CSC 680 Project Report: BugLocator: Automated Bug Localization from Bug Reports to Buggy Code Methods

Enze Xu*
exu03@wm.edu
College of William & Mary
Williamsburg, Virginia, USA

Yi Lin*
ylin13@wm.edu
College of William & Mary
Williamsburg, Virginia, USA

ACM Reference Format:
Enze Xu and Yi Lin. 2024. CSC 680 Project Report: BugLocator: Automated Bug Localization from Bug Reports to Buggy Code Methods. In *Proceedings of (CSCI 680 AI for Software Engineering).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Software development is a highly collaborative and iterative process, wherein identifying and resolving bugs is a critical component of ensuring software quality. Users often report bugs they encounter via issue tracking systems, initiating a sequence of activities for developers that typically includes bug replication, localization, and resolution. Among these, bug localization—pinpointing the specific source code elements (e.g., methods, classes, or files) that cause the bug—is particularly challenging. In large-scale software systems, the sheer volume and complexity of code can make this task extremely time-consuming, even for experienced developers. Similarly, in smaller systems, developers unfamiliar with the domain or the software's architecture may face considerable difficulties in efficiently identifying buggy elements.

To address these challenges, numerous techniques have been proposed to assist developers in bug localization. Traditional approaches often leverage information retrieval (IR) methods or, more recently, deep learning models. These methods take textual bug reports as input and generate ranked lists of code elements likely to contain the bug. While effective to an extent, these techniques frequently suffer from limitations: their performance metrics remain suboptimal, and they generally provide coarse-grained results, such as identifying buggy files, without pinpointing specific locations within these files.

In recent years, advancements in deep learning and structural code analysis have opened new avenues for improving bug localization. A notable example is Athena, a technique designed to assess the impact of code changes by combining graph neural networks with structural information, such as call graphs, to generate rich code embeddings. Athena excels in encoding inter-method relationships within software projects, enabling it to suggest methods likely affected by changes. Motivated by these capabilities, we hypothesize that Athena's deep representation learning framework can be adapted to improve buggy code localization. By leveraging its ability to encode both semantic and structural information, Athena holds promise for addressing the granularity and accuracy issues that plague existing bug localization techniques.

In this project, we aim to develop and evaluate a novel bug localization approach that leverages Athena's capabilities in combination with large language models (LLMs). The approach will integrate textual bug report embeddings derived from LLMs, with Athena's code embeddings to identify specific methods likely to contain the reported bug. By associating natural language descriptions with structural code representations, the proposed method seeks to address the limitations of existing techniques, providing a more granular and accurate bug localization solution.

2 Background

2.1 Bug Locallization

Bug localization is the process of identifying specific parts of a software system's source code that are responsible for a reported bug. When users or testers encounter an issue, they often describe the bug in natural language and submit it as a bug report through an issue tracker. Bug localization aims to bridge the gap between this textual description and the software's codebase by identifying the files, classes, methods, or lines of code that likely contain the bug.

This process is critical because manually searching for bugs can be extremely time-consuming, especially in large-scale software projects with thousands of interconnected code components. Bug localization not only accelerates the debugging process but also helps developers focus their efforts on relevant areas of the codebase.

Traditional bug localization techniques often rely on information retrieval (IR) approaches, which calculate the similarity between the bug report and code artifacts to rank potential buggy files. More recent methods leverage machine learning or deep learning models to extract patterns from bug reports and code structures, offering improved precision and granularity. These techniques may also incorporate auxiliary data, such as code change histories, stack traces, or execution logs, to enhance the accuracy of their predictions.

2.2 Large Language Model(LLM)

Large Language Models (LLMs) are advanced artificial intelligence systems designed to process and generate human-like text based

*Both authors contributed equally to this research.

CSCI 680 AI for Software Engineering, Williamsburg, VA, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

on extensive training on vast amounts of textual data. Examples of LLMs include GPT (Generative Pre-trained Transformer), BERT (Bidirectional Encoder Representations from Transformers), and their derivatives. These models are typically built using Transformer architectures, which allow them to effectively handle sequential data, capture contextual relationships between words, and understand language nuances.

