

Article

Test-Free Fault Localization Through Deep Source Code Analysis Using Encoder-Based Language Models

Jesus-Angel del-Hoyo-Gabaldon , Antonio Garcia-Cabot * , Eva Garcia-Lopez 
and David de-Fitero-Dominguez 

Departamento de Ciencias de la Computación, Universidad de Alcalá, 28801 Madrid, Spain

* Correspondence: a.garciac@uah.es

Abstract

Fault Localization (FL) has evolved with the emergence of the Transformer architecture. Traditionally, FL relied on generating a ranking of suspiciousness scores for each code unit using information obtained during program execution from a set of test cases. However, Language Models trained with source code have enabled approaches that avoid the need for test suites, which are, sometimes, difficult or time-consuming to obtain. In this work, we present a refined training strategy tested on DeBERTa-v3-base, leveraging an encoder-based approach, to address two key challenges in FL: (1) the accurate classification of lines of code according to their suspiciousness scores, and (2) the class imbalance inherent in this type of problem. To this end, we extract the last hidden state corresponding to the embeddings of the new line tokens to generate a suspiciousness score for each line. This score then classifies each line as defective or not. The evaluation of our approach on the ConDefects dataset, repeated over ten random seeds, shows promising results compared to existing approaches, with ROC AUC gains up to 0.27 in cross-dataset comparisons. Thereby, we offer a competitive baseline for future test-free fault localization research while using language models that are approximately 200 times smaller than those used in prior work.

Keywords: fault localization; line-level granularity; transformer architecture; language models; LLM; encoders

1. Introduction

Traditional Fault Localization (FL) techniques primarily rely on program execution with a suite of test cases to generate suspiciousness scores for each analyzable code unit [1]. These units can vary in granularity, with function and line levels being especially attractive for practitioners [2]. To extract suspiciousness scores, one of the most extended solutions in traditional FL techniques is to build a coverage matrix that captures execution traces of programs using existing test cases and then compute metrics that capture relationships between the results obtained and the expected results [1,3]. With the increasing presence of machine learning and deep learning-based methods, new approaches based on neural networks that extract relevant features from code and code coverage matrices have emerged [1]. Despite their potential, most of these methodologies still require executable programs and comprehensive test suites, resources that are often inaccessible or challenging to design [4]. Furthermore, while software practitioners are enthusiastic about research in FL, they exhibit high adoption thresholds regarding analysis granularity (e.g., method, line, or block level) and the maximum number of code units that practitioners need to



Academic Editor: Christos Bouras

Received: 30 May 2026

Revised: 13 July 2026

Accepted: 14 July 2026

Published: 17 July 2026

Copyright: © 2026 by the authors.

Licensee MDPI, Basel, Switzerland.

This article is an open access article distributed under the terms and conditions of the [Creative Commons Attribution \(CC BY\)](https://creativecommons.org/licenses/by/4.0/) license.

analyze (typically up to five elements, beyond which practitioners' satisfaction decreases significantly) [2].

Among the most relevant solutions in FL are Tarantula [5] and Ochiai [3], which are oriented towards line-level and method-level FL, respectively. Subsequent studies have investigated methods to enhance these solutions, a significant number of which are outlined in the work of Wong et al. [6]. However, most of these approaches are contingent on the utilization of test cases, mainly because the literature has been historically focused on Spectrum-based FL [3] and Mutation-based FL [7], both of which need source code execution [6]. Indeed, regarding test-free FL, findings in the existing literature have revealed minimal evidence. However, with the advent of Transformer architecture, researchers have explored new ways to find an end-to-end solution that overcomes these challenges.

Transformer architecture is a type of neural network that overcomes the limitations associated with prior topologies commonly used in language modeling, such as Feedforward Neural Networks and Long-Short Term Memory networks, thanks to its parallelization capabilities provided by the inclusion of the attention mechanism [8]. In recent years, Language Models (LMs), Large Language Models (LLMs), and Language Models for Code (LMCs) have proliferated, all of them capable of processing source code. Unlike previous approaches, these models can identify faults in code even in the presence of syntax errors, without requiring code execution [4]. This represents a significant benefit, since it avoids the need for test cases. In light of these characteristics, LLMAO emerges [4], the first language model-based architecture that detects line-level errors without program execution and that offers a more unified approach closely aligned with the expectations of practitioners when compared with previous work. The authors define line-level FL as a classification task where they prompt a decoder-based language model to compute a suspiciousness score per line. This suspiciousness score ranges from 0 to 1, where higher values represent lines more likely to be defective. However, we have identified some room for improvement. First, encoder-based language models excel at classification tasks, whereas decoder-based language models seem to be more limited. In addition, the utilization of neural networks to solve FL faces a substantial challenge: in general, the number of correct lines in source code exceeds the number of defective lines. Consequently, neural networks may become biased during their training, misclassifying source code as non-vulnerable. To overcome the previous limitations, we present a work that employs the encoder-based language model DeBERTa-v3-base [9,10], combined with a balancing technique based on downsampling, to improve the performance on the line-level FL task.

In summary, this work presents three key contributions to the field.

