BRaIn leverages such capabilities to obtain intelligent feedback against its query, reformulates the query, and returns the buggy code as the topmost result by executing the query against an IR-based method (e.g., BM25 [133]).

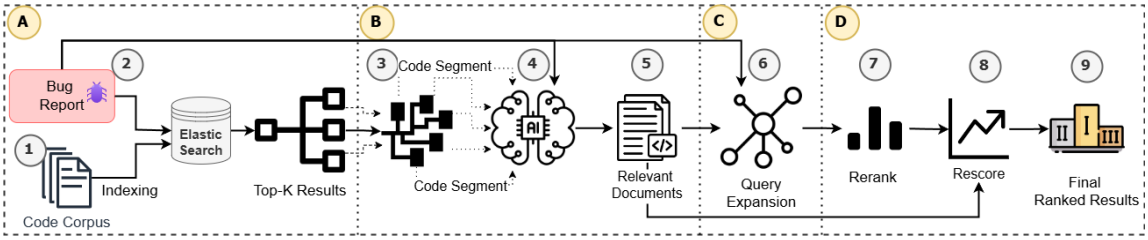


Figure 4.2: Schematic Diagram of *BRaIn*:
(A) Document Indexing & Retrieval, (B) Intelligent Relevance Feedback, (C) Query Expansion, and (D) Bug Localization

4.3 Methodology

Fig. 4.2 shows the schematic diagram of our proposed technique – BRaIn – for software bug localization. We discuss its different steps in the following section.

4.3.1 Document Indexing and Retrieval

Indexing

To detect software bugs using Information Retrieval (IR), the first step is to index the source code documents from a code repository. We chose Elasticsearch [6] for indexing due to its reliability, support for diverse data types, and easy integration with computing systems (e.g., cloud). We collected 45 subject systems from an existing benchmark dataset – Bench4BL [96] – and indexed the source code (Step 1, Fig. 4.2) from 684 buggy versions of these systems. Our idea was to detect a bug in the exact version of the software system stated in the corresponding bug report. During the indexing, we employed Elasticsearch's default analyzer to perform common preprocessing operations (e.g., tokenization, lowercase conversion, and removal of stop

words).

Retrieval of Potentially Buggy Documents using Textual Relevance

To retrieve potentially buggy documents, we use bug reports (i.e., bug title and description) as queries (Step 2, Fig. 4.2). When we pass these queries to Elasticsearch, it preprocesses them using the standard analyzer and returns the top-K (e.g., 50) results through query execution. To narrow down the search results, we also apply additional filters, such as system and version information from each bug report. Without these filters, the retrieved documents could be irrelevant or noisy. This step provides a set of source documents ranked by their textual relevance against a bug report by employing Elasticsearch's default retrieval algorithm, Okapi BM25 [133].

4.3.2 Intelligent Relevance Feedback

Once we have the results from Elasticsearch, we employ advanced prompt methods and Large Language Models (LLM) to determine the relevance between a bug report and each result (i.e., source code). LLMs have shown remarkable capabilities understanding natural language texts and source code [51, 67, 99]. We leverage their capabilities to capture intelligent relevance feedback against a query (a.k.a., bug report). To achieve this, we use prompt engineering, document segmentation, and finally relevance estimation as follows.

Prompt Engineering: *Prompting* is a novel method that instructs the LLMs (e.g., Mistral) to generate meaningful responses without any expensive training [100, 113, 169]. It involves crafting appropriate instructions to guide LLM outputs and make them applicable to different problem-solving tasks [22, 33, 53, 106, 141]. LLMs have been found to be effective with well-designed prompts that are clear, specific, and actionable [100, 138]. Following the insights, we first developed a candidate prompt based on efficient prompt-building guidelines [9, 153]. Our goal was to determine whether a given code segment triggers a reported bug. It instructs the LLM to find the relevance, deliver the output in a JSON format, and act as a rational software engineer, incorporating the contextual information from the bug report and code segment.

To refine our candidate prompt, we employed SAMMO [137], a compile-time

framework that optimizes prompts by exploring various configurations through mutation operations. We configured SAMMO with LLaMA-3 [155] and used a small dataset of 20 bug reports with corresponding buggy code segments (ground truth) to guide our optimization process. SAMMO iteratively generated prompt variants by applying various modification operations to the candidate prompt with LLaMA's assistance. In each iteration, we used LLaMA, bug reports and ground truth code to determine the fitness of each prompt and provide a performance update to SAMMO. Through an extensive search, SAMMO was able to find an optimized version of the prompt. Table 4.2 shows the optimized prompt template, used in the subsequent steps of our technique.

Segmentation: We divide the source code documents from Elasticsearch into smaller segments to determine their relevance using prompting and LLM (Step 3, Fig. 4.2). According to an existing work [102], breaking up texts into smaller segments helps the attention mechanism focus on specific parts, which could be useful for our relevance estimation task. In our work, we adopt a simple method to capture code segments rather than collecting program slices. The slicing methods often create slices that are either too small to capture meaningful context or too large, introducing irrelevant contexts and potentially exceeding the token limits of the LLMs [147]. Therefore, we used a widely adopted library for static analysis – JavaParser [55], to extract code segments such as methods, constructors, interfaces, and enums from a document.

Determining the relevance of code: To determine code relevance, we employ LLaMA [155], Mistral [151], and Qwen [152] models in a zero-shot setting, and provide a bug report and a code segment (e.g., method, constructor) as *context* (Step 4, Fig. 4.2), respecting the token limits of these models (e.g., 8,192). We used the optimized template in Table 4.2 for LLaMA and Qwen, consisting of three key elements: *system, user, assistant*. On the other hand, the *system* element does not apply to Mistral, and thus its prompt template was adapted accordingly. We use Hugging Face's [176] AutoTokenizer to perform model-specific formatting of the prompt and vLLM [90] to parallelize computations across the GPU in batches, increasing throughput. Each employed model against our context provides a response. For example, for the showcase bug report (Table 4.1) and corresponding buggy code (Fig. 4.1), we obtained

the JSON response `{"relevance": "yes"}` from the Mistral. We also found a small number of cases where the outputs are malformed or incomplete JSON. In such instances, we perform string matching within the response (e.g., yes, no) to capture the relevance estimate of the code by the LLM. The relevance estimates of the code segments (collected from the results of Elasticsearch) serve as an intelligent feedback by the LLM to the original query. We coin this as *Intelligent Relevance Feedback (IRF)*.

Table 4.2: Prompt Template for Relevance Feedback

System:
You are a helpful AI software engineer specializing in identifying buggy code segments given a bug report. Analyze the provided bug report and the JAVA code segment to determine if the code segment is responsible for causing the bug described in the bug report. You need to understand the functionality of the code segment and the details of the bug report to determine the relevance of the code segment to the bug report.
There are two possible outputs: 'yes', 'no'.
- 'yes': The code is responsible for the bug described in the bug report.
- 'no': The code is NOT responsible for the bug described in the bug report.
Provide your output in JSON format like this sample: {"relevance": "yes"}.
Act like a rational software engineer and provide output. Avoid emotion and extra text other than JSON.
User:
Analyze the following bug report and code segment:
Bug Report: <BUG REPORT>
Code Segment: <CODE SEGMENT>

Please determine if the code segment is responsible for the bug described in the bug report.
Assistant:

4.3.3 Query Expansion

Using the Intelligent Relevance Feedback (IRF), we expand an original query (Step 6, Fig. 4.2). Unlike the earlier work that relies on pseudo-relevance feedback [68, 85], we choose the source code documents marked as relevant by the LLM for query expansion. In pseudo-relevance feedback, the top few documents retrieved by an IR method are naively considered as relevant and are used for query expansion. On

the other hand, our underlying idea is that documents contextually relevant to the bug reports (i.e., IRF) provide terms that can complement the original query. We adapt an existing work of Rahman et al. [128] to capture appropriate terms from the relevant source documents as follows.

First, we parse each of the source documents retrieved by Elasticsearch and extract class, method, and field signatures from them. These signatures capture the intent of the code, whereas the detailed implementation code could be noisy [128]. We extract the signatures using a lightweight Python library – Javalang [7]. Next, we split camel case tokens from these signatures and turn them into textual phrases by combining their split tokens. We preprocess each of the phrases by filtering out stop words and programming keywords. Then, we construct a term graph $G(V, E)$ by encoding terms as vertices (V) and co-occurring terms as connecting edges (E). Subsequently, we apply the PageRank algorithm [24] in Eq. 4.1 to the term graph to select the influential terms.

$$PR(V_i) = \frac{1-d}{N} + d \sum_{V_j \in M(V_i)} \frac{PR(V_j)}{L(V_j)} \tag{4.1}$$

Here, $PR(V_i)$ represents the PageRank of vertex V_i. The term d denotes the damping factor with a default set to 0.85. N refers to the total number of vertices in the graph, while $M(V_i)$ indicates the set of vertices linked to V_i. Finally, $PR(V_j)$ and $L(V_j)$ represent the PageRank and outbound links of vertex V_j.

The algorithm assigns an initial score to each vertex (V_i) and iteratively updates it, prioritizing the vertices with higher connections. This process repeats until the scores stabilize or the algorithm reaches its maximum iteration limit (e.g., 100). Once the computation is done, we select the top-N (e.g., 10) weighted terms returned by the algorithm [128]. Finally, we expand the original query (i.e., bug report) with these terms, complementing bug reports with contextually relevant terms from source code, leveraging intelligent relevance feedback.

4.3.4 Bug Localization

We leverage our expanded query above and Intelligent Relevance Feedback (IRF) from the LLM to localize the buggy source documents as follows.

