Bug Localiser-Team8

Ashish Sai Vignesh Nithin Sreya Rajeev

Cs21b006 cs21b028 cs21b039 cs21b041 cs21b049

ABSTRACT

Bug Localizer is a smart tool made to help software developers find and fix bugs in their Java code. What makes it special is that it uses deep learning to predict where bugs might be hiding.

Here's how it works: First, it looks at bug reports and figures out what might be causing the problem. Then, it digs deep into the structure of the code using source code representations called Abstract Syntax Trees (ASTs) and Control Flow Graphs (CFGs) and Data Flow Graphs(DFGs).

1 INTRODUCTION

In the dynamic realm of software development, the meticulous identification and resolution of bugs stand as pivotal processes ensuring the reliability, efficiency, and overall quality of the codebase. The Bug Localizer, emerges as a beacon of efficiency in this landscape. Its core mission is to revolutionize the bug-fixing journey by seamlessly integrating deep learning techniques.

2 DESIGN AND DEVELOPMENT

Data Collection: Initially we collected bug reports and their corresponding Java source code files directly from the repository and then formed the core data sources for both training and testing of the bug localization model. This robust approach ensures the model is trained on the most relevant and up-to-date information available.

The tool intelligently parses these bug reports, extracting crucial details such as issue descriptions, unique IDs, and accompanying metadata. This process not only captures the essence of the reported bugs but also ensures that all crucial information is fed into the model for comprehensive analysis.

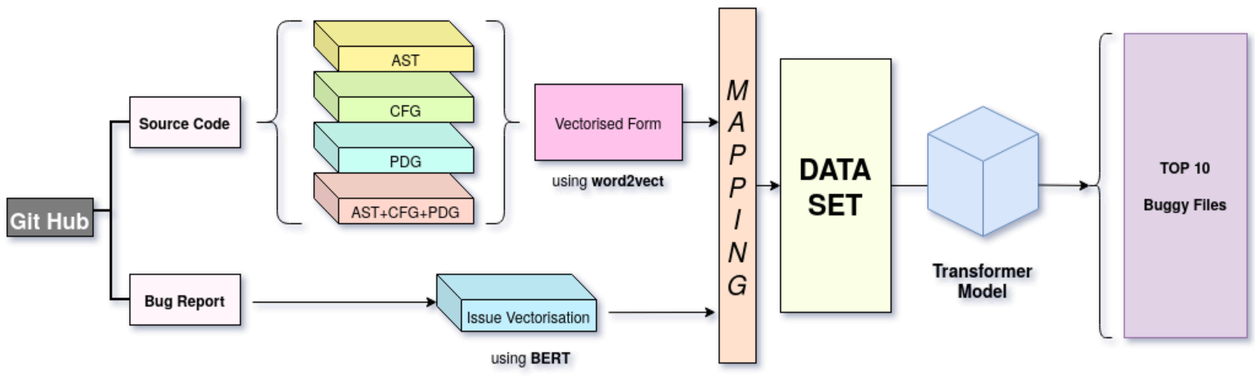
Language and Preprocessing: Python is used as the primary language for developing the Bug Localizer tool, providing a versatile and efficient framework for its implementation. On the other hand, Java stands as the language of the codebase under scrutiny, ensuring that the tool is well-equipped to navigate and analyze the Java source code effectively.

Before diving into the intricate task of bug localization, both bug reports and source code files undergo meticulous preprocessing. Bug reports are subjected to essential preprocessing steps such as tokenization, stop-word removal, and stemming. These steps are crucial for extracting meaningful keywords and standardizing the textual data, laying a solid foundation for the subsequent analysis.

In parallel, source code files are parsed to construct essential Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs). This multifaceted approach enables the Bug Localizer to deeply understand the structure and flow of the codebase, facilitating accurate bug localization.

Tools Used:

Pandas is harnessed for reading and preprocessing CSV files containing bug reports, providing a



robust and efficient means of handling large datasets.

SrcML plays a pivotal role in extracting the intricate Abstract Syntax Trees (ASTs), offering a detailed representation of the code's syntactic structure.

Progex steps in to extract the intricate Control Flow Graphs (CFGs), providing a deeper insight into how the code's logic flows.

Comex tool proves to be invaluable in extracting Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), which offer a nuanced view of the code's dependencies.