- We explore the application of encoder-based architectures to improve Test-Free Fault Localization, specifically the DeBERTa-v3-base model. The proposed methodology receives source code as input and generates a ranked list of suspiciousness scores corresponding to each line of code (LOC).
- We analyze the impact of class imbalance on the performance of neural networks applied to FL. We identify that this imbalance, inherent in most source code databases, can bias models towards the majority class (fault-free code), which reduces their ability to detect actual faults. To mitigate this issue, we propose and implement a downsampling strategy aimed at balancing the representation of both classes during training, and we compare it against two standard alternatives, class-weighted loss and focal loss [11].
- We present an initial evaluation of encoder performance on the ConDefects benchmark, specifically designed to mitigate data leakage issues. To the best of our knowledge, this is the first evaluation of an encoder-based approach for test-free fault localization using this benchmark; so, this work provides a baseline for future research in FL, which will

enable rigorous comparisons in this domain. All reported results are averaged over multiple random seeds and accompanied by statistical significance analysis, and the proposed strategy is additionally validated on three alternative encoder architectures (CodeBERT [12], GraphCodeBERT [13], and RoBERTa [14]).

The remainder of this paper is structured as follows. First, Section 2 reviews the most relevant advances in the literature relative to the traditional and modern FL task. Subsequently, Section 3 details the dataset used, the modifications made to DeBERTa-v3-base to retrieve a list of suspiciousness scores, the training procedure, and the evaluation metrics employed. Section 4 presents the evaluation of our experiments and a comparison with other existing works. Finally, Section 5 first summarizes the key findings and then proposes future research directions to further enhance the results.

2. Related Work

FL techniques traditionally analyzed source code components to obtain a suspiciousness score for each component [1]. These units are characterized by varying levels of granularity, with line level granularity being the most studied in the literature [2], although some studies indicate that practitioners prefer method granularity slightly over line granularity [2]. One of the most influential techniques in this field is Spectrum-based Fault Localization (SBFL), which involves executing a program with some passing and failing test suites to generate a coverage matrix. A suspiciousness score is then computed from this matrix using a specific metric [6]. Among the techniques for constructing the coverage matrix, one of the most used approaches is block hit spectra [3]. Block hit spectra create a matrix that contains the code units executed during each test case, alongside a column vector that indicates the failing test cases. This structure allows for the comparison of the vector of each analyzed code unit with this last column to compute the suspiciousness score. This score is derived from a mathematical formula that quantifies the similarity between the corresponding vector for each code unit and the fault vector. Higher scores correlate with a greater probability of containing defects. The most common metrics are Jaccard [15], Tarantula [5], AMPLE [16,17], and Ochiai [3].

However, SBFL is not the only approach to solve the FL task. Other relevant methodologies include State-based Fault Localization, Model-based Fault Localization, Statistical-based Fault Localization, or Slice-based Fault Localization, among others [6]. We highlight Slice-based Fault Localization due to its ability to isolate faults within simplified versions of the original program. Essentially, Slice-based Fault Localization encompasses a set of techniques that facilitate fault localization by decomposing the program into smaller subprograms that preserve the original behavior [6], thus reducing the search space [18]. The first definition of a slice refers to any program derived from the original one by eliminating or modifying statements, while maintaining its functional behavior [19]. To address the main limitations of the original slices, more advanced variants have been developed, such as dynamic slices [20], which are focused on including more relevant statements in slices; execution slices [21], which extract statements that affect the target statement relative to specific input values; and observation-based slicing [22], which is a language-independent technique.

More recently, there has been a growing interest in Mutation-Based Fault Localization (MBFL) [7]. MBFL is based on mutation analysis, which consists of the generation of mutants from the Program Under Test through the application of diverse mutation operators. Subsequently, suspicious mutants are identified to guide the localization of potentially defective lines of code. Significant advances in this area include Metallaxis-FL, which uses mutants to identify defective lines [23], and MUSE, which leverages features from multiple mutant groups for fault localization, based on the observation that it is more likely to intro-

duce a defect by mutating a correct statement than a defective one [24]. Recent research has shown that higher-order mutants (i.e., those that have undergone multiple mutations) can enhance FL performance in comparison to first-order mutants (those obtained from a single mutation) [25], and it has also shown that reducing bias—defined as the discrepancy between the actual fault location and the fault location of the defect inserted by the mutant during its creation—improves the effectiveness of mutants in fault localization [7].

The rise of artificial neural networks has led to a shift in FL, making previous approaches less attractive to researchers, with emerging methods increasingly relying solely on neural network architectures. Initially, researchers explored the application of Convolutional Neural Networks [26] and Long-Short Term Memory (LSTM) networks [27,28] to improve results, an architecture-driven progress that mirrors trends in other recognition domains, where specialized designs such as graph convolutional networks have been proposed to sustain performance under data constraints [29]. A key advancement was the work of Meng et al. [28], who demonstrated improved performance by training LSTM networks with a novel approach that allows for FL and Automatic Program Repair. However, this approach was later superseded by the introduction of Transformer architecture in 2017, which led to the emergence of newer approaches that outperformed previous results [1,4,30]. This architecture, proposed by Vaswani et al. [8], incorporates an attention mechanism and parallelization capabilities that accelerate neural network training, rendering it a substantially more efficient solution than the prior approaches mentioned, leading to the development of Transformer-based solutions that outperformed state-of-the-art results. The original definition of Transformer architecture comprises two main components: an encoder, responsible for transforming a symbolic representation into a continuous representation, and a decoder, which transforms a continuous representation into a symbolic one [8]. Furthermore, a fundamental distinction exists between these approaches regarding their attention mechanism. An encoder exhibits bidirectional attention, allowing it to access the entire context, while a decoder employs causal attention, restricting its access solely to the preceding context of the token currently being processed within an input sequence [8]. This has significant implications, since these characteristics provide encoders with superior performance in classification tasks, whereas decoders excel in text generation.