LLMs are pre-trained on diverse datasets, ranging from books and articles to code repositories, enabling them to generalize across a wide range of tasks. After pre-training, they can be fine-tuned for specific applications, such as text summarization, sentiment analysis, code generation, and even domain-specific problem-solving.

In the context of bug localization, LLMs can be employed to analyze the natural language descriptions in bug reports and generate embeddings (vectorized representations) that capture semantic information. These embeddings can then be combined with representations of source code to identify potential buggy components. LLMs offer significant advantages due to their ability to handle unstructured text, generalize across different domains, and leverage semantic understanding to bridge the gap between bug reports and code elements.

3 Related Work

The paper “Where Should the Bugs Be Fixed? More Accurate Information Retrieval-Based Bug Localization Based on Bug Reports” by Jian Zhou, Hongyu Zhang, and David Lo [6] presents a method called BugLocator, which enhances bug localization using a revised Vector Space Model (rVSM) and information from previously fixed similar bugs. Traditional bug localization methods based on Information Retrieval (IR) models struggle with ranking accuracy and tend to rely heavily on the quality of bug reports. BugLocator addresses these limitations by refining the IR-based approach and considering both the similarity between bug reports and source code files and the historical data of previously fixed bugs.

“Localization” by Fan Fang et al. [1] proposes a method to enhance bug localization by classifying bug reports as either “informative” or “uninformative”. Many bug reports lack useful information, reducing the accuracy of information retrieval (IR)-based bug localization systems. The authors introduce a model that combines both implicit features (learned through neural networks like LSTM, CNN, and multilayer perceptron) and explicit features (such as report length and inclusion of code samples) to classify reports.

The study uses over 9,000 bug reports from open-source Java projects for evaluation. It shows that filtering out uninformative reports before running the localization process significantly improves precision and recall in locating buggy files. The LSTM-based classifier outperforms other models, and the combination of both explicit and implicit features further enhances performance. By classifying bug reports, the method helps reduce false positives and improves the efficiency of IR-based bug localization, making it more accurate and effective.

The paper “Bug Localization Based on Code Change Histories and Bug Reports” by Klaus Changsun Youm et al. [5] introduces BLIA (Bug Localization with Integrated Analysis), an enhanced approach for bug localization that combines information from bug reports, source code change histories, and stack traces. BLIA improves upon traditional information retrieval (IR)-based methods by using a revised Vector Space Model (rVSM) to analyze the similarity between bug reports and source files. It also incorporates historical bug fix data and analyzes stack traces to identify potential buggy files.

The integration of these multiple information sources enables more accurate bug localization, improving the ranking of suspicious files. Experiments conducted on three open-source projects (AspectJ, SWT, and ZXing) demonstrated that BLIA outperforms existing tools like BugLocator, BLUiR, and AmaLgam, with improvements of up to 34% in mean average precision (MAP). This method significantly enhances bug localization accuracy, reducing developer effort in identifying faulty code.

The paper “Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs” by Yanfu Yan et al. [4] introduces Athena, a novel approach for impact analysis (IA) that integrates Transformer-based neural models, such as GraphCodeBERT, with program dependence graphs to improve method-level impact prediction. Athena enhances traditional conceptual IA techniques by combining semantic code representations with structural dependencies through an embedding propagation strategy inspired by graph convolutional networks. Additionally, the authors present Alexandria, a large-scale IA benchmark based on untangled bug-fix commits from 25 open-source Java projects. Athena significantly outperforms baseline methods, demonstrating state-of-the-art performance in estimating the impact of code changes.