Reranking

We determine the relevance between each result from Elasticsearch and our expanded query employing BM25 algorithm, and rerank them according to their relevance (Step 7, Fig. 4.2). This step provides us with ranked results and their BM25 scores. The updated ranks could be useful since the expanded query contains more meaningful terms from the source documents.

Rescoring

We also enhance the ranking of source documents by incorporating IRF from LLM into their scores (Step 8, Fig. 4.2). First, we normalize the BM25 scores of the retrieved documents using a softmax function [58], producing a set of scores that add up to 1. The softmax function amplifies the differences among its input values exponentially, making their difference clearer. Given that the BM25 scores of the documents could have high variance, this allows the softmax function to highlight the textually relevant results. However, since textual relevance might not be sufficient, we also leverage the LLM's feedback against each result document. To incorporate this, we promote the relevant documents and penalize the irrelevant ones, marked by the LLM. This combined approach (Eq. 4.2) incorporates both textual and contextual relevance in the document ranking as follows.

$$score_i = \frac{e^{z_i}}{\sum_{j=1}^{n} e^{z_j}} \cdot r_i, \quad r_i = \begin{cases} 1, & \text{if } i^{\text{th}} \text{ doc is relevant} \\ 0, & \text{otherwise} \end{cases} \tag{4.2}$$

Here, $score_i$ represents the score of the i^{th} document. The term r_i denotes the binary relevance feedback, indicating whether the document is relevant or not (details in Section 4.3.2). The terms z_i and z_j in the softmax function correspond to the BM25 scores of documents i and j.

Finally, we rank the source documents based on their scores for their potential to be buggy (Step 9, Fig. 4.2) and return the top-K (e.g., K=10) documents. Our scoring process aims to bridge the gap between bug reports and source code by incorporating a deeper contextual understanding of the LLM and going beyond their textual and semantic relevance (Table 4.1).

Table 4.3: Dataset

(a) Dataset Summary

Project	Systems	Bug Reports
Spring	25	1,802
Apache	25	1,802
Wildfly	5	806
Commons	8	507
JBoss	1	9
Total	42	4,683

(b) Train-Test Split

Project	Train	Test
Spring	1,429	373
Apache	1,246	313
Wildfly	643	163
Commons	402	105
JBoss	7	2
Total	3,727	956

4.4 Experiments

We curate a dataset of ≈4.7K bug reports from the benchmark dataset Bench4BL and evaluate using three appropriate metrics from the relevant literature — Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K (K=1, 5, 10) [129, 136]. We experiment with three different LLMs and compare our solution –BRaIn– against eight relevant baselines to place our work in the literature. Through our experiments, we answer three research questions as follows:

- $\boldsymbol{RQ}_1$: *(a)* How does BRaIn perform in localizing software bugs? *(b)* Does BRaIn enhance the localization of bugs that require changing multiple documents? *(c)* Can it improve the localization of bugs that are reported poorly?
- $\boldsymbol{RQ}_2$: How do IRF-based query expansion and document ranking contribute to the performance of BRaIn?
- $\boldsymbol{RQ}_3$: Can BRaIn outperform the relevant baseline techniques in bug localization?

4.4.1 Dataset Construction

In our experiment, we used the Bench4BL [96], a comprehensive benchmark dataset that contains 10,017 bug reports from 51 open-source systems, covering a total of 695 software versions. Our initial assessment revealed that bug reports from the older systems lacked crucial versioning information, making them unsuitable for our study. Additionally, we could not accurately link some bug reports to their corresponding

buggy code within the code repositories. Hence, we excluded these bug reports from our dataset. We also found bug reports containing only stack traces without accompanying any textual descriptions of their bugs. We identified those bug reports using regular expressions [129] and excluded them from the dataset. Following these refinement steps, our final dataset comprised 4,683 bug reports from 42 different systems spanning across 684 versions. Table 4.3-a summarizes our curated dataset.

To conduct our experiments and compare with deep learning-based baseline techniques, we used an optimal 80:20 dataset split [65]. This split was done chronologically within each system to ensure that the training set consists of the older 80% of the data, while the test set contains the newest 20% to imitate a real-world scenario. Table 4.3-b provides a summary of our training and test datasets.

4.4.2 Evaluation Metrics

Mean Average Precision (MAP)

Precision@K indicates the precision for each instance of a buggy source document in the ranked list. Average Precision computes the average Precision@K for all buggy documents in relation to a specific search query. Consequently, Mean Average Precision (MAP) is obtained by averaging the Average Precision values across all queries (Q) within a dataset.

$$P_k = \frac{\textit{No. of Relevant Items in Top-k}}{k}$$

$$AP@K = \frac{1}{|D|} \sum_{k=1}^{K} P_k \times B_k$$

$$MAP = \frac{1}{|Q|} \sum_{q=1}^{Q} AP@K_q$$

Here, $AP@K$ computes average precision for top-K results, where P_k is precision at position k and B_k indicates if item k is buggy (1) or not (0). MAP averages this across all queries q in dataset Q, with D being the ground truth documents.

Mean Reciprocal Rank (MRR)

Reciprocal Rank (RR) refers to the rank of the first relevant result retrieved by a technique. It is defined as the reciprocal of the rank of the first relevant source document within the ranked list for each query.

$$RR_q = \frac{1}{\textit{Rank of First Relevant Item}}$$

$$MRR = \frac{1}{|Q|}\sum_{q=1}^{|Q|} RR_q$$

Here, MRR averages the Reciprocal Ranks (RR_q) across all queries q in set Q, where RR_q is the Reciprocal Rank for query q.

HIT@K

HIT@K [136] measures the proportion of queries for which a technique retrieves at least one relevant document among the top-K results. Higher HIT@K values indicate better performance in bug localization techniques.

$$HIT@K = \frac{1}{|\mathcal{Q}|}\sum_{q=1}^{|Q|} \begin{cases} 1, & r_q \in \mathcal{G} \\ 0, & \text{otherwise} \end{cases}$$

Here, r_q returns 1 if query q has a ground truth item in the top-K results (0 otherwise), where Q is the set of all queries.

4.4.3 Selection of LLM

We select three Large Language Models (LLM) to design our techniques and conduct our experiments. We adopt three important criteria to select the models: (a) they should be open-source instruction models, (b) they need to have a quantized version to reduce computational demand, and (c) they should have similar numbers of parameters to allow for fair comparisons. Based on these criteria, we chose LLaMA-3 8B Instruct [155], Mistral v0.3 7B Instruct [151], and Qwen 1.5 7B Chat [152]. They were the top models from instruct category on the Huggingface Open LLM leaderboard [16] during July 2024. We use the 8 bit quantized GPTQ versions of these

models [59] to achieve computational efficiency and leverage vLLM [90] for parallelization. We conducted the experiments on Nvidia V100 GPU-enabled machines with 16 GB vRAM in a cluster computing environment.

Table 4.4: Performance of BRaIn

Techniques	MAP	MRR	HIT@1	HIT@5	HIT@10
Elasticsearch	0.484	0.513	0.413	0.647	0.732
BRaIn (LLaMA)	0.534	0.568	0.470	0.701	0.766
BRaIn (Mistral)	0.537	0.571	0.469	0.709	0.781
BRaIn (Qwen)	0.492	0.523	0.411	0.678	0.755

4.4.4 Evaluating BRaIn

Answering RQ_1 - Performance of BRaIn

We evaluate the performance of *BRaIn* using Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K against top 1, 5, and 10 results. Table 4.4 summarizes our performance details.

From Table 4.4, we see that our proposed technique performs well in detecting the software bugs. BRaIn, powered by Mistral exhibits strong performance, with a Mean Average Precision (MAP) of 0.537. This indicates BRaIn's ability to rank the relevant documents (a.k.a., buggy source documents) higher than the irrelevant ones. Our technique achieves a Mean Reciprocal Rank (MRR) of 0.570 suggesting that the first relevant document is found within the top two positions. BRaIn (Mistral)'s HIT@1 score of 0.469 shows that, for nearly 47% of bug reports, the most relevant document appears at the top position. BRaIn (Mistral) also performs well in HIT@5 and HIT@10, with approximately 71% and 78% of bug reports having at least one relevant buggy document found within the top 5 and top 10 positions, respectively. BRaIn (LLaMA) delivers nearly comparable results to BRaIn (Mistral), trailing by 1.9% in HIT@10. Although BRaIn (Qwen) demonstrates decent performance, it lags behind both BRaIn (Mistral) and BRaIn (LLaMA) in all metrics. It achieves MAP and MRR scores of 0.492 and 0.523, which are about 9.1% lower than BRaIn (Mistral)'s best performance in each metric.