As Transformer-based models gained presence, researchers began to evaluate their performance on new tasks. Initial findings indicated that decoder-based models were particularly effective for FL, demonstrating that models like GPT-3.5 and GPT-4 improved previous results [30]. It was also reported that including additional information about software execution enhances the performance of LMs in defect identification. This performance can be further improved by limiting the input length: LMs provide better results at the method level than at the class level. Based on these findings, LLMAO [4] sought to leverage decoders while retaining the beneficial features of encoders, specifically bidirectionality, to perform line-level, test-free FL and improve the state of the art. To this end, the authors developed a workflow that takes a source code snippet as input and returns a list of suspiciousness scores for each of its lines. In this workflow, the snippet is fed into CodeGen, a decoder-based language model, to extract the relationships between lines of code prior to obtaining each suspiciousness score that is used for classification purposes. Lines with higher suspiciousness scores are classified as defective, while lines with a suspiciousness score closer to zero will be considered non-defective. These relationships are captured, for each line, within the embeddings of the corresponding new line tokens; specifically, the final hidden states are used to compute the suspiciousness scores. However, since decoder-based architectures are not well suited to classification tasks, the authors added additional Transformer layers on top of CodeGen to build a minimal encoder. This encoder relates later lines in the source code to earlier ones, thereby preserving bidirectionality. This approach

has been further probed and compared to other architectures [1], and findings suggest that encoder-based and encoder–decoder-based LMs, even with smaller configurations, can outperform decoder-based LMs. However, this latter proposal, while effective, can be optimized in terms of complexity and computational efficiency: it employs a text-to-text approach, which is suboptimal for encoder-based models, since they are not well suited to text generation [31]. This step can be simplified to directly produce suspiciousness scores instead of asking the model to reconstruct the specific lines in order or priority, thereby avoiding additional errors caused by hallucinations in language models. In addition, using large datasets to train neural networks on FL introduces a new problem: class imbalance. Generally, source code contains more correct than defective lines and this, if not handled carefully, generates a bias in classification towards correct lines of code. Consequently, we propose an alternative architecture that corrects the bias caused by that imbalance and delivers a list of probabilities, mirroring the original LLMAO approach but utilizing encoders to enhance efficiency. This efficiency improvement stems from the removal of the text generation step and from the use of an architecture that, by design, implements bidirectionality. This allows all lines of code to incorporate the context of the entire source code without the need to add an additional encoder to the architecture.

3. Materials and Methods

To verify that encoders can more effectively solve the FL task, we test DeBERTa-v3-base [9,10] with the ConDefects benchmark. In Section 3.1, we describe the main specifications of ConDefects, why it was chosen and the preprocessing performed to extract the train, validation and test splits. Then, Section 3.2 presents DeBERTa-v3-base, which is the selected language model in this work, and the changes required in its architecture and hyperparameters used to adapt DeBERTa to line-level FL. In Section 3.3, we explain the replacement of newline characters (`\n`) with the new [NL] token, in addition to the down-sampling technique applied to improve results and the alternative methods we employed to verify the effectiveness of our approach. To ensure transparency and reproducibility, the hardware and software resources utilized to conduct the experiments are detailed in Section 3.4. Lastly, Section 3.5 explains the Top-N accuracy and ROC AUC metrics used to quantify the performance of the models, as well as the statistical protocol applied to report and compare the results.

3.1. Data Extraction and Processing

ConDefects [32] is the dataset employed in this work, a dataset specifically designed for the tasks of FL and Automated Program Repair. ConDefects emerged to address the problem of data leakage, which was an important concern since the widely adopted benchmarks used to evaluate LLM performance were created before the burgeoning of LLMs. To this end, the authors collected different programs from the AtCoder platform from October 2021 to September 2023 and curated them to have a single fault that spans one line of code per sample. In addition, each register is labeled with the creation date, which enables selection using a temporal window. The authors have been actively working on expanding the dataset, and they have added more registers up to June 2024 (<https://github.com/appmlk/ConDefects>, accessed on 2 October 2025).

The main reason for choosing this dataset is as follows: DeBERTa-v3 was trained on data from multiple textual corpora up to October 2021, including Wikipedia data [10], which are known to contain source code. This could potentially introduce a risk of data leakage; however, ConDefects includes only original source code first published in October 2021 or later, thereby minimizing this risk.

ConDefects contains 2879 faulty programs, of which 1254 are Java programs (with an average of 49.03 functions and 259.22 lines of code per sample) and 1625 are Python programs with 2.91 functions and 49.03 lines of code per script on average. These programs are structured hierarchically: directories follow the pattern `abcXXX_Y`, where `abc` stands for AtCoder Beginner Contest, `XXX` represents the contest number, and `Y` the specific problem. Within each directory, multiple participant submissions are stored. Problems within a contest are generally ordered by increasing difficulty, with only rare exceptions marked as low difficulty when the exact level is unavailable. To leverage this structure while preventing data leakage, we constructed a customized Python-only subset and applied a chronologically ordered, contest-level time split. We iterated through the contests sequentially from oldest to newest, accumulating all source submissions to construct a 70% training, 10% validation and 20% testing partition, while ensuring that all solutions to the same problem are kept within a single partition. The validation set, taken chronologically from the most recent portion of the training block, is used exclusively for model selection (in particular, for choosing the downsampling ratio), whereas the test set remains untouched until the final evaluation. In addition, exact duplicate submissions are removed before training and evaluation. This time-based strategy guarantees that all similar solutions to a given problem remain confined to a single split, effectively eliminating leakage from homologous code fragments. Moreover, this chronological split inherently preserves a balanced difficulty distribution across both partitions, as each contest naturally features a standardized mix of problems with varying difficulty levels. While the temporal separation from the cutoff of DeBERTa-v3-base already mitigates contamination, this time-based strategy provides a rigorous safeguard against cross-contamination between similar submissions. The splitting procedure is fully deterministic, eliminating the need for random seed control during data partitioning.