4 Preliminaries

4.1 Problem Statement

In this project, our research problem focuses on improving bug localization in Java repositories. Giving a bug report br and a list of method-level code snippets $M = \{m_i\}$, the task is to locate the method m with the highest similarity score, which can be written as Eq. 1.

$$m = \arg\max_{m \in M} f(br, m), \tag{1}$$

where $f : BR \times M \rightarrow R$ denote the similarity score function representing how similar br and m are. The problem can be divided into two distinct tasks: In-Distribution Task and Out-of-Distribution Task. In this project, our work focuses on the In-Distribution Task. Giving a dataset $\{(br_i, m_i, s_i)\}$ from a repo R and a new bug report br_{new} not seen in the training set, traverse all Java methods$\{m_k\}$ in the same repo R. Predict similarity pairs $\{(m_k, s_k)\}$ for the new bug report and recommend a fixed number of methods most likely to be buggy.

4.2 T5 & CodeT5

Before training our model, it is essential to convert bug reports and methods into word embeddings to enable the model to process and understand the textual and structural data effectively. In our work, we use T5 (Text-to-Text Transfer Transformer) [2] for this purpose.

T5 is a pre-trained Transformer-based model capable of generating high-quality contextual embeddings. Specifically, we leverage T5’s encoder to process the bug reports, which are provided in natural language, and generate dense vector representations for each word or subword token. These embeddings capture both the semantic meaning of the words and their contextual relationships within the bug report, making them well-suited for tasks requiring a deep understanding of language.

Similarly, for the methods in the source code, we preprocess them into a suitable textual representation (e.g., method signatures, comments, or relevant code snippets) and input them into the CodeT5 [3] - another pre-trained model based on the T5. This process yields embeddings that represent the structural and semantic characteristics of the methods, facilitating the alignment between bug reports and code during training.

By using T5 and CodeT5 embeddings, we ensure that both the bug reports and methods are represented in a unified, meaningful space, enabling our model to effectively learn associations and improve its bug localization capabilities.

5 Method

We propose BLNT5 (Bug-Localization Network with T5 and CodeT5), a framework designed to address the proposed tasks. BLNT5 includes two variants: a Concatenation-based approach (Concat) and a Cosine Similarity-based approach (CosSim). Below, we provide a detailed description of each variant.

5.1 Concat Approach

The framework for BLNT5-Concat is illustrated in Figure 1. First, we process the bug report (br) and the method code snippet (m) through pretrained T5 (google-t5/t5-small) and CodeT5 (Salesforce/codet5-base) models (as described in Section 4.2), extracting their respective embedding layer outputs, denoted as v_1 and v_2. These vectors represent the latent space of the bug report and method, respectively, after applying a normalization step.

In the Concat approach, we concatenate the normalized vectors v_1 and v_2 to form a combined vector v. This combined vector is then passed through a Multi-Layer Perceptron (MLP) followed by a sigmoid activation function to compute the similarity score $\hat{s} \in [0, 1]$.

5.2 CosSim Approach

The framework for BLNT5-CosSim is shown in Figure 2. Similar to the Concat approach, we first extract and normalize the latent representations of the bug report and method, obtaining v_1 and v_2.

In the CosSim approach, the vectors v_1 and v_2 are independently transformed into the same latent space dimensions, resulting in v_1' and v_2', using two separate MLPs. The cosine similarity between v_1' and v_2' is then computed to quantify their similarity. Finally, a sigmoid activation function is applied to map the similarity score to the range $[0, 1]$.

5.3 Loss Function

Both variants of our method use the same loss function, a cross-entropy-based formulation, defined as:

$$\mathbb{L} = -(s \cdot \log(\hat{s}) + (1 - s) \cdot \log(1 - \hat{s})), \tag{2}$$

where $\hat{s} = \sigma(s_{\text{out}})$ is the predicted similarity score, s denotes the ground-truth similarity score, and s_{out} represents either the output of the MLP in the Concat approach or the cosine similarity in the CosSim approach. The sigmoid function $\sigma(x)$ is defined as:

$$\sigma(x) = \frac{1}{1 + e^{-x}}. \tag{3}$$

6 Dataset

6.1 Dataset Format

Our dataset consists of samples in the format (br_i, m_i, s_i), where:

- br_i: Bug report (issue), including the issue title and body.
- m_i: Java code snippet (at the method level).
- $s_i \in [0, 1]$: Similarity score.