Additionally, the tool can combine these extracted representations, such as AST+CFG and CFG+DFG+AST, to create a comprehensive view of the codebase, enhancing the bug localization process with multifaceted insights.

Feature Extraction: Bug Localizer employs advanced techniques for feature extraction from both bug reports and the various code representations, such as ASTs, CFGs, DFGs, and their combinations. This involves transforming textual data of issues into numerical vectors, using the Pre-trained **BERT** model, which is very good for natural language processing, Bug Localizer gains a rich understanding of the semantics and context within bug reports. Now the source code representations are converted to numerical vectors using **Word2Vec** model which not only converts a tokens into a vector but also captures semantic relationships between tokens by considering their contextual usage in a large corpus. We can use other vectorization techniques like the **Count Vectorization** but it won't capture the semantic relationships between tokens.

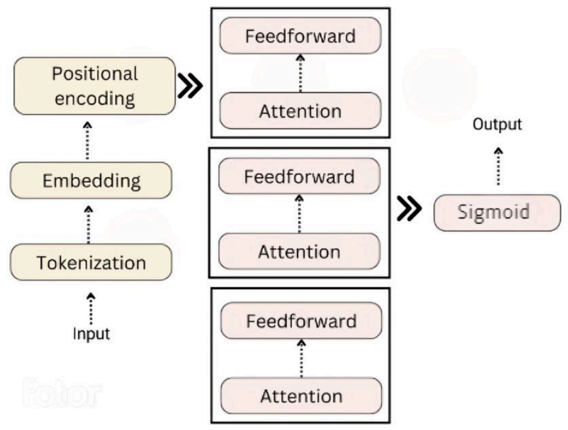
Source Code Analysis: Delving deep into the Java source code files, Bug Localizer meticulously extracts Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), and Data Flow Graphs (DFGs) alongside their combinations.

These representations serve as a structured view of the code's syntax and relationships between different code elements. This detailed analysis is crucial for the tool to grasp the intricate logic and dependencies within the codebase.

Model Training: The extracted features become the building blocks for training a sophisticated transformer model, a powerful deep-learning architecture chosen for its effectiveness in capturing complex relationships.

Initially we used a **Neural network model** with fully connected layers, but it suffered from overfitting and producing an accuracy of 98% on the whole dataset which is not desirable.

We used a **powerful Transformer model** with encoders and decoders, this eliminated the overfitting problem and produced good outputs.



The model undergoes training on labeled data, where bugs are mapped with specific code components. This meticulous training process enables the model to learn and generalize patterns, enhancing its bug localization capabilities.

Individual and Combined Training: To further enhance the model's understanding, we separately tried different combinations of source code representations and trained the transformer model with ASTs, CFGs, DFGs, and then their combinations. This approach allows the model to gain insights into the nuances of each code representation and how they interact when combined. The **Comex** tool plays a vital role in this process, facilitating the extraction of these combined representations for comparison.

Bug Localization: Armed with its trained deep learning model, Bug Localizer excels in predicting potential bug locations within the codebase when presented with new bug reports. The model's predictive capabilities enable it to identify the most probable areas where bugs are likely to occur. This functionality is invaluable for developers, as it streamlines the bug-fixing process by providing targeted insights into where to focus their efforts.

Evaluation: The effectiveness of the trained model is rigorously evaluated using test data, ensuring its robustness and accuracy in bug localization. Evaluation metrics such as accuracy, Mean Average Recall (MAR), and Mean Reciprocal Rank (MRR) scores are calculated.

These metrics offer valuable insights into the model's performance, providing developers with a clear understanding of its effectiveness in accurately localizing bugs within the codebase.

3. RELATED WORK

Previous research in bug localization has paved the way for tools like **DRAST**, which focused only on the Abstract Syntax Tree (AST) representation of source code.

Another tool, **BLUiR** requires only the source code and bug reports, it uses information retrieval technique but this is considering only bug similarity data.

Another tool, **SOBER** evaluates patterns of predicates in both correct and incorrect runs respectively, but it is a statistical debugging approach which doesn't consider syntax details.

Another method, **BLIA** integrates analyzed data by utilizing texts, stack traces and comments in bug reports, structured information of source files, and the source code change history, but this is mainly focusing only on the history of source codes.

Another model, **DNNLoc** based on textual similarity between bug reports and source files, relevancy feature from the DNN model to locate buggy files, but this also focuses only on bug similarity data.