For evaluation, we used three different sets of programs: (1) the aforementioned test partition to test the model's performance on Python code, (2) ConDefects' Java set to test the model performance on languages other than Python, and (3) the entire ConDefects dataset (except the training and validation data) to obtain a comprehensive view of the data.

3.2. Model Architecture and Training Setup

In this work, DeBERTa-v3-base [9,10] has been employed as the classifier. This encoder-based model has a reduced size of 86M backbone parameters, which makes it highly efficient in terms of computational resources. To adapt DeBERTa-v3-base to the classification task, several modifications were made to both the architecture and the training process. Regarding the former, three additional linear layers were added to improve its classification capabilities and to return a list of suspiciousness scores for each statement (line) inside the analyzed program. The first layer takes an input vector of dimension 768, corresponding to the hidden size of DeBERTa-v3-base. To compress this representation into two classes, we set the output size of the first layer to 512, the second to 256, and the third to 1 (a single output dimension enables binary classification). The first two layers employ a GELU activation function, as it is standard in Transformer architectures and aligns with the output activation used by DeBERTa-v3-base. The final layer has a sigmoid activation function to return the probability associated with each LOC. These additional layers allow the model to capture data nonlinearities without significantly increasing the number of parameters, adding approximately 350,000 parameters to the model. The final representation can be found in Figure 1, where these components are shown alongside the output size of each component. The head configuration is detailed in Table 1, which also lists other parameters and hyperparameters employed during the training process.

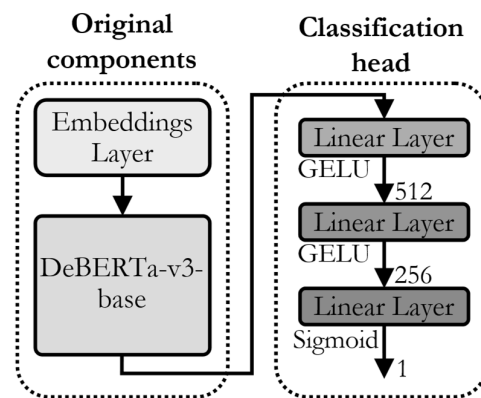


Figure 1. Model architecture. Original DeBERTa-v3-base components (**left**) and the proposed head (**right**) used to calculate suspiciousness scores based on [NL] tokens following the bidirectional attention-based encoding process.

Table 1. Hyperparameters used during training.

Parameter	Value
Context length	4096
Lines per code chunk (approx.)	~200
Special tokens added	[NL] (ID: 128001)
Final vocabulary size	128002
Classification head input size	768
Classification head hidden size (Layer 1)	512
Classification head hidden size (Layer 2)	256
Classification head output dimension	1
Activation function	GELU
Batch size	16
Gradient accumulation steps	16
Effective batch size	256
Training epochs	7
Learning rate	2.50×10^{-5}
Learning rate scheduler	Linear decay
Warmup steps	0
Optimizer	8-bit AdamW (bitsandbytes)
AdamW epsilon	1.00×10^{-6}
Weight decay	0.01
Parameters exempt from weight decay	Bias terms and LayerNorm weights
Early stopping	Disabled
Loss function	Binary Cross-Entropy Loss with Logits
Random seeds	0–9 (one independent run per seed)

One important hyperparameter is the context length, which determines the maximum number of tokens the model can process. We set this value to 4096 tokens (Table 1), which allows for a context of approximately 200 lines of code. We selected this value because it enables us to process most of the code in the dataset without the need to divide it into

smaller chunks. In ConDefects, only 91 programs exceed 200 lines of code, which represents a minority (3%). Nevertheless, the DeBERTa family is pretrained to handle inputs up to 512 tokens [9], although it supports longer contexts by leveraging disentangled attention and relative positional encoding. When this pretraining limit is exceeded, computational efficiency begins to degrade. To avoid this, we maintain a balance between the number of lines of code that can be processed and the performance of the model. In this scenario, codes exceeding 200 lines are fragmented into blocks, consisting of 200 lines each. This enables a wider context at the cost of potential performance degradation, since critical contextual information required to understand bugs may be lost.

The remaining training hyperparameters were selected following established protocols for fine-tuning Transformer-based language models. We set the batch size to 16 samples with 16 gradient accumulation steps to achieve an effective batch size of 256 samples, thereby optimizing computational resource usage. We trained the model for 7 epochs to allow it to learn the data distribution while avoiding overfitting.

A base learning rate of 2.5×10^{-5} and a linear schedule are widely adopted in BERT fine-tuning pipelines, providing a stable trade-off between convergence speed and the avoidance of catastrophic forgetting.

Regarding the warmup strategy, we set the number of warmup steps to zero. This decision stems from our use of a relatively small dataset, as we wanted to avoid dedicating training samples to a warmup procedure. Furthermore, given the use of a stable pretrained model and a small learning rate, this process is less critical in our context.

For optimization, we employ 8-bit AdamW (via bitsandbytes) to reduce memory overhead without measurable accuracy loss. The AdamW epsilon parameter is set to 1×10^{-6} for consistent numerical stability in the adaptive denominator, aligning with common implementations in efficient Transformer fine-tuning. Regularization is handled via a weight decay of 0.01, a standard value for AdamW, with bias terms and LayerNorm weights explicitly exempted from decay. This exemption follows established practice, as these parameters are small, highly stable, and sensitive to weight decay, which can otherwise degrade optimization dynamics or slow convergence.