6.2 Dataset Definition

To construct the dataset:

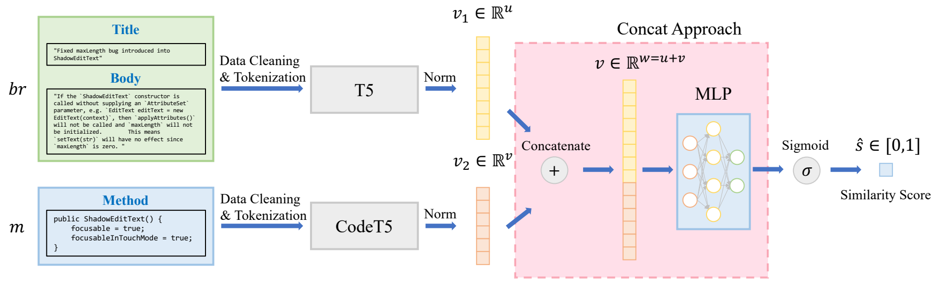
- If a method m in a closed bug report (br) is related to the bug report, we assign a similarity score $s = 1.0$.
- For other non-related methods m', randomly selected within the same Java file, we assign $s = 0.0$.

This strategy may have limitations, as the similarity score between a method and a bug report could theoretically be a continuous value between 0 and 1. However, for the purpose of automatic dataset collection, this approach is the most practical.

6.3 Dataset Collection Process

The dataset collection process follows the steps illustrated in Figure 3:

(1) We search GitHub repositories using GitHub Search (https://seart-ghs.si.usi.ch/) with specific filtering criteria. The repository list is curated to include over 50 repositories that meet our filter criteria, focusing on large, long-term, and popular Java repositories with a substantial history of bug reports.
(2) We focus exclusively on closed bug reports (GitHub Issues), as they often include related code commits that address the reported issues. These closed bug reports are retrieved using the GitHub REST API.
(3) We use the GitHub REST API to retrieve specific issues in JSON format.
(4) From the retrieved JSON, we extract the title and body of the bug report, which together form the textual representation of the bug report (br).
(5) We use the GitHub REST API and a pull request localization algorithm to retrieve a list of pull requests related to the bug report.
(6) From each pull request, we locate a list of code commits using the GitHub REST API.
(7) Positive code snippets (m) are localized and downloaded from each commit using a code snippet localization algorithm.
(8) Negative code snippets (m') are randomly selected from the repository using a random code snippet localization algorithm.
(9) The bug report (br) together with the positive code snippets (m) form the positive data samples $(br, m, 1.0)$. The bug report (br) together with the negative code snippets (m') form the negative data samples $(br, m', 0.0)$.

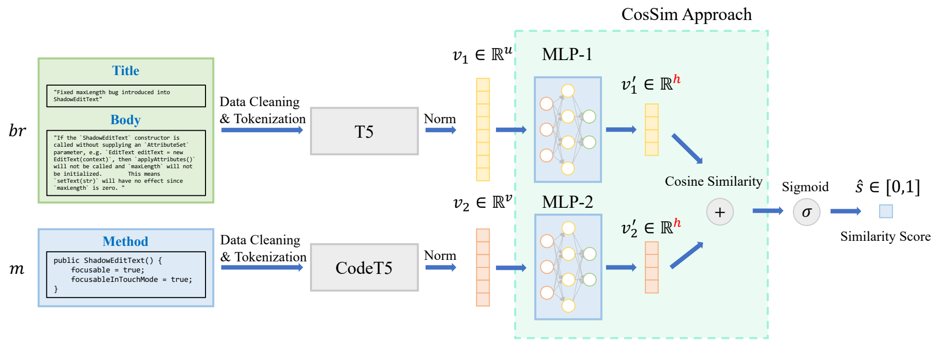


```
"Fixed maxLength bug introduced into ShadowEditText"
```

```
"If the `ShadowEditText` constructor is called without supplying an `AttributeSet` parameter, e.g. `EditText editText = new EditText(context)`, then `applyAttributes()` will not be called and `maxLength` will not be initialized.        This means `setText(str)` will have no effect since `maxLength` is zero. "
```

```
public ShadowEditText() {
    focusable = true;
    focusableInTouchMode = true;
}
```

Figure 1: Framework of the BLNT5-Concat approach, where the bug report and method embeddings are concatenated, followed by an MLP and sigmoid activation to compute the similarity score.