According to our investigation, some bugs trigger changes to a single document

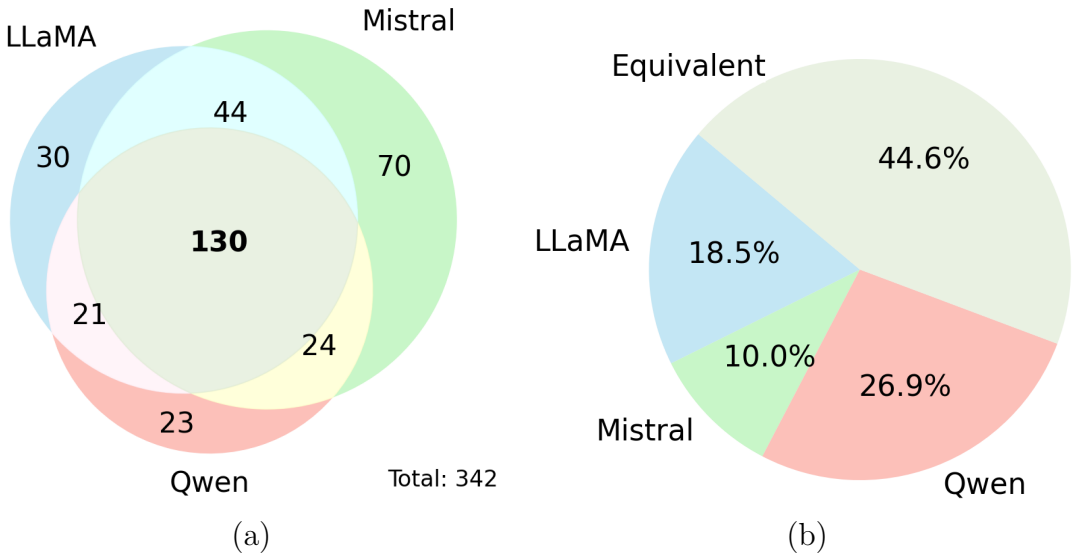
Table 4.5: Performance of BRaIn against multi-document bugs

Changed Documents	Bug Report Count	Elasticsearch (ES)			LLaMA		
		MAP	MRR	HIT@10	MAP	MRR	HIT@10
1	1,949	0.474	0.474	0.690	0.535	0.535	0.726
2	1,436	0.528	0.573	0.767	0.577	0.628	0.801
3	525	0.462	0.518	0.743	0.493	0.554	0.781
4≥	773	0.445	0.501	0.765	0.480	0.551	0.793

Changed Documents	Bug Report Count	Mistral			Qwen		
		MAP	MRR	HIT@10	MAP	MRR	HIT@10
1	1,949	0.542	0.542	0.739	0.483	0.483	0.712
2	1,436	0.565	0.618	0.820	0.529	0.578	0.792
3	525	0.511	0.573	0.789	0.469	0.528	0.758
4≥	773	0.492	0.554	0.811	0.461	0.520	0.794

during bug resolution, whereas others trigger changes to multiple documents. We thus evaluate BRaIn's performance in localizing bugs that warrant changes across multiple source documents. Table 4.5 shows the performance of BRaIn, powered by three different LLMs, in terms of MAP, MRR, and HIT@10. We grouped bugs from our dataset into four categories based on the number of their changed documents: 1, 2, 3, and 4 or more. Our findings show that BRaIn performs strongest when paired with Mistral. For the 1,949 bugs requiring changes to a single document, BRaIn (Mistral) achieves a MAP of 0.542, representing a significant 14.3% improvement over the baseline counterpart. The improvements extend to other metrics, with a 14.3% increase in MRR and 7.1% in HIT@10. BRaIn (Mistral) also excels in resolving bugs that require multiple document changes, achieving improvements of 7.0-10.6% in MAP, 7.9-10.6% in MRR, and 6.0-6.9% in HIT@10. The other variants of BRaIn also outperform the Elasticsearch baseline in localizing bugs that require multiple document changes, achieving improvements of up to 9.2% in MAP, 10.0% in MRR, and 5.1% in HIT@10.

We also investigate how BRaIn performs in localizing bugs where the bug reports could be of low quality (Fig. 4.3-a). According to existing literature [127], low-quality bug reports lack sufficient information and provide queries that cannot retrieve at least one relevant result within their top 10 positions. In our dataset, we identified 1,101 bug reports that fall into this category. Of these, 581 bug reports (i.e., queries) do not



(a) (b)

Figure 4.3: Performance of BRaIn with Low Quality Bug Reports

contain any ground truth within their top 50 results returned by Elasticsearch. These bug reports were not considered, which leaves us with 520 low-quality bug reports for our analysis. Our findings demonstrate BRaIn's promising results even with the low-quality reports. BRaIn (Mistral) emerges as the top performer, successfully localizing 268 bug reports (51.5% of low-quality bug reports) within the top 10 results. BRaIn (LLaMA) and BRaIn (Qwen) follow, identifying 225 and 198 reports, respectively. Notably, all three models identified 130 bugs, with BRaIn (Mistral) uniquely localizing an additional 70 bugs, followed by BRaIn (LLaMA) and BRaIn (Mistral) with 30 and 23 bugs. We also assess BRaIn's capability to detect the first relevant document (a.k.a., buggy source document), where 130 bug reports from above were considered (Fig. 4.3-b). Interestingly, BRaIn (Qwen) outperformed the other variants in this metric, localizing 26.9% of the bugs at the top positions, followed by BRaIn (LLAMA) at 18.5% and BRaIn (Mistral) at 10%. All the findings above suggest BRaIn's ability to analyze, enhance, and localize bug reports, even with low-quality reports.

RQ1 Summary: BRaIn significantly improves bug localization, particularly with Mistral, reaching a high MAP score of 0.537. This performance is due to BRaIn's effective handling of bug reports with up to ≈11% multiple changed documents. Moreover, BRaIn demonstrates an impressive performance with limited information, successfully localizing ≈52% of low-quality bug reports within the top 10 results where the baseline failed.

Table 4.6: Impact of Query Expansion and Scoring

BRaIn Components	BRaIn (LLaMA)		BRaIn (Mistral)		BRaIn (Qwen)	
	MAP	MRR	MAP	MRR	MAP	MRR
Expansion + Reranking	0.534	0.568	0.537	0.571	0.492	0.523
Expansion + No Reranking	0.498	0.568	0.498	0.569	0.496	0.567
No Expansion + Reranking	0.518	0.574	0.520	0.573	0.489	0.529

Answering RQ_2 - Contribution of Query Expansion and Document reranking

BRaIn leverages Intelligent Relevant Feedback (IRF) to expand its original query and rerank the documents. Query expansion can improve bug localization by adding relevant keywords to an original query. Similarly, incorporating contextual understanding into document scoring can help go beyond just textual and semantic matching during document ranking. We examine the contribution of these two components (Table 4.6) to BRaIn's performance as follows.

To determine the impact of query expansion in isolation, we evaluated BRaIn's performance without the reranking component (Table 4.6). Interestingly, all BRaIn variants achieved similar MAP scores of around 0.49, falling short of optimal performance. For example, BRaIn (LLaMA) and BRaIn (Mistral) saw MAP scores decrease by 7.2% and 7.8%, respectively, while their MRR scores remained relatively stable. BRaIn (Qwen), on the other hand, showed a 7.8% increase in MRR alongside a slight improvement in MAP. However, all BRaIn variants outperformed the Elasticsearch baseline by 10.5-10.9% in MRR and 2.4-2.9% in MAP.

In contrast, when reranking was applied isolately, MAP scores for BRaIn (LLaMA), BRaIn (Mistral), and BRaIn (Qwen) dropped by 3.0%, 3.2%, and 6.1%, respectively, while their MRR scores remained consistent. Despite these decreases, all variants showed improvements of 3.1-11% in MRR and 1.0-7.1% in MAP over the Elasticsearch baseline.

These findings show how Intelligent Relevance Feedback improves query expansion and reranking. However, they also underscore the importance of the synergy between these components for optimal performance.

RQ2 Summary: Query expansion and reranking individually decrease MAP and MRR by 3.0-7.8%, yet both improve bug localization performance over the baseline by 2.4-11% in these metrics. Their individual results highlight the contribution of Intelligent Relevance Feedback and underscore the importance of a synergistic combination in BRaIn.

Table 4.7: Comparison Between BRaIn and Baseline Techniques

(a) Comparison with Non-ML Baselines

Metrics	Traditional IR			Relevance Feedback		
	Elasticsearch	BLUiR	Blizzard	Rocchio	Sysman-SCP	BRaIn
MAP	0.484	0.450	0.506	0.489	0.472	0.537
MRR	0.513	0.471	0.536	0.558	0.541	0.571
HIT@10	0.732	0.696	0.758	0.765	0.753	0.781

(b) Comparison with ML Baselines

Metrics	Machine Learning			
	DNNLOC	RLocator	NextBug	BRaIn
MAP	0.283	0.488	0.469	0.531
MRR	0.296	0.561	0.540	0.564
HIT@10	0.518	0.735	0.743	0.771

Answering RQ_3 - Comparison with Basline Techniques

To place our work in the literature, we compare BRaIn with relevant baseline techniques in terms of their MAP, MRR, and HIT@10. Given our methodology, we choose two types of baseline techniques – IR methods [68, 129, 136] and deep learning based

methods [29, 92, 180]. For comparison with baselines, we use BRaIn (Mistral) in our experiments since it is the best-performing variant of BRaIn.

To replicate the traditional IR-based bug localization with Elasticsearch baseline, we index all source documents of a repository and use bug reports (title + description) as queries. These queries are executed with Elasticsearch [6], which retrieves relevant documents using the BM25 algorithm [133] and Boolean queries, with default parameters for k and b. Other traditional approaches from literature– BLUiR [136] and Blizzard [129]– use structured information from bug reports and source code for bug localization. BLUiR calculates suspiciousness scores using class names, method names, variable names, comments, and bug report elements (title, description), combining multiple searches into an overall score. Blizzard categorizes bug reports into three types and constructs text graphs from these reports to generate queries and retrieve relevant buggy source documents. For both approaches, we employ Apache Lucene [57] for retrieval. We replicated these methods by adapting them from Bench4BL repository [96] and Blizzard's replication package [3] from the authors. We compare BRaIn with these established IR-based techniques to validate our technique and place it in the literature.