Finally, to assess the robustness of the results, each configuration reported in this study was trained multiple times using different random seeds to control all stochastic processes, including model head initialization, data shuffling, and downsampling. Ten seeds were used for the main configurations and five for the rest of the experiments. Aggregate statistics across seeds are reported as described in Section 3.5.

3.3. Data Preprocessing and Downsampling Strategy

In addition to the prior processing for lengthy codes, in which we split those exceeding 200 lines into chunks of this length, every code fragment undergoes a specific treatment related to line breaks. Specifically, newline characters (`\n`) are replaced by a new special token [NL] designed to capture the semantics of the entire line in relation to the rest of the code. This token is not initially present in the vocabulary of the tokenizer or the pretrained model; rather, during our training process, both the tokenizer and the model learn to identify and use [NL] for inference. The ID assigned to this new token, along with the final vocabulary size, is detailed in Table 1. Then, after the language model processes the embeddings extracted from a code snippet, we take the final hidden states corresponding to [NL] tokens that encode the relationship of each LOC with the rest of the code fragment. These embeddings are then processed by additional layers and a sigmoid activation function to yield feasible suspiciousness scores. Figure 2 shows the complete training and inference processes.

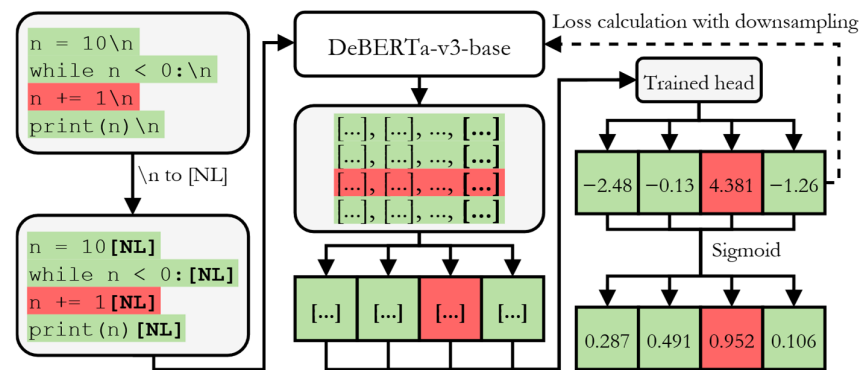


Figure 2. Training and inference processes. First, line breaks are replaced with the [NL] token. Then, the model receives the modified source code to return the final hidden states for its embeddings, from which the embeddings corresponding to [NL] tokens are selected and the rest of the embeddings are discarded. These data are further processed by the three additional layers, which return the final logits. Downsampling is applied to the logits during training, and a sigmoid activation is applied during inference. Green elements represent correct lines of code or their respective embeddings, while red elements represent defective lines or their respective embeddings. Each “[...]” depicts the embeddings corresponding to each token in the input.

Furthermore, a significant class imbalance was identified. Generally, a program contains more lines of correct code than defective lines, which is also reflected in ConDefects. Specifically, the Python set contains 2864 defective lines and 138,526 correct lines, which implies a ratio of approximately 48 correct lines per defective line.

Consequently, a balancing technique based on downsampling is applied during training where some random values that correspond to correct lines are disregarded during loss calculation following a predefined ratio. Specifically, we allow DeBERTa-v3-base to process all lines of code so that the [NL] tokens can capture the relationship with the rest of the code. Then, after the classification head compresses the representation from DeBERTa-v3-base, we compute the loss in four steps, at which point downsampling is applied: (1) we first apply Binary Cross-Entropy Loss with Logits (BCEWithLogitsLoss; Table 1) to the raw logits with reduction set to ‘none’ (note that the sigmoid activation is only applied for inference), (2) immediately after, we create a binary mask that selects the loss values for defective lines within the batch and for N non-defective lines, based on a ratio relative to the number of defective lines, (3) we apply the mask, thereby retaining the selected loss values and setting the remaining ones to zero, and (4) the average is computed considering only this newly selected set of values. This approach yields a more stable loss and reduces bias toward the majority class.

For this study, we use N ratios of $2\times$ and $3\times$ the number of defective lines; thus, if a batch contained 16 defective lines, we would select 32 or 48 non-defective lines depending on the specified ratio. Section 4 delves into the different ratios applied to downsampling and their impact in the effectiveness of classification.

To determine if downsampling is a better choice than other common class imbalance techniques, we evaluated two alternatives that operate on the loss function while maintaining the original data distribution: (1) class-weighted BCE, in which the loss contribution of defective lines is scaled by the ratio of non-defective to defective lines within each batch and (2) focal loss [11], with a focusing parameter of $\gamma = 2$ and a class-balancing factor of $\alpha = 0.75$ assigned to the minority (defective) class. The value of γ follows the recommendation of the original work [11], while α is set to favor the minority class according to the usual interpretation of this factor as a class-frequency weight. We did not perform an exhaustive search over these hyperparameters, as the goal of this comparison is to contrast

representative configurations of each strategy rather than to optimize them individually. Section 4 compares these strategies with downsampling.

3.4. Hardware and Runtime Details

All experiments were conducted on a dedicated server equipped with four NVIDIA H100 (94 GB HBM3) GPUs connected via NVLink, dual AMD EPYC Genoa 6934 processors (84 cores/168 threads, 2.25 GHz base frequency, 384 MB cache), and 768 GB of DDR5 system memory. The training infrastructure runs on Ubuntu 24.04.4 LTS, leveraging CUDA 12.6, PyTorch 2.4.0, and transformers 4.44.2. Training was executed on a single GPU within the server, with a total runtime of approximately 20 min.