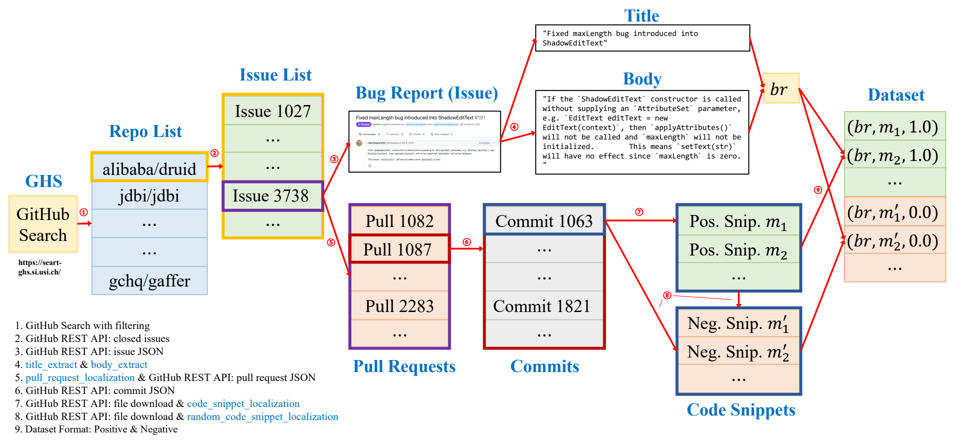


```
"Fixed maxLength bug introduced into ShadowEditText"
```

```
"If the `ShadowEditText` constructor is called without supplying an `AttributeSet` parameter, e.g. `EditText editText = new EditText(context)`, then `applyAttributes()` will not be called and `maxLength` will not be initialized.        This means `setText(str)` will have no effect since `maxLength` is zero. "
```

```
public ShadowEditText() {
    focusable = true;
    focusableInTouchMode = true;
}
```

Figure 2: Framework of the BLNT5-CosSim approach, where the bug report and method embeddings are transformed to a shared latent space, and cosine similarity is used to compute the similarity score.



```
"Fixed maxLength bug introduced into ShadowEditText"
```

```
"If the `ShadowEditText` constructor is called without supplying an `AttributeSet` parameter, e.g. `EditText editText = new EditText(context)`, then `applyAttributes()` will not be called and `maxLength` will not be initialized.        This means `setText(str)` will have no effect since `maxLength` is zero. "
```

1. GitHub Search with filtering
2. GitHub REST API: closed issues
3. GitHub REST API: issue JSON
4. title_extract & body_extract
5. pull_request_localization & GitHub REST API: pull request JSON
6. GitHub REST API: commit JSON
7. GitHub REST API: file download & code_snippet_localization
8. GitHub REST API: file download & random_code_snippet_localization
9. Dataset Format: Positive & Negative

Figure 3: Dataset Collection Process.

6.4 Dataset Example

Each dataset example comprises the following components:

- **Bug Report (*br*):** The title and body of a closed (resolved) issue.
- **Java Method (*m*):** A code snippet that implements the fix for the corresponding issue.
- **Similarity Score (*s*):** A value of 1.0, indicating a positive match between the bug report and the code snippet.

While our model primarily utilizes these three entities (*br*, *m*, *s*), the dataset collection process involves gathering a total of 15 fields for each data sample. These additional fields provide valuable context and are outlined in Table 1.

6.5 Dataset Statistics

For our proposed In-distribution task introduced, we collect data samples from the same GitHub repository robolectric/robolectric.