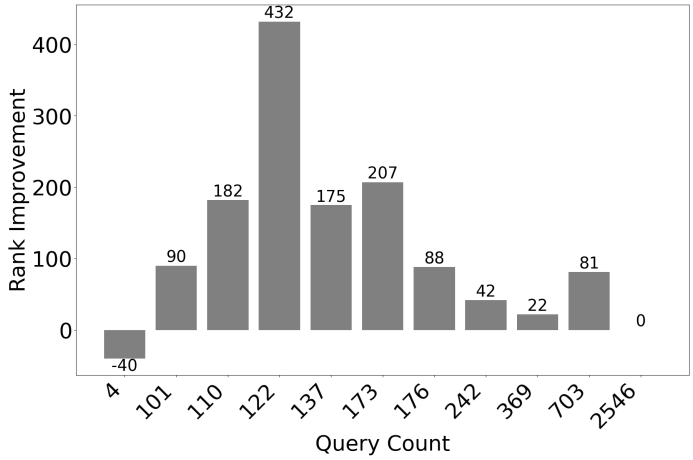
Relevance feedback-based techniques like Rocchio [42] and the Spatial Code Proximity (SCP) model [68] aim to enhance bug localization by refining queries based on the results of initial searches. Rocchio is a widely-used relevance feedback technique for information retrieval [42]. We leverage relevance feedback to reformulate queries and Apache Lucene to execute the queries and retrieve the documents. Our reformulated queries were optimized using α, β, and γ parameters [42]. Similarly, we implemented Sisman et al.'s SCP model [68] to reformulate queries based on term proximity within source code. It prioritizes terms that frequently co-occur within the same method or class, using the best parameters w, x, and y suggested by the authors. We compare these techniques to our approach to highlight the importance of contextual understanding during relevance feedback of search queries.

Since Machine Learning (ML) techniques can capture complex patterns in data using non-linear relationships, we compare BRaIn against three ML-based techniques– DNNLOC [92], NextBug [180], and RLocator [29]. DNNLOC combines multiple

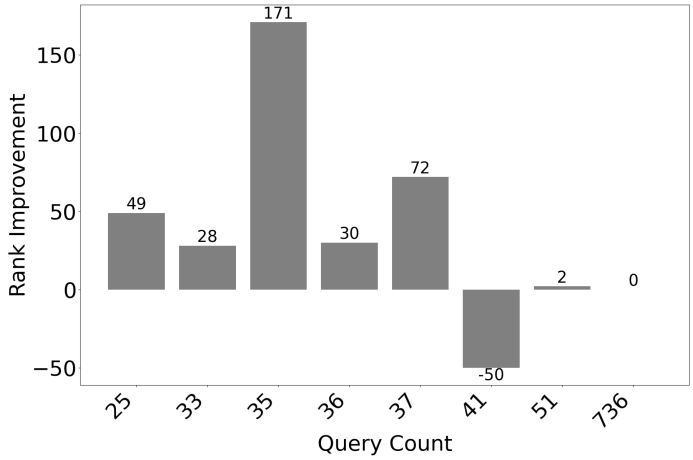
features— rVSM score [84] for bug report-source code similarity, class name similarity, collaborative filtering, and bug report recency and frequency—and uses a neural network to predict suspiciousness scores to rank documents. NextBug employs Word2Vec [107] embeddings to capture semantic relations between bug reports and source code and thus to localize the buggy documents. In our experiments, we substituted Word2Vec with CodeT5 embeddings [168] to capture more nuanced text-level semantic associations, as opposed to token-level. RLocator is a recent deep-learning technique that employs a reinforcement learning model, framed as a Markov Decision Process, to optimize ranking of buggy documents. We replicated DNNLOC and NextBug by following the respective authors' approaches and replicated RLocator using the authors' provided replication package on Zenodo [12]. To ensure consistency with the authors' specifications, we replicated the methods using cross-validation.

Table 4.7-a summarizes our comparison details with the baseline techniques. Among traditional IR-based approaches, Blizzard achieves a MAP score of 0.506, while Elasticsearch (ES) and BLUiR scores are 0.484 and 0.450. BRaIn outperforms them with a MAP of 0.537, achieving a maximum improvement of 19.3% over these techniques. Similarly, BRaIn achieves notable gains in MRR and HIT@10, with increases of up to 17.5% and 12.2%. Among the IR based approaches that leverage relevance feedback, Rocchio's algorithm achieves a MAP of 0.489, slightly above the baseline, while Sysman-SCP falls short by 2.5%. BRaIn again leads here with the improvements in MAP, MRR, and HIT@10 of 13.8%, 5.5%, and 3.7%, respectively. These results underscore the advantages of Intelligent Relevance Feedback, which uses contextual understanding over traditional techniques based on textual relevance for bug localization.

As shown in 4.7-b, BRaIn also outperforms the machine learning techniques that require training. We evaluated both BRaIn and the baseline techniques on the test set only to ensure a fair comparison. It should be noted that old bug reports and their corresponding code were used for training and the recent bugs and their corresponding code were used for testing. DNNLOC performs significantly lower with a MAP of 0.283, 87.6% lower than BRaIn's optimal score of 0.531. In comparison, RLocator and NextBug achieve MAP scores of 0.488 and 0.469, with BRaIn outperforming them by 8.8% and 13.2%, respectively. Similar improvements are observed for other



(a) Full Dataset (4,683 Bug Beports)



(b) Low Quality Bug Reports (1,101 Bug Reports)

Figure 4.4: Rank Improvement: BRaIn vs Blizzard

metrics, with BRaIn showing 4.4-89.5% improvements in MRR and 3.7-48.8% in HIT@10. Such performance underscores the superiority of BRaIn's performance with Intelligent Relevance Feedback (IRF) compared to baseline techniques.

Finding the first buggy document is very important during bug localization [68]. We further investigate how BRaIn performs in such a case. We chose Blizzard for this investigation as it is the best-performing model against BRaIn in our experiments. Fig. 4.4-a compares BRaIn and Blizzard by analyzing the difference in ranks for each query within the top 10 results. A positive value indicates that BRaIn found the first ground truth at a better rank than Blizzard, while a negative value suggests the opposite. For 122 bug reports, the large difference of 432 indicates that BRaIn

Table 4.8: Statistical Test: BRaIn vs. Blizzard

Evaluation Point	p-value	Effect Size (Cliff's δ)
Top-1	0.0023 **	Medium (0.41)
Top-5	0.0015 **	Large (0.66)
Top-10	0.0008 ***	Large (0.82)

*=statistical significance

identified the first buggy documents more often than Blizzard. In contrast, Blizzard only outperformed BRaIn for 4 bug reports. For the set of 2,546 bug reports, there are two distinct possibilities: Either (a) the rank difference between the two techniques was 0, or (b) neither technique identified a buggy document within the top-10 results. We extend our analysis to the 1,101 low-quality bug reports discussed in RQ_1. Here we also see BRaIn's dominance in rank improvement over Blizzard for 80.34% low-quality bug reports (Fig. 4.4-b). These results strongly indicate the superiority of our approach. To further validate our findings against Blizzard, we conducted non-parametric statistical tests –Mann-Whitney Wilcoxon and Cliff's δ [17]– and compared BRaIn to Blizzard in identifying the first buggy document on the entire dataset. The tests in Table 4.8 show $p-values < 0.05$ for the top-1, 5, and 10 results, with medium to large effect sizes (i.e., $0.41 \leq \delta \leq 0.82$). Thus, it confirms BRaIn's consistent ability to detect the first buggy document more effectively than its closest competitor.

RQ3 Summary: BRaIn outperforms traditional, relevance feedback-based, and ML-based baseline techniques, achieving an 87.6% improvement in MAP. This demonstrates BRaIn's ability to localize relevant buggy documents at the top positions using Intelligent Relevance Feedback (IRF), surpassing relevance feedback-based techniques by 3.7–13.8% across various metrics. Statistical significance tests further confirm BRaIn's superiority.

4.5 Related Work

4.5.1 IR based Bug Localization

Bug localization techniques can broadly be classified into two main groups: spectra-based and information retrieval (IR)-based approaches [163]. Spectra-based methods use program execution traces and test methods to localize bugs, making them complex and expensive [110, 162]. In contrast, IR-based techniques rely on textual overlap between bug reports and source code to localize the bugs.

Traditional IR-based methods bug localization leveraging the vector space model (VSM) [95], have been enhanced by integrating additional contexts, such as bug report history, code modifications, and version history [135, 143, 172]. For instance, Saha et al. [136] leverage bug report and source code structures, capture eight components from the code and bug reports, and perform eight pairwise searches using a sophisticated retrieval technique, Indri [148]. On the other hand, BugLocator [84] combines a modified VSM (rVSM [84]) score with previous bug fix history to improve bug localization. AmaLgam [164] integrates BLUiR, BugLocator, and version history to better detect buggy documents. AmaLgam+ [165] further incorporates stack traces and bug reporter history, refining bug localization across five ranking components. While advanced, computationally expensive methodologies such as LSI or LDA [84,121] are available, their bug localization effectiveness is similar to that of more basic methods [96].

In our work, we use a textual similarity-based retrieval with BM25 in Elasticsearch. However, it was complemented by contextual understanding of bugs and Intelligent Relevance Feedback, leveraging the capabilities of LLMs.

4.5.2 Query Reformulation

Poorly constructed queries from software bug reports can significantly hinder IR-based bug localization [109, 127]. To tackle this issue, researchers have developed query reformulation techniques that can improve search queries by incorporating better terms or eliminating unnecessary ones. For instance, Refoqus [68] uses query characteristics and machine learning to recommend strategies like query reduction or expansion for a given query. Graph-based methods analyze semantic and syntactic

relationships within bug reports to identify key terms. Rahman and Roy [129] create text graphs to collect important terms based on three different bug types to improve query reformulation. The authors later demonstrate generating optimal queries in bug reports by employing Genetic algorithms [127], which iteratively refine queries based on search results. Gay et al. [68] used Rocchio's algorithm to improve queries with developer feedback, while Sisman et al. [144] expanded queries by selecting terms from the top-ranked documents using Spatial Code Proximity (SCP), without using any explicit relevance feedback.