3.5. Evaluation Metrics

The performance of our trained classifiers has been measured using two widespread metrics in the FL area: Top-N accuracy and Area Under the ROC Curve (ROC AUC). All metrics are reported as the mean and standard deviation across multiple independent random seeds, and statistical significance is evaluated using two-sided paired *t*-tests and Wilcoxon signed-rank tests.

3.5.1. Top-N

Top-N accuracy computes the rate of instances for which the ground truth is found within the first N positions of the list of probabilities predicted by the model (sorted in descending order) for every sample in the dataset. In this study, the values of N used were 1, 3 and 5, which are aligned with previous approaches and practitioners' expectations [2,4]. In this study, we adapt the metric as follows. Based on the suspiciousness scores obtained for each line, we rank the line numbers in descending order of suspiciousness. We then select the top N line numbers according to this ranking and check whether the line number corresponding to the defective line (ground truth) is included among them.

3.5.2. ROC AUC

The Area Under the ROC Curve summarizes the performance of a classifier using a single value, obtained from a previously computed ROC Curve. Values range from 0.5 to 1, where 0.5 indicates a completely random classifier, a value between 0.5 and 0.7 suggests a slightly better prediction but close to the random guess, and values greater or equal to 0.7 are considered indicative of an effective classifier, with 1 representing a perfect classifier [33].

3.5.3. Statistical Protocol

Since every configuration is trained with several independent random seeds, all metrics are reported as the mean and standard deviation across seeds. Pairwise comparisons between configurations trained with the same set of seeds are assessed with two-sided paired *t*-tests, complemented with Wilcoxon signed-rank tests [34] as a non-parametric alternative [35], using a significance level of 0.05.

4. Results and Discussion

Regarding the results, Figure 3 shows the evolution of the ROC AUC values obtained from the Python test set for different downsampling ratios that were selected using the Python validation set, where each point corresponds to the mean over five independent random seeds, and the shaded band represents one standard deviation. To keep the computational cost of this analysis manageable, the models were trained with a reduced context length of 1024 tokens. The configurations selected were then retrained with the full 4096-token context specified in Table 1. The ratios, ranging from left to right, start at

1:1, where we pick one correct line per defective line for every code snippet used during training. This ratio increases, reaching the original data distribution (Orig.), where each entire snippet is used during training. As demonstrated in Figure 3, performance peaks at a ratio of 2:1 (validation ROC AUC of 0.862) and remains within a stable plateau up to approximately 4:1. From that point onwards, the ROC AUC values slowly decrease, converging towards the performance obtained with the original data distribution (0.746). Considering these results, we select for further analysis and comparison one encoder trained with the original data distribution, one encoder trained with a ratio of 2:1, and another encoder trained with a ratio of 3:1.

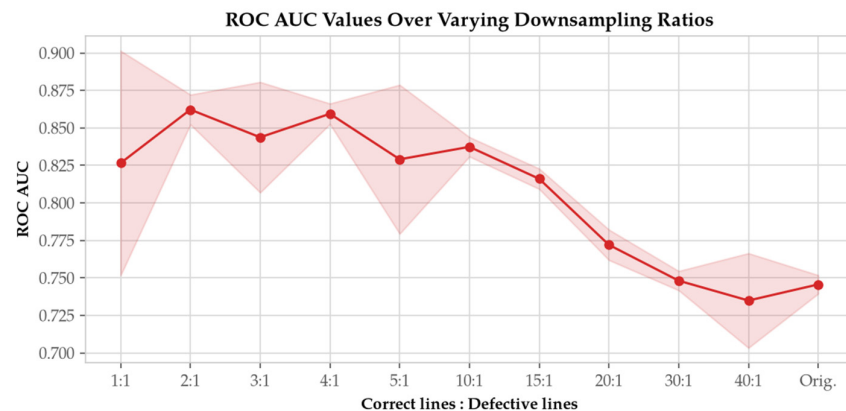


Figure 3. Evolution of the Area Under the ROC Curve on the validation set for different down-sampling ratios (mean $\pm$ standard deviation over five random seeds). The ratio is represented by “Correct Lines (considered during training) to Defective Lines”. The remaining lines in each code were discarded during the loss calculation. Initially, a 1:1 ratio is used, and the number of correct lines is incremented by one unit until a ratio of 5:1. From that point onwards, the increment is in batches of five up to a 20:1 ratio, and later, in batches of 10, up to the original data distribution. The red line shows the average across five random seeds, and the shaded area shows one standard deviation.

The mentioned comparison is presented in Table 2, which illustrates the three different scenarios that correspond to the experiments conducted tested on the Python subset. Each configuration was trained ten times with different random seeds, and the table reports the mean and standard deviation of the Top-N accuracy and the Area Under the ROC Curve for every scenario. The first row corresponds to the results obtained using the encoder trained with the original data distribution (no class balancing technique applied), while the second and third rows correspond to the results obtained using the encoders trained with a ratio of 2:1 and 3:1, respectively. Figure 4 complements the table with the corresponding ROC curves on the Python test set, plotted as the mean true positive rate across seeds with a one-standard-deviation band.