OUR WORK:

Our aim is to push the boundaries of this research by incorporating Control Flow Graphs (CFG), Data Flow Graphs (DFG), and Mocktail representations of source code. This comprehensive approach allows for a more nuanced understanding of code structure and dependencies, leading to improved bug localization accuracy.

While other tools have been developed for bug localization, **none have utilized the Mocktail representation of source code**. Mocktail representations capture the dynamic behavior of code execution, providing valuable insights into runtime interactions. By integrating Mocktail alongside CFG and DFG, our tool aims to offer a holistic view of code behavior, enabling more precise bug localization.

Additionally, our approach includes a novel model for training the data with a powerful transformer architecture, by leveraging the combined power of these representations. This enhanced model not only considers the syntax but also the semantics and behavior of the code, resulting in a more robust and effective bug localization tool.

By expanding the scope to include multiple representations and introducing a sophisticated training model, our research contributes to the advancement of bug localization techniques, offering a more comprehensive and accurate solution for software developers.

4. RESULTS:

Here In our analysis, it's evident that the choice between CFG (Control Flow Graph) and AST (Abstract Syntax Tree) significantly impacts the MRR (Mean Reciprocal Rank) and MAR (Mean Average Rank) scores. CFG demonstrates superiority in handling larger files, yielding better MRR and MAR scores due to its compact vector representation. This efficiency enables CFG to efficiently capture the structural complexities present in larger-scale codebases. However, this advantage diminishes when dealing with smaller files, as the CFG representation tends to oversimplify the code structure, leading to suboptimal performance.

Conversely, AST, with its more comprehensive vector representation, excels in capturing nuanced details in smaller-scale files, resulting in better performance for such cases. The richer representation offered by AST allows for a more nuanced understanding of the code's structure and semantics, making it particularly effective for smaller files where intricate details play a significant role.

However, when applied to larger-scale files, the extensive vector representation of AST becomes a hindrance, as it leads to increased computational overhead and memory consumption. The exhaustive representation tends to generalize the code structure, limiting its effectiveness in capturing the intricate dependencies and relationships present in larger files. Consequently, AST may struggle to provide meaningful insights or accurate predictions for such scenarios.

Therefore, the choice between CFG and AST should be made considering the specific characteristics of the codebase being analyzed. For large-scale projects with complex code structures, CFG may offer better performance and scalability, whereas AST might be more suitable for smaller-scale projects requiring a detailed understanding of code semantics. Additionally, exploring hybrid approaches or refining the vector representations to strike a balance between comprehensiveness and efficiency could potentially enhance the effectiveness of both CFG and AST in various scenarios.

In conclusion, In the realm of code representation, the future holds promising avenues with the advent of mocktail-based approaches. These methodologies harness the power of diverse source code representations, such as CFG, AST, and DFG, to create a comprehensive and nuanced understanding of codebases. Building upon the insights garnered from our previous discussions, it's evident that each source code representation possesses distinct advantages and limitations across varying contexts.

In summary, mocktail-based representations represent the future frontier of code understanding and modeling. By harnessing the collective power of various source code representations, these models hold the key to unlocking new levels of performance, robustness, and versatility in software engineering applications.

INPUT	MAP	MRR
AST	0.367	0.125
CFG	0.486	0.201
DFG	0.276	0.108
AST+CFG	0.572	0.194
AST+DFG	0.06	0.15
CFG+DFG	0.20	0.11
AST+CFG+DFG	0.3	0.2

Table: Metric (MAP & MRR) calculation for different inputs

The above tabular results are for minimized dataset. To run for the complete dataset, we require higher computation resources.

5 USER SCENARIO

Imagine a scenario where a dynamic software development team is deeply engrossed in a large-scale Java project. As they navigate through the complexities of their codebase, encountering bug reports becomes a routine part of their workflow. In this crucial juncture, the Bug Localizer tool emerges as an indispensable ally.

Developers seamlessly input bug reports into the Bug Localizer tool, which swiftly springs into action by meticulously analyzing the associated Java source code. Powered by its robust deep-learning model, the tool embarks on a quest to unveil potential bug locations within the intricate codebase.

With its high precision, the Bug Localizer highlights these potential bug hotspots, effectively guiding developers to the heart of the matter. Armed with these invaluable insights, developers can strategically focus their efforts on these identified areas, accelerating the debugging process.