These approaches rely on statistical properties or co-occurrence relations to reformulate an original query without meaningful knowledge of source code or bug reports. Our approach digs deeper to select relevant source code by contextually understanding bug reports to formulate queries for better bug resolution.

4.5.3 Deep Learning for Bug Localization

Recent advancements in deep learning have encouraged its applications in bug localization. DNNLOC [92], a seminal work on this topic, identifies buggy documents by learning from multiple text-based features and metadata (e.g., rVSM score [84], class name similarity, bug report recency). However, its reliance on features like bug fixing recency can limit its application [84]. A recent technique, FBL-BERT [37] uses a BERT-based model, ColBERT [82], for document scoring with late interaction. However, it relies on changesets for resolution, which are difficult to track in large, fast-changing projects. Another recent technique, RLocator [29], optimizes ranking metrics in the bug localization process. It formulates bug localization as a Markov Decision Process (MDP) and employs reinforcement learning (RL) to localize bugs. Other approaches like TRANP-CNN [76] and CooBa [185] use Convolutional Neural Networks (CNN) and Graph Convolutional Networks (GCN) to improve cross-project bug localization. However, these methods may struggle with scalability when handling large volumes of documents.

In contrast, our technique ensures scalability by using a limited set of documents retrieved via Elasticsearch [6]. By combining efficient IR-based filtering with the contextual depth of language models, BRaIn accurately ranks buggy documents, improving bug localization.

4.6 Threats to Validity

Threats to internal validity concern experimental errors and biases. Replication of existing baselines poses such a threat. We mitigated this by using replication packages from the original authors (Blizzard [3], RLocator [12]) and from Bench4BL [96] (BLUiR [136]). Due to Indri's obsolescence, we substituted it with Lucene in the BLUiR replication. On the other hand, we replicated DNNLOC [92], NextBug [180], and Sysman-SCP [68] adhering strictly to the original authors' settings and parameters. To minimize bias, we tested on two distinct datasets and found only a negligible difference compared to baseline performances.

Threats to external validity relate to generalizability. While BRaIn was evaluated only on Java code, the underlying models (e.g., LLaMa [155]) are designed to adapt to various programming languages, potentially mitigating this limitation.

Threats to construct validity concern the appropriateness of our evaluation metrics. We employed widely used metrics such as Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K, which were commonly used in existing literature on bug localization [129, 131, 136] and Information Retrieval studies [71]. Therefore, this choice of metrics minimizes threats to construct validity.

Finally, we used 20 bug reports from our dataset to optimize prompts with LLaMA. Since they are part of our experimental dataset, it could introduce bias. We repeated a limited experiment and found similar performance to the reported ones in the paper. Thus, any relevant threat of bias is minimal.

4.7 Summary

To summarize, we incorporate Intelligent Relevance Feedback into IR methods and leverage deep contextual understanding of large language models (LLMs) to support bug localization. Our evaluation using three performance metrics—MAP, MRR, and HIT@K—demonstrated significant improvements over baseline methods, achieving the margins of 87.6%, 89.5%, and 48.8%, respectively. Additionally, BRaIn effectively localized bugs involving multiple source documents, with improvements ranging from 6.0% to 10.6% across various metrics. Notably, it also demonstrated the ability to handle low-quality bug reports, outperforming the baseline textual retrieval by

51.5%. These results highlight the potential of Intelligent Relevance Feedback (IRF) in significantly advancing bug localization efforts.

Chapter 5

Coclusion and Future Works

5.1 Conclusion

Software bugs consume valuable development effort and time, and cause significant financial losses. Their effective, timely resolution has been a major focus, with researchers working on solutions for decades. Unfortunately, existing tools often fail to capture the contextual variability of bug reports or source code and struggle to localize bugs efficiently. In this RAD report, we conduct two complementary studies and propose novel methods that leverage contextual understanding from Large Language Models to support bug localization. Our first work incorporates the reasoning capabilities of Large Language Models (LLMs) into traditional Information Retrieval (IR) methods, refines search queries, and reranks search results to improve bug localization. We evaluate our technique – IQLoc – using three widely accepted metrics: Mean Average Precision (MAP), Mean Reciprocal Rank (MRR), and HIT@K. Across all measures, IQLoc consistently outperformed existing techniques, with improvements of up to 58.52% and 60.59% in MAP, 61.49% and 64.58% in MRR, and 69.88% and 100.90% in HIT@K for test bug reports with random and time-wise splits, respectively. Our second work employs novel Intelligent Relevance Feedback (IRF) into IR methods to support bug localization. Using the contextual understanding of LLM, our technique – BRaIn – captures intelligent feedback against its queries and uses the enhanced queries in localizing bugs. We evaluate BRaIn using three established metrics and observe improvements of 87.6%, 89.5%, and 48.8% in MAP, MRR, and HIT@K, respectively. Additionally, BRaIn successfully localizes approximately 52% of bugs that baseline techniques fail to identify, demonstrating its benefits and strong potential for localizing software bugs.

5.2 Future Work

Our work has inspired several directions for future work. In the following section, we outline several of those key directions.

5.2.1 Impact of Large Language Models on Understanding Source Code

LLMs have demonstrated great potential for effectively understanding both text and code. However, they are far from perfect, and their existing capabilities also might not have been leveraged well. Thus, we would like to explore the following problems.

- Structural representations for code like Data Flow Graphs (DFGs), Control Flow Graphs (CFGs), and Program Dependency Graphs (PDGs) can capture important relationships between variables, program flow, and code dependencies [50, 116, 183]. Incorporating these structures—along with graph-based neural models such as Graph Attention Networks (GAT) [160] could help LLMs better understand how different parts of a program interact to better capture the semantics and intents of the code.

- Developers use various (e.g., camelCase, snake_case [38, 39]) and often inconsistent naming conventions [14, 43, 144] for identifiers in source code. LLMs tokenize texts using techniques like Byte Pair Encoding (BPE) or SentencePiece [89, 139], which split identifiers in different ways. For example, Googles T5 [125] language model splits source code identifer `sendHTTPRequest` into five tokens (i.e., `send`, `HT`, `TP`, `Re`, and `quest`). While a developer can infer the purpose of the identifier from the name, this type of splitting can hinder the semantic understanding of a code language model. The effects of such subword tokenization on bug localization remain underexplored and deserve further investigation.

In our first study, we fine-tuned a CodeBERT model for bug localization. As a next step, we aim to explore how semantic understanding of the large language models can be influenced with structural representation or tokenization of code during bug localization.

5.2.2 Agentic Bug Localization

With the advancement of LLMs, agent-based computing [179] has been recognized as a viable choice for solving complex tasks in an adaptive and efficient manner, allowing autonomous decision-making. We envision a multi-agent system for bug localization, where specialized agents with distinct capabilities will collaborate to address different aspects of the problem.

- In our first study, we observed that bug reports written purely in natural language struggle in localizing bugs. However, in our second study, we found that such reports improved significantly when relevance feedback from the language model was incorporated. This suggests the importance of designing agents that specialize in handling different types of bug reports. For instance, one agent could handle natural language-heavy bug reports, while another focuses on ones with stack traces or program artifacts. Building on Blizzard [129], a traditional approach addressing different types of bug reports, we aim to explore agent-based solution for bug localization as part of future work.

- Bug reports often contain acronyms, abbreviations, or unrelated texts that are not helpful for localization [112, 127]. An agent could be responsible for identifying and filtering out irrelevant texts while expanding common acronyms and abbreviations to their full forms. Previous work has shown promising results in explaining and expanding terms in duplicate bug report detection [112]. Incorporating such techniques would improve the clarity and effectiveness of the subsequent queries generated from the report, ensuring that the system better understands the report's intent.

- Bug reports are often ambiguous [127], making bug localization and resolution more challenging. To address this, a separate agent could assess the ambiguity of certain bug reports and generate multiple self-queries [48,104] and generative queries [166]. This would allow bug localization tools to explore different aspects of bug reports. Other agents could then collaboratively refine these focused queries to enhance localization performance.

- To adapt across different software domains and projects, one or more agents

could be designed to learn from available software documentation (e.g., API references, READMEs) and, from human feedback (e.g., developer input, validation of retrieved results). This would enable the system to understand project-specific terminology, structures, and conventions for better bug localization.

Bibliography

[1] Beautiful soup documentation. https://beautiful-soup-4.readthedocs.io/en/latest/.

[2] Blizzard replication package. https://github.com/masud-technope/BLIZZARD.

[3] Blizzard replication package. https://github.com/masud-technope/BLIZZARD.

[4] The cost of poor quality software. https://www.synopsys.com/software-integrity/resources/analyst-reports/cost-poor-quality-software.html.

[5] diff-tool — pypi.org. https://pypi.org/project/diff-tool/.

[6] Elasticsearch. https://www.elastic.co/guide/en/elasticsearch/reference/current/.

[7] Javalang: Pure python java parser and tools. https://github.com/c2nes/javalang.

[8] Mariner 1 — science.nasa.gov. https://science.nasa.gov/mission/mariner-1/.

[9] Prompting: how to guides.

[10] Rank-BM25. https://pypi.org/project/rank-bm25/. Accessed: May 2024.

[11] Rlocator: Reinforcement learning for bug localization, May 2024.

[12] Rlocator replication. https://doi.org/10.5281/zenodo.11265302, July 2024.

[13] Muhammad Bello Aliyu. Efficiency of boolean search strings for information retrieval. *American Journal of Engineering Research (AJER)*, 6(11):216–222, 2017.

[14] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4):1–37, 2018.

[15] Felipe Almeida and Geraldo Xexéo. Word embeddings: A survey. *arXiv preprint arXiv:1901.09069*, 2019.

[16] First Author and Second Author. Open-llm-leaderboard: From multi-choice to open-style questions for large language models. *arXiv preprint arXiv:2406.07545*, 2024.