Field	**Value**
Repository	`robolectric/robolectric`
Repository API URL	https://api.github.com/repos/robolectric/robolectric
Issue API URL	https://api.github.com/repos/robolectric/robolectric/issues/9608
Issue ID	9608
Issue Title	"Fix resetting the AppOpsManager OnOpNotedCallback"
Issue Body	"After the first call to the resetter, it wasn't being reset any more. Update the resetter to avoid clearing staticallyInitialized."
Pull Request API URL	https://api.github.com/repos/robolectric/robolectric/pulls/9608
Pull Request ID	9608
Pull Request Title	"Fix resetting the AppOpsManager OnOpNotedCallback"
Pull Request Body	"After the first call to the resetter, it wasn't being reset any more. Update the resetter to avoid clearing staticallyInitialized."
Commit API URL	https://api.github.com/repos/robolectric/robolectric/commits/93781ff1de42b93765c6b4f190d896db3c2334ce
Commit ID	93781ff1de42b93765c6b4f190d896db3c2334ce
Commit Message	"Fix resetting the AppOpsManager OnOpNotedCallback. After the first call to the resetter, it wasn't being reset any more. Update the resetter to avoid clearing staticallyInitialized."

Commit Code Snippet	`public static void reset() {`
	`    // The callback passed in AppOpsManager#setOnOpNotedCallback is stored statically.`
	`    // The check for staticallyInitialized is to make it so that we don't load AppOpsManager if it`
	`    // hadn't already been loaded (both to save time and to also avoid any errors that might`
	`    // happen if we tried to lazy load the class during reset)`
	`    if (RuntimeEnvironment.getApiLevel() >= R && staticallyInitialized) {`
	`        ReflectionHelpers.setStaticField(AppOpsManager.class, "sOnOpNotedCallback", null);`
	`    }`
	`    storedOps.clear();`
	`    appModeMap.clear();`
	`    longRunningOp.clear();`
	`    appOpListeners.clear();`
	`    audioRestrictions.clear();`
	`}`

Similarity Score	1.0

Table 1: Detailed structure of a dataset example, including the bug report, associated Java method, similarity score, and additional contextual fields collected during the dataset creation process.

It contains a total of 18,583 (br_i, m_i, s_i) pairs, with the following distribution:

- Positive samples ($s = 1.0$): $9,555$ pairs.
- Negative samples ($s = 0.0$): $9,028$ pairs.

The data is split into training, validation, and test sets as follows:

- Train: 14,866 (80%).
- Validation: 1,858 (10%).
- Test: 1,859 (10%).

Figure 4 shows the token length distributions for bug reports and methods.

7 Evaluation

7.1 Metrics

To evaluate the performance of our BLNT5 model, we apply a threshold of 0.5 to convert the predicted similarity scores into binary values (0 or 1) and use standard classification metrics: Accuracy, Precision, Recall, and F1-Score.

Accuracy is computed as follows: first, the Sigmoid function is applied to the outputs of the trained MLP model, transforming the raw outputs into probabilities. Next, a threshold $\tau = 0.5$ is used to classify the probabilities into binary values (0 or 1). Finally, the predictions are compared with the binary ground truth labels to calculate accuracy. The same thresholding process is used to compute Precision, Recall, and F1-Score.

7.2 Baselines

We do not include comparisons with existing methods in this study. Instead, we establish baselines using our two approaches (Concat and CosSim) with randomly initialized weights for the MLP layers. These baselines serve as a point of reference for evaluating the effectiveness of our proposed methods.

7.3 Experiment Settings

All experiments are conducted on an Ubuntu 20.04 machine with the following specifications:

- Processor: AMD EPYC 7543 32-Core (64 threads, 2 threads per core).
- Memory: 256 GB.
- GPUs: Four NVIDIA RTX A5000, each with 24 GB memory.

In the experiments, we set the batch size to 64 and use the Adam optimizer with a learning rate of 0.001. The dataset is split into training, validation, and test sets in an 8:1:1 ratio.