This symbiotic relationship between developers and the Bug Localizer tool serves as a catalyst for efficiency in the software development lifecycle. By streamlining the bug localization process, developers can swiftly and confidently address issues, leading to smoother development cycles and, ultimately, the delivery of high-quality software in a more timely manner.

6 DISCUSSION LIMITATIONS

While our Bug Localizer tool offers substantial advantages in the realm of bug localization, it is vital to address and recognize its inherent limitations:

System Configuration Constraints: The systems utilized for this project have limitations when it comes to handling larger datasets. Consequently, we were compelled to restrict the dataset size, which inevitably impacted the quality of the tool's output. This constraint highlights a potential area for future improvement in system scalability.

However, due to system constraints mentioned earlier, we had to compress the dataset into a smaller size for better performance. Enhancing the diversity of the dataset could significantly enhance the tool's effectiveness in varied scenarios.

Dependency on Bug Report Quality: The efficacy of our bug localization tool is intrinsically linked to the quality and detail of the bug reports it processes. Incomplete or ambiguous reports may pose challenges and potentially impact the tool's accuracy. This emphasizes the importance of maintaining high-quality bug reports to maximize the tool's utility.

Java Source Code Limitation: At its current iteration, our Bug Localizer tool is tailored specifically for Java source code analysis. Expanding its functionality to encompass other programming languages would necessitate substantial additional development efforts. This limitation is essential to consider for teams working with diverse codebases spanning multiple languages.

Training Data Quality and Bias: The accuracy and generalization capabilities of the deep learning model hinge on the quality and diversity of the training data. Biases or inadequacies within the training data can lead to suboptimal bug localization results. Continuous efforts to improve and diversify the training dataset are crucial for enhancing the tool's performance and reducing biases.

Addressing these limitations is pivotal for the ongoing development and enhancement of our Bug Localizer tool. By acknowledging these constraints, we pave the way for future improvements and refinements to overcome these challenges, ultimately striving towards a more robust and versatile bug localization solution.

7 CONCLUSION AND FUTURE WORK

This Project represents an advancement in bug localization within Java projects. By harnessing the power of deep learning, developers can more effectively identify and address bugs, improving software quality and development efficiency.

This Bug Localizer is poised to become an indispensable tool for software development teams, offering a proactive approach to bug detection and localization. As development continues, it holds the potential to revolutionize how developers approach debugging in software projects, fostering greater efficiency, code quality, and collaboration across the software development lifecycle.

In the future, several enhancements and directions for improvement can be explored:

Expansion to Other Languages: Extending the tool's capabilities to support languages beyond Java, such as C++, Python, and more. This expansion will enable a wider range of developers to benefit from its bug localization features, enhancing the tool's usability and impact across various software ecosystems.

Integration with Development Environments: Seamless integration with popular IDEs (Integrated Development Environments) to provide real-time bug localization suggestions as developers write code. This integration will empower developers with immediate feedback on potential bugs, allowing for swift resolution during the coding process. Features like intelligent code completion and context-aware bug suggestions within the IDE will streamline the development workflow and reduce the time spent on debugging.

References:

Tools:

1)comex -link

2) srcml -link

3)progex - link

Transformers:

1)Attention is All You Need - Original Transformer Paper -link

2)BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding-link

3)XLNet: Generalized Autoregressive Pretraining for Language Understanding -link

4)GPT (Generative Pre-trained Transformer)-link

5)RoBERTa: A Robustly Optimized BERT Approach-link

6)T5: Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer-link

7)BART: Denoising Sequence-to-Sequence Pre-training for Natural Language Generation, Translation, and Comprehension-link

8)ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators-link

9)Reformer: The Efficient Transformer-link

10)Longformer:Long-Document Transformer-link

Bug Localization:

1)Roadmap for Bug Localization -link

2)Survey on Bug Localization Techniques-link

3)Deep Learning Based Bug Localization-link

4)Bug Localization Using Neural Networks-link

5)Graph-Based Bug Localization-link

6) BLUiR: A Bug Localization Tool for Large-Scale Software Development-link

7)Bug Localization using Bug Reports -link

8)Bug Localization in Large-Scale Parallel Programs-link

9)Bug Localization Using Machine Learning Techniques-link

10)Combining Textual and Structural Information for Bug Localization-link