[17] Tyler Barnes, Scott C. Moore, and Katerine Osatuke. *Testing Significance Tests: A Simulation with Cliff's Delta, t-tests, and Mann-Whitney U*. National Center for Organizational Development, Department of Veteran Affairs, 2018.

[18] Tyler Barnes, Scott C Moore, and Katerine Osatuke. Testing significance tests: A simulation with cliff's delta, t-tests, and mann-whitney u. *National Center for Organizational Development, Department of Veteran Affairs*, 2018.

[19] Yoshua Bengio, Réjean Ducharme, Pascal Vincent, and Christian Jauvin. A neural probabilistic language model. *Journal of Machine Learning Research*, 3:1137–1155, 2003.

[20] Kamil Bennani-Smires, Claudiu Musat, Andreea Hossmann, Michael Baeriswyl, and Martin Jaggi. Simple unsupervised keyphrase extraction using sentence embeddings. *arXiv preprint arXiv:1801.04470*, 2018.

[21] David B Bracewell, Fuji Ren, and Shingo Kuriowa. Multilingual single document keyword extraction for information retrieval. In *2005 international conference on natural language processing and knowledge engineering*, pages 517–522. IEEE, 2005.

[22] Stephen Brade, Bryan Wang, Mauricio Sousa, Sageev Oore, and Tovi Grossman. Promptify: Text-to-image generation through interactive prompt exploration with large language models. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology*, pages 1–14, 2023.

[23] Gianni Brauwers and Flavius Frasincar. A general survey on attention mechanisms in deep learning. *IEEE Transactions on Knowledge and Data Engineering*, 35(4):3279–3298, 2021.

[24] Sergey Brin and Lawrence Page. The anatomy of a large-scale hypertextual web search engine. *Computer networks and ISDN systems*, 30(1-7):107–117, 1998.

[25] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 229, 2013.

[26] Junming Cao, Shouliang Yang, Wenhui Jiang, Hushuang Zeng, Beijun Shen, and Hao Zhong. Bugpecker: Locating faulty methods with deep learning on revision graphs. In *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*, pages 1214–1218, 2020.

[27] Jaime Carbonell and Jade Goldstein. The use of mmr, diversity-based reranking for reordering documents and producing summaries. In *Proceedings of the 21st annual international ACM SIGIR conference on Research and development in information retrieval*, pages 335–336, 1998.

[28] Laura Carnevali. Evaluation Measures in Information Retrieval. https://www.pinecone.io/learn/offline-evaluation/.

[29] Partha Chakraborty, Mahmoud Alfadel, and Meiyappan Nagappan. Rlocator: Reinforcement learning for bug localization. *IEEE Transactions on Software Engineering*, 2024.

[30] Oscar Chaparro, Juan Manuel Florez, and Andrian Marcus. Using observed behavior to reformulate queries during text retrieval-based bug localization. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 376–387. IEEE, 2017.

[31] Oscar Chaparro and Andrian Marcus. On the reduction of verbose queries in text retrieval based software maintenance. In *Proceedings of the 38th International Conference on Software Engineering Companion*, pages 716–718, 2016.

[32] An Ran Chen, Tse-Hsun Chen, and Shaowei Wang. Pathidea: Improving information retrieval-based bug localization by re-constructing execution paths using logs. *IEEE Transactions on Software Engineering*, 48(8):2905–2919, 2021.

[33] Banghao Chen, Zhaofeng Zhang, Nicolas Langrené, and Shengxin Zhu. Unleashing the potential of prompt engineering in large language models: a comprehensive review. *arXiv preprint arXiv:2310.14735*, 2023.

[34] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. *arXiv preprint arXiv:1603.02754*, 2016.

[35] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[36] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

[37] Agnieszka Ciborowska and Kostadin Damevski. Fast changeset-based bug localization with bert. In *Proceedings of the 44th International Conference on Software Engineering*, pages 946–957, 2022.

[38] Wikipedia contributors. Camel case. https://en.wikipedia.org/wiki/Camel_case, 2023. Accessed: 2024-04-03.

[39] Wikipedia contributors. Snake case. https://en.wikipedia.org/wiki/Snake_case, 2023. Accessed: 2024-04-03.

[40] Atlassian Corporation. *Jira REST API Documentation*. Atlassian, 2025. Accessed: 2025-04-06.

[41] Forbes Technology Council. Costly code: The price of software errors, 2023. Accessed: 2024-08-04.

[42] Ronan Cummins, Natural Language, and Information Processing (NLIP) Group. Lecture 7: Relevance feedback and query expansion, 2017.

[43] Florian Deissenboeck and Markus Pizka. Concise and consistent naming. *Software Quality Journal*, 14:261–282, 2006.

[44] Hervé Déjean, Stéphane Clinchant, and Thibault Formal. A thorough comparison of cross-encoders and llms for reranking splade. *arXiv preprint arXiv:2403.10407*, 2024.

[45] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

[46] DevOps. Survey: Fixing bugs stealing time from development, 2024. Accessed: 2024-08-04.

[47] Yangruibo Ding, Jinjun Peng, Marcus J Min, Gail Kaiser, Junfeng Yang, and Baishakhi Ray. Semcoder: Training code language models with comprehensive semantics. *arXiv preprint arXiv:2406.01006*, 2024.

[48] LangChain Documentation. How to: Self-query with langchain, 2025. Accessed: 2025-04-03.

[49] Paul Dourish. What we talk about when we talk about context. *Personal and ubiquitous computing*, 8:19–30, 2004.

[50] Yali Du, Ying Li, Yi-Fan Ma, and Ming Li. Capturing the context-aware code change via dynamic control flow graph for commit message generation. *Machine Learning*, 114(4):94, 2025.

[51] Yassir Fathullah, Chunyang Wu, Egor Lakomkin, Junteng Jia, Yuan Shangguan, Ke Li, Jinxi Guo, Wenhan Xiong, Jay Mahadeokar, Ozlem Kalinli, et al. Prompting large language models with speech recognition abilities. In *ICASSP 2024-2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 13351–13355. IEEE, 2024.

[52] Federal Communications Commission. February 22, 2024 AT&T Mobility Network Outage: Report and Findings. Technical report, Federal Communications Commission, 2024.

[53] Sidong Feng and Chunyang Chen. Prompting is all you need: Automated android bug replay with large language models. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*, pages 1–13, 2024.

[54] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[55] Christoph Fischer. Javaparser, 2019.

[56] Consortium for IT Software Quality. Cpsq 2020 report. Technical report, IT-CISQ, 2020. Accessed: 2024-08-04.

[57] The Apache Software Foundation. Apache lucene. 2021.

[58] Michael Franke and Judith Degen. The softmax function: Properties, motivation, and interpretation, 2023.

[59] Elias Frantar, Saleh Ashkboos, Torsten Hoefler, and Dan Alistarh. Gptq: Accurate post-training quantization for generative pre-trained transformers. *arXiv preprint arXiv:2210.17323*, 2022.

[60] George W. Furnas, Thomas K. Landauer, Louis M. Gomez, and Susan T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30(11):964–971, 1987.

[61] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *2009 IEEE International Conference on Software Maintenance*, pages 351–360, 2009.

[62] Gregory Gay, Sonia Haiduc, Andrian Marcus, and Tim Menzies. On the use of relevance feedback in ir-based concept location. In *2009 IEEE international conference on software maintenance*, pages 351–360. IEEE, 2009.

[63] Benyamin Ghojogh and Ali Ghodsi. Recurrent neural networks and long short-term memory networks: Tutorial and survey, 2023.

[64] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation. *International Journal of Intelligent Technologies and Applied Statistics*, 11(2):105–111, 2018.

[65] Afshin Gholamy, Vladik Kreinovich, and Olga Kosheleva. Why 70/30 or 80/20 relation between training and testing sets: A pedagogical explanation. *Int. J. Intell. Technol. Appl. Stat*, 11(2):105–111, 2018.

[66] Maarten Grootendorst. Keybert: Minimal keyword extraction with bert, 2020. Accessed: 2025-03-12.

[67] Qiuhan Gu. Llm-based code generation method for golang compiler testing. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 2201–2203, 2023.

[68] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 842–851. IEEE, 2013.

[69] Sonia Haiduc, Gabriele Bavota, Andrian Marcus, Rocco Oliveto, Andrea De Lucia, and Tim Menzies. Automatic query reformulations for text retrieval in software engineering. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 842–851. IEEE, 2013.

[70] Jiawei Han, Micheline Kamber, and Jian Pei. 2 - getting to know your data. In Jiawei Han, Micheline Kamber, and Jian Pei, editors, *Data Mining (Third Edition)*, The Morgan Kaufmann Series in Data Management Systems, pages 39–82. Morgan Kaufmann, Boston, third edition edition, 2012.

[71] Donna Harman. *Information retrieval evaluation*. Morgan & Claypool Publishers, 2011.

[72] Zellig S. Harris. Distributional structure. *Word*, 10(2-3):146–162, 1954.

[73] Matthew Hennessy. *The semantics of programming languages: an elementary introduction using structural operational semantics*. John Wiley & Sons, Inc., 1990.

[74] Po-Sen Huang, Xiaodong He, Jianfeng Gao, Li Deng, Alex Acero, and Larry Heck. Learning deep structured semantic models for web search using clickthrough data. CIKM '13, page 2333–2338, New York, NY, USA, 2013. Association for Computing Machinery.

[75] Samuel Humeau, Kurt Shuster, Marie-Anne Lachaux, and Jason Weston. Poly-encoders: Transformer architectures and pre-training strategies for fast and accurate multi-sentence scoring. *arXiv preprint arXiv:1905.01969*, 2019.

[76] Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *IJCAI*, pages 1909–1915, 2017.