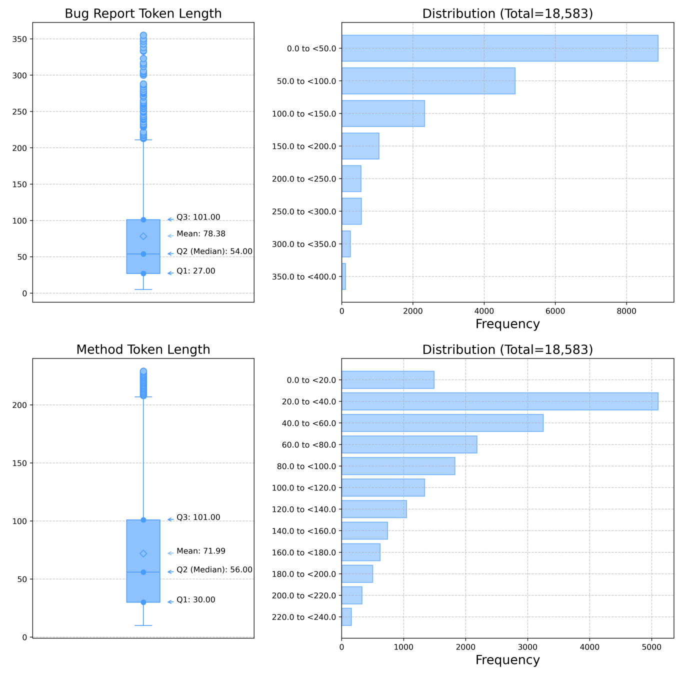


Figure 4: Distribution of bug report and method token lengths in the In-dis dataset.

7.4 Experiment Results

Method	**Accuracy ↑**	**Precision ↑**	**Recall ↑**	**F1-Score ↑**
Random-CosSim	0.4982	0.4989	0.5750	0.4937
Random-Concat	0.5019	0.4992	0.5497	0.4335
BLNT5-CosSim	**0.6218**	**0.6258**	0.6527	0.6389
BLNT5-Concat	0.6100	0.6044	**0.6925**	**0.6455**

Table 2: Comparison of performance metrics for different methods. Random methods use randomly initialized MLP weights, averaged over 10 random seeds.

As shown in Table 2, both BLNT5 variants, CosSim and Concat, significantly outperform their respective random baselines. Notably, BLNT5-Concat achieves the highest F1-Score of 0.6455.

8 Conclusions

In this work, we constructed a dataset of over 217,000 samples from 50+ GitHub repositories, pairing bug reports with associated methods. To address the bug localization In-Dis task, we proposed the BLNT5 framework with two variants: Concat and CosSim. Both approaches demonstrate superior performance compared to their random baselines, as shown in the experimental results.

For future work, we plan to explore:

- Out-of-Distribution (OOD) experiments to assess the robustness of BLNT5.
- Extension to additional programming languages, such as Python and JavaScript.
- Comparisons with existing baseline methods to further evaluate performance.

9 Acknowledgments

We, Enze Xu and Yi Lin, would like to express our heartfelt gratitude to Dr. Antonio Mastropaolo for the invaluable guidance and mentorship throughout this work. Your insights and support have been instrumental in shaping the direction and execution of our research.

We also extend our sincere thanks to Dr. Oscar Chaparro, and Daoxuan Xu for their assistance and contributions. Their expertise and helpful suggestions have significantly enriched this study.

Thank you to all for your encouragement and support in making this project a success.

References

[1] Fan Fang, John Wu, Yanyan Li, Xin Ye, Wajdi Aljedaani, and Mohamed Wiem Mkaouer. 2021. On the classification of bug reports to improve bug localization. *Soft Computing* 25 (2021), 7307–7323.

[2] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. 2020. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of machine learning research* 21, 140 (2020), 1–67.

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

[4] Yanfu Yan, Nathan Cooper, Kevin Moran, Gabriele Bavota, Denys Poshyvanyk, and Steve Rich. 2024. Enhancing Code Understanding for Impact Analysis by Combining Transformers and Program Dependence Graphs. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 972–995.

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

[6] Jian Zhou, Hongyu Zhang, and David Lo. 2012. Where should the bugs be fixed? more accurate information retrieval-based bug localization based on bug reports. In *2012 34th International conference on software engineering (ICSE)*. IEEE, 14–24.