As shown in Table 2, the implementation of class balancing techniques improves model performance considerably. The mean ROC AUC increased from 0.737 to 0.840 (an absolute gain of 0.10). Top-1 accuracy nearly doubled, increasing from 0.095 to 0.185. Additionally, Top-3 and Top-5 accuracies improved by approximately 64% and 50%, respectively. These differences are statistically significant for every metric (paired t -test, $p < 10^{-5}$; Wilcoxon signed-rank test, $p < 0.01$). However, the differences between the 2:1 and 3:1 ratios are not statistically significant for any metric (paired t -test, $p \geq 0.23$; Wilcoxon, $p \geq 0.29$). This indicates that both configurations lie within the stable plateau identified in Figure 3. Since the 2:1 ratio attains the highest mean validation ROC AUC, it is adopted as the reference configuration for the remaining analyses. It is also worth noting that although the classifier trained on the original distribution retains some discriminative ability in terms

of the ROC AUC, its practical usefulness is limited, as reflected by its substantially lower top-N accuracy.

Table 2. The corresponding results (mean $\pm$ standard deviation over ten random seeds) to the three experiments conducted in this study, evaluated on the Python test set. The first scenario applies no downsampling, whereas the second scenario applies a ratio of two correct lines of code per defective LOC and the third a ratio of three correct lines of code per defective LOC. The highest mean scores are highlighted in bold.

Model	Top-1	Top-3	Top-5	ROC AUC
DeBERTa-v3-base Orig.	0.0946 $\pm$ 0.0103	0.2616 $\pm$ 0.0336	0.3747 $\pm$ 0.0367	0.7365 $\pm$ 0.0167
DeBERTa-v3-base 2:1	0.1852 $\pm$ 0.0245	0.4300 $\pm$ 0.0384	0.5623 $\pm$ 0.0364	0.8400 $\pm$ 0.0123
DeBERTa-v3-base 3:1	0.1923 $\pm$ 0.0307	0.4372 $\pm$ 0.0439	0.5670 $\pm$ 0.0424	0.8378 $\pm$ 0.0187

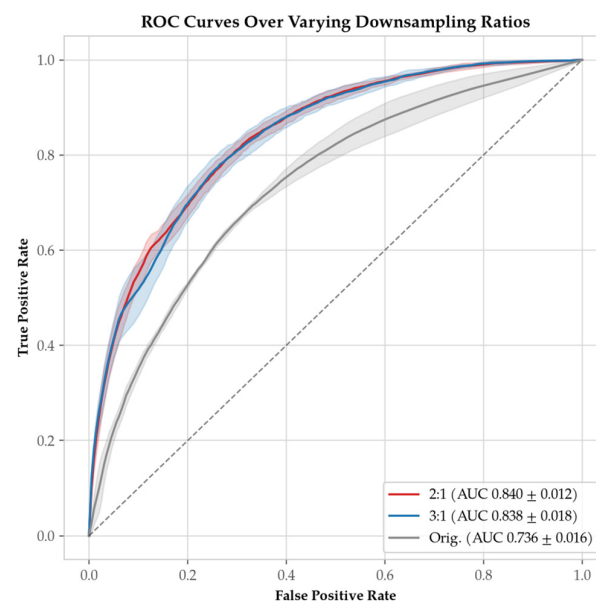


Figure 4. ROC curves on the Python test set for the three main configurations. Each curve corresponds to the mean true positive rate across ten random seeds, and the shaded band represents one standard deviation. The dashed line represents a random classifier.

Two additional analyses were conducted beyond the main configurations to isolate the contribution of each component of the proposed approach. First, to verify that the reported improvements are not specific to DeBERTa-v3-base and to compare our choice against encoders pretrained on source code specifically, we repeated the experiments with CodeBERT [12], GraphCodeBERT [13], and RoBERTa [14]. Since these three models only support 512 positional embeddings, we trained all the encoders in this comparison, including DeBERTa-v3-base, with a context length of 512 tokens. This guaranteed identical conditions. We used five random seeds per configuration and kept every other hyperparameter unchanged. The results are summarized in Table 3. Two findings emerge. First, the downsampling strategy benefits all four architectures, meaning every encoder substantially improves when transitioning from the original distribution to the 2:1 or 3:1 ratios. This indicates that the training strategy is architecture-agnostic rather than a DeBERTa artifact. Second, under identical conditions, DeBERTa-v3-base consistently outperforms the code-specific encoders (e.g., ROC-AUC of 0.8303 vs. 0.8082 for Graph CodeBERT at 2:1 with a p -value of ≤ 0.011 against every alternative in both ratios), and the ranking of the four models remains consistent across both downsampling ratios. We attribute this advantage

to DeBERTa’s disentangled attention mechanism, relative positional encoding, and more recent pretraining procedure.

Table 3. Encoder comparison under identical conditions (512-token context, mean $\pm$ standard deviation of the test ROC AUC over five random seeds). The highest mean scores are highlighted in bold.

Encoder	Orig.	2:1	3:1
DeBERTa-v3-base	0.6821 $\pm$ 0.0288	0.8303 $\pm$ 0.0076	0.8072 $\pm$ 0.0122
GraphCodeBert [13]	0.6350 $\pm$ 0.0196	0.8082 $\pm$ 0.0046	0.7857 $\pm$ 0.0077
CodeBERT [12]	0.6506 $\pm$ 0.0301	0.7979 $\pm$ 0.0039	0.7719 $\pm$ 0.0093
RoBERTa [14]	0.6491 $\pm$ 0.0184	0.7947 $\pm$ 0.0058	0.7573 $\pm$ 0.0154

Second, we compared the proposed downsampling strategy with two loss-based class imbalance techniques: class-weighted binary cross-entropy (BCE) and focal loss. We trained these techniques on the original data distribution with five random seeds each and a 1024-token context. This setting matches that of the ratio sweep in Figure 3, whose original distribution and 2:1 runs serve as references. The results are presented in Table 4. Weighted BCE performs similarly to 2:1 downsampling (test ROC-AUC of 0.8468 vs. 0.8416, a paired *t*-test yielded $p \geq 0.08$ for every metric). However, focal loss clearly improves upon the unbalanced baseline yet remains below both alternatives (0.8127 $\pm$ 0.0226). These results suggest that rebalancing the contribution of both classes during training is decisive, regardless of the mechanism, and that downsampling is a competitive choice that reduces the number of loss terms computed per batch.