[77] IEEE. Ieee standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, pages 1–84, 1990.

[78] Jones, E., Oliphant, T., Peterson, P., et al. Scipy: Open source scientific tools for python. http://www.scipy.org/, 2001.

[79] V Roshan Joseph. Optimal ratio for data splitting. *Statistical Analysis and Data Mining: The ASA Data Science Journal*, 15(4):531–538, 2022.

[80] Leslie Pack Kaelbling, Michael L Littman, and Andrew W Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[81] Anjan Karmakar and Romain Robbes. What do pre-trained code models know about code?, 2021.

[82] Omar Khattab and Matei Zaharia. Colbert: Efficient and effective passage search via contextualized late interaction over bert. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*, pages 39–48, 2020.

[83] Deniz Kılınç, Fatih Yücalar, Emin Borandağ, and Ersin Aslan. Multi-level reranking approach for bug localization. *Expert Systems*, 33(3):286–294, 2016.

[84] Dongsun Kim, Yida Tao, Sunghun Kim, and Andreas Zeller. Where should we fix this bug? a two-phase recommendation model. *IEEE transactions on software Engineering*, 39(11):1597–1610, 2013.

[85] Misoo Kim and Eunseok Lee. A novel approach to automatic query reformulation for ir-based bug localization. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, SAC '19, page 1752–1759, New York, NY, USA, 2019. Association for Computing Machinery.

[86] Misoo Kim and Eunseok Lee. A novel approach to automatic query reformulation for ir-based bug localization. In *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing*, pages 1752–1759, 2019.

[87] Pavneet Singh Kochhar, Yuan Tian, and David Lo. Potential biases in bug localization: Do they matter? In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 803–814, 2014.

[88] Vijay R Konda and John N Tsitsiklis. Actor-critic algorithms. *Advances in Neural Information Processing Systems*, pages 1008–1014, 2000.

[89] Taku Kudo and John Richardson. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 66–71, 2018.

[90] Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph E. Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In *Proceedings of the ACM SIGOPS 29th Symposium on Operating Systems Principles*, 2023.

[91] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229. IEEE, 2017.

[92] An Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *2017 IEEE/ACM 25th International Conference on Program Comprehension (ICPC)*, pages 218–229, 2017.

[93] Annu Lambora, Kunal Gupta, and Kriti Chopra. Genetic algorithm-a literature review. In *2019 international conference on machine learning, big data, cloud and parallel computing (COMITCon)*, pages 380–384. IEEE, 2019.

[94] Tien-Duy B Le, Richard J Oentaryo, and David Lo. Information retrieval and spectrum based bug localization: Better together. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 579–590, 2015.

[95] D.L. Lee, Huei Chuang, and K. Seamons. Document ranking and the vector-space model. *IEEE Software*, 14(2):67–75, 1997.

[96] Jaekwon Lee, Dongsun Kim, Tegawendé F Bissyandé, Woosung Jung, and Yves Le Traon. Bench4bl: reproducibility study on the performance of ir-based bug localization. In *Proceedings of the 27th ACM SIGSOFT international symposium on software testing and analysis*, pages 61–72, 2018.

[97] Sung-Jick Lee and Han-Joon Kim. Keyword extraction from news corpus using modified tf-idf. *The Journal of Society for e-Business Studies*, 14(4):59–73, 2009.

[98] Canjia Li, Yingfei Sun, Ben He, Le Wang, Kai Hui, Andrew Yates, Le Sun, and Jungang Xu. Nprf: A neural pseudo relevance feedback framework for ad-hoc information retrieval, January 2018.

[99] Long Lian, Boyi Li, Adam Yala, and Trevor Darrell. Llm-grounded diffusion: Enhancing prompt understanding of text-to-image diffusion models with large language models. *arXiv preprint arXiv:2305.13655*, 2023.

[100] Pengfei Liu, Weizhe Yuan, Jinlan Fu, Zhengbao Jiang, Hiroaki Hayashi, and Graham Neubig. Pre-train, prompt, and predict: A systematic survey of

prompting methods in natural language processing. *ACM Computing Surveys*, 55(9):1–35, 2023.

[101] Yiwei Lu, Shuxia Ye, and Liang Qi. Codetranfix: A neural machine translation approach for context-aware java program repair with codebert. *Applied Sciences*, 15(7):3632, 2025.

[102] Hongyin Luo, Lan Jiang, Yonatan Belinkov, and James Glass. Improving neural language models by segmenting, attending, and predicting the future. *arXiv preprint arXiv:1906.01702*, 2019.

[103] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[104] Yuetian Mao, Chengcheng Wan, Yuze Jiang, and Xiaodong Gu. Self-supervised query reformulation for code search. ESEC/FSE 2023, page 363–374, New York, NY, USA, 2023. Association for Computing Machinery.

[105] Stephen Merity, Nitish Shirish Keskar, and Richard Socher. An analysis of neural language modeling at multiple scales, 2018.

[106] Bertalan Meskó. Prompt engineering as an important emerging skill for medical professionals: tutorial. *Journal of medical Internet research*, 25:e50638, 2023.

[107] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffrey Dean. Distributed representations of words and phrases and their compositionality, 2013.

[108] Tomáš Mikolov, Wen-tau Yih, and Geoffrey Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 conference of the north american chapter of the association for computational linguistics: Human language technologies*, pages 746–751, 2013.

[109] Chris Mills, Esteban Parra, Jevgenija Pantiuchina, Gabriele Bavota, and Sonia Haiduc. On the relationship between bug reports and queries for text retrieval-based bug localization. *Empirical Software Engineering*, 25:3086–3127, 2020.

[110] Laura Moreno, John Joseph Treadway, Andrian Marcus, and Wuwei Shen. On the use of stack traces to improve text retrieval-based bug localization. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pages 151–160, 2014.

[111] Katsuhisa Morita, Tadahaya Mizuno, and Hiroyuki Kusuhara. Investigation of a data split strategy involving the time axis in adverse event prediction using machine learning. *Journal of Chemical Information and Modeling*, 62(17):3982–3992, 2022.

[112] Usmi Mukherjee and Mohammad Masudur Rahman. Understanding the impact of domain term explanation on duplicate bug report detection. *arXiv preprint arXiv:2503.18832*, 2025.

[113] MD Muktadir, Golam. A brief history of prompt: Leveraging language models. (through advanced prompting), September 2023.

[114] Anh Tuan Nguyen, Tung Thanh Nguyen, Jafar Al-Kofahi, Hung Viet Nguyen, and Tien N Nguyen. A topic-based approach for narrowing the search space of buggy files from a bug report. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 263–272. IEEE, 2011.

[115] Devon H O'Dell. The debugging mindset: Understanding the psychology of learning strategies leads to effective problem-solving skills. *Queue*, 15(1):71–90, 2017.

[116] Department of Computer Science. *Dependence and Data Flow Models*. University of Toronto, 2016. Chapter 6: Control Flow Graph and State Machine Models.

[117] Bhawna Paliwal, Deepak Saini, Mudit Dhawan, Siddarth Asokan, Nagarajan Natarajan, Surbhi Aggarwal, Pankaj Malhotra, Jian Jiao, and Manik Varma. Cross-jem: Accurate and efficient cross-encoders for short-text ranking tasks. *arXiv preprint arXiv:2409.09795*, 2024.

[118] Razvan Pascanu, Tomas Mikolov, and Yoshua Bengio. On the difficulty of training recurrent neural networks, 2013.

[119] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*, pages 1532–1543, Doha, Qatar, 2014. Association for Computational Linguistics.

[120] Sebastian Pier. Here's what caused the AT&T outage that blocked 92 million phone calls (plus 25,000 attempts to reach 911). *PhoneArena*, 2024.

[121] Denys Poshyvanyk, Yann-Gal Guneuc, Andrian Marcus, Giuliano Antoniol, and Vaclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Transactions on Software Engineering*, 33(6):420–432, 2007.

[122] Ronak Pradeep, Yuqi Liu, Xinyu Zhang, Yilin Li, Andrew Yates, and Jimmy Lin. Squeezing water from a stone: a bag of tricks for further improving cross-encoder effectiveness for reranking. In *European Conference on Information Retrieval*, pages 655–670. Springer, 2022.

[123] Binhang Qi, Hailong Sun, Wei Yuan, Hongyu Zhang, and Xiangxin Meng. Dreamloc: A deep relevance matching-based framework for bug localization. *IEEE Transactions on Reliability*, 71(1):235–249, 2022.

[124] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. Improving language understanding by generative pre-training. *OpenAI*, 2018.

[125] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems (NeurIPS 2019)*, 2019.

[126] M. M. Rahman and C. K. Roy. STRICT: Information Retrieval Based Search Term Identification for Concept Location. In *Proc. SANER*, pages 79–90, 2017.

[127] Mohammad Masudur Rahman, Foutse Khomh, Shamima Yeasmin, and Chanchal K Roy. The forgotten role of search queries in ir-based bug localization: an empirical study. *Empirical Software Engineering*, 26(6):116, 2021.

[128] Mohammad Masudur Rahman and Chanchal K Roy. Improved query reformulation for concept location using coderank and document structures. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 428–439. IEEE, 2017.

[129] Mohammad Masudur Rahman and Chanchal K Roy. Improving ir-based bug localization with context-aware query reformulation. In *Proceedings of the 2018 26th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 621–632, 2018.

[130] Mohammad Masudur Rahman and Chanchal K Roy. A systematic review of automated query reformulations in source code search. *ACM Transactions on Software Engineering and Methodology*, 2021.

[131] Mohammad Masudur Rahman, Chanchal K. Roy, and David Lo. Rack: Automatic api recommendation using crowdsourced knowledge. pages 349–359. Institute of Electrical and Electronics Engineers (IEEE), 5 2016.

[132] Stephen Robertson and Hugo Zaragoza. The probabilistic relevance framework: Bm25 and beyond. *Found. Trends Inf. Retr.*, 3(4):333–389, April 2009.

[133] Stephen E Robertson, Steve Walker, Susan Jones, Micheline M Hancock-Beaulieu, Mike Gatford, et al. Okapi at trec-3. *Nist Special Publication Sp*, 109:109, 1995.

[134] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533–536, 1986.

[135] Ripon K Saha, Julia Lawall, Sarfraz Khurshid, and Dewayne E Perry. On the effectiveness of information retrieval based bug localization for c programs. In *2014 IEEE international conference on software maintenance and evolution*, pages 161–170. IEEE, 2014.

[136] Ripon K Saha, Matthew Lease, Sarfraz Khurshid, and Dewayne E Perry. Improving bug localization using structured information retrieval. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 345–355. IEEE, 2013.

[137] Tobias Schnabel and Jennifer Neville. Prompts as programs: A structure-aware approach to efficient compile-time prompt optimization. *arXiv preprint arXiv:2404.02319*, 2024.

[138] Sander Schulhoff, Michael Ilie, Nishant Balepur, Konstantine Kahadze, Amanda Liu, Chenglei Si, Yinheng Li, Aayush Gupta, HyoJung Han, Sevien Schulhoff, et al. The prompt report: A systematic survey of prompting techniques. *arXiv preprint arXiv:2406.06608*, 2024.

[139] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 1715–1725, 2016.

[140] David Shepardson. AT&T February wireless outage blocked more than 92 million calls, agency says. *Reuters*, 2024.

[141] Jiho Shin, Clark Tang, Tahmineh Mohati, Maleknaz Nayebi, Song Wang, and Hadi Hemmati. Prompt engineering or fine tuning: An empirical assessment of large language models in automated software engineering tasks. *arXiv preprint arXiv:2310.10508*, 2023.

[142] Farhad Mortezapour Shiri, Thinagaran Perumal, Norwati Mustapha, and Raihani Mohamed. A comprehensive overview and comparative analysis on deep learning models: Cnn, rnn, lstm, gru, 2024.

[143] Bunyamin Sisman and Avinash C Kak. Incorporating version histories in information retrieval based bug localization. In *2012 9th IEEE working conference on mining software repositories (MSR)*, pages 50–59. IEEE, 2012.

[144] Bunyamin Sisman and Avinash C. Kak. Assisting code search with automatic query reformulation for bug localization. In *2013 10th Working Conference on Mining Software Repositories (MSR)*, pages 309–318, 2013.

[145] Anders Søgaard, Sebastian Ebert, Jasmijn Bastings, and Katja Filippova. We need to talk about random splits. In Paola Merlo, Jorg Tiedemann, and Reut Tsarfaty, editors, *Proceedings of the 16th Conference of the European Chapter of the Association for Computational Linguistics: Main Volume*, pages 1823–1832, Online, April 2021. Association for Computational Linguistics.

[146] Tim Sonnekalb, Bernd Gruner, Clemens-Alexander Brust, and Patrick Mäder. Generalizability of code clone detection on codebert, 2022.

[147] Venkatesh Srinivasan and Thomas Reps. An improved algorithm for slicing machine code. *ACM SIGPLAN Notices*, 51(10):378–393, 2016.

[148] Trevor Strohman, Donald Metzler, Howard R. Turtle, and W. Bruce Croft. Indri : A language-model based search engine for complex queries (extended version). 2005.

[149] Yi Sun, Hangping Qiu, Yu Zheng, Zhongwei Wang, and Chaoran Zhang. Sifrank: a new baseline for unsupervised keyphrase extraction based on pre-trained language model. *IEEE Access*, 8:10896–10906, 2020.

[150] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. Sequence to sequence learning with neural networks. *Advances in neural information processing systems*, 27, 2014.

[151] Mistral Team. Mistral: A new approach to language models, 2023.

[152] Qwen Team. Qwen: A high-performance language model, 2023.

[153] MIT Sloan Teaching & Learning Technologies. Effective prompts for ai: The essentials, September 16 2024.

[154] New York Times. Tech outage grounds flights across u.s., 2024. Accessed: 2024-08-04.

[155] Hugo Touvron, Antoine Bosselut, Kapil Sinha, and et al. Llama: Open and efficient foundation language models, 2023.

[156] Rishabh Upadhyay, Arian Askari, Gabriella Pasi, and Marco Viviani. Enhancing documents with multidimensional relevance statements in cross-encoder reranking. *arXiv preprint arXiv:2306.10979*, 2023.

[157] Greg Van Houdt, Carlos Mosquera, and Gonzalo Nápoles. A review on the long short-term memory model. *Artificial Intelligence Review*, 53(8):5929–5955, 2020.

[158] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[159] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

[160] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

[161] Junmei Wang, Min Pan, Tingting He, Xiang Huang, Xueyan Wang, and Xinhui Tu. A pseudo-relevance feedback framework combining relevance matching and semantic matching for information retrieval. *Information Processing & Management*, 57(6):102342, 2020.

[162] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, ISSTA 2015, page 1–11, New York, NY, USA, 2015. Association for Computing Machinery.

[163] Qianqian Wang, Chris Parnin, and Alessandro Orso. Evaluating the usefulness of ir-based fault localization techniques. In *Proceedings of the 2015 international symposium on software testing and analysis*, pages 1–11, 2015.

[164] Shaowei Wang and David Lo. Version history, similar report, and structure: Putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension*, pages 53–63, 2014.

[165] Shaowei Wang and David Lo. Amalgam+: Composing rich information sources for accurate bug localization. *Journal of Software: Evolution and Process*, 28(10):921–942, 2016.

[166] Xiao Wang, Sean MacAvaney, Craig Macdonald, and Iadh Ounis. Generative query reformulation for effective adhoc search, 2023.

[167] Xiao Wang, Craig Macdonald, and Iadh Ounis. Deep reinforced query reformulation for information retrieval. *arXiv preprint arXiv:2007.07987*, 2020.

[168] Yue Wang, Weishi Wang, Shafiq Joty, and Steven CH Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

[169] Albert Webson and Ellie Pavlick. Do prompt-based models really understand the meaning of their prompts? *arXiv preprint arXiv:2109.01247*, 2021.

[170] Lian Kit Wee. Here comes the wave of insurance claims for the crowdstrike outage, July 2024.

[171] Zeng Wei, Jun Xu, Yanyan Lan, Jiafeng Guo, and Xueqi Cheng. Reinforcement learning to rank with markov decision process. In *Proceedings of the 40th international ACM SIGIR conference on research and development in information retrieval*, pages 945–948, 2017.

[172] Ming Wen, Rongxin Wu, and Shing-Chi Cheung. Locus: Locating bugs from software changes. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 262–273, 2016.

[173] Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.

[174] David Weston. Helping our customers through the crowdstrike outage - the official microsoft blog, July 2024.

[175] Glynn Winskel. *The formal semantics of programming languages: an introduction*. MIT press, 1993.

[176] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, and Delangue. Huggingface's transformers: State-of-the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[177] Xi Xiao, Renjie Xiao, Qing Li, Jianhui Lv, Shunyan Cui, and Qixu Liu. Bugradar: Bug localization by knowledge graph link prediction. *Information and Software Technology*, page 107274, 2023.

[178] Yan Xiao, Jacky Keung, Qing Mi, and Kwabena E Bennin. Improving bug localization with an enhanced convolutional neural network. In *2017 24th Asia-Pacific Software Engineering Conference (APSEC)*, pages 338–347. IEEE, 2017.

[179] John Yang, Carlos Jimenez, Alexander Wettig, Kilian Lieret, Shunyu Yao, Karthik Narasimhan, and Ofir Press. Swe-agent: Agent-computer interfaces enable automated software engineering. *Advances in Neural Information Processing Systems*, 37:50528–50652, 2024.

[180] Xinli Yang, David Lo, Xin Xia, Lingfeng Bao, and Jianling Sun. Combining word embedding with information retrieval to recommend similar bug reports. In *2016 IEEE 27Th international symposium on software reliability engineering (ISSRE)*, pages 127–137. IEEE, 2016.

[181] Klaus Changsun Youm, June Ahn, Jeongho Kim, and Eunseok Lee. Bug localization based on code change histories and bug reports. In *2015 Asia-Pacific Software Engineering Conference (APSEC)*, pages 190–197, 2015.

[182] Shipeng Yu, Deng Cai, Ji-Rong Wen, and Wei-Ying Ma. Improving pseudo-relevance feedback in web information retrieval using web page segmentation. In *Proceedings of the 12th international conference on World Wide Web*, pages 11–18, 2003.

[183] Xuejun Zhang, Xia Hou, Xiuming Qiao, and Wenfeng Song. A review of automatic source code summarization. *Empirical Software Engineering*, 29(6):162, 2024.

[184] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models, 3 2023.

[185] Ziye Zhu, Yun Li, Hanghang Tong, and Yu Wang. Cooba: Cross-project bug localization via adversarial transfer learning. In *IJCAI*, 2020.

[186] Armin Zirak and Hadi Hemmati. Improving automated program repair with domain adaptation. *ACM Trans. Softw. Eng. Methodol.*, 33(3), mar 2024.

[187] Weiqin Zou, David Lo, Zhenyu Chen, Xin Xia, Yang Feng, and Baowen Xu. How practitioners perceive automated bug report management techniques. *IEEE Transactions on Software Engineering*, 46(8):836–862, 2018.

Appendix A

Complimentary Materials

A.1 Replication Packages

- IQLoc: https://github.com/asifsamir/IQLoc
- BRaIn: https://github.com/asifsamir/BRaIn