Table 4. Comparison of class imbalance handling strategies (1024-token context, mean $\pm$ standard deviation over five random seeds). The highest mean scores are highlighted in bold.

Strategy	Top-1	Top-3	Top-5	ROC AUC
None (plain BCE)	0.0995 $\pm$ 0.0133	0.2653 $\pm$ 0.0280	0.3805 $\pm$ 0.0265	0.7405 $\pm$ 0.0036
Focal loss ($\gamma = 2$, $\alpha = 0.75$)	0.1602 $\pm$ 0.0372	0.3822 $\pm$ 0.0374	0.5155 $\pm$ 0.0359	0.8127 $\pm$ 0.0226
Weighted BCE	0.1913 $\pm$ 0.0141	0.4325 $\pm$ 0.0173	0.5515 $\pm$ 0.0131	0.8468 $\pm$ 0.0033
Downsampling 2:1	0.1930 $\pm$ 0.0104	0.4373 $\pm$ 0.0186	0.5627 $\pm$ 0.0119	0.8416 $\pm$ 0.0094

To the best of our knowledge, this is the first study to evaluate a fault localization scenario based on an encoder architecture using the ConDefects dataset. However, we note that the authors of the mentioned dataset presented two initial evaluations for the FL task using ConDefects: one that obtained the results using SBFL with the Ochiai metric on the Java subset [32] and another one that utilized GPT-3.5 and GPT-4 through prompting techniques on 50 randomly selected registers [36]. In our study, 70% of the data contained in the ConDefects Python subset was used to train our encoder-based classifier and 10% was reserved for validation. For evaluation, we applied three different subsets: the remaining 20% of the ConDefects Python subset, the entire ConDefects Java partition, and the complete ConDefects test data, which combine both. The first scenario is used to assess the performance of the model on the original Python-based dataset, whereas the other two scenarios evaluate its generalization ability to other languages (Java) and its performance on the full dataset, which includes samples from both languages.

As shown in Table 5, the best accuracy results reported in the literature are obtained using GPT-4 with prompting techniques. We acknowledge that DeBERTa-v3-base fine-tuned for FL operates on a different architecture, employs distinct fine-tuning methodologies, and is evaluated on a separate dataset. Nevertheless, it yields promising results that demonstrate competitive potential against significantly larger proprietary models like GPT-4,

particularly when adapted to broader scenarios. Importantly, this approach achieves comparable predictive quality while requiring a fraction of the computational resources and eliminating reliance on closed-source inference. This efficiency advantage is particularly evident on the Python subset, where the model attains its strongest performance.

Table 5. A general comparison between our experiments and related work. The results are reported under different experimental settings. The first row corresponds to the results reported for the LLMAO model with the Defects4J dataset, the three subsequent rows correspond to the results reported with SBFL with the Ochiai metric, GPT-3.5 and GPT-4 using ConDefects, and the last three rows represent our results using ConDefects (mean $\pm$ standard deviation over ten random seeds, 2:1 configuration). Further evaluation using a unified benchmark would be required for direct comparison. The highest mean scores are highlighted in bold.

Model	Dataset	Top-1	Top-3	Top-5	ROC AUC
LLMAO Defects4J (Yang et al.) [4]	Defects4J (Java)	0.2230	0.3770	0.4630	0.677
LLMAO BugsInPy (Yang et al.) [4]	BugsInPy (Python)	0.1030	0.1200	0.1520	0.571
ConDefects Ochiai (Wu et al.) [32]	ConDefects (Java)	0.0572	0.2034	0.3686	—
ConDefects GPT-3.5 (Wu et al.) [36]	ConDefects (50 registers)	0.2000	0.3680	0.4080	—
ConDefects GPT-4 (Wu et al.) [36]	ConDefects (50 registers)	0.3280	0.5280	0.6000	—
DeBERTa-v3-base (ours)	ConDefects (Python test split)	0.1852 $\pm$ 0.0245	0.4300 $\pm$ 0.0384	0.5623 $\pm$ 0.0364	0.8400 $\pm$ 0.0123
DeBERTa-v3-base (ours)	ConDefects (Java)	0.1794 $\pm$ 0.0348	0.3317 $\pm$ 0.0333	0.4112 $\pm$ 0.0258	0.7396 $\pm$ 0.0166
DeBERTa-v3-base (ours)	ConDefects (except training and validation sets)	0.1807 $\pm$ 0.0309	0.3532 $\pm$ 0.0324	0.4443 $\pm$ 0.0263	0.7644 $\pm$ 0.0140

Regarding the LLMAO architecture, initial evaluations on the Defects4J dataset reported Top-N accuracy metrics comparable to our results. However, a direct comparison remains challenging due to differences in datasets, programming languages, sample sizes and experimental protocols. Despite these methodological variations, we have adopted the ROC AUC values reported for LLMAO as a contextual reference to facilitate a rough comparative assessment. Replicating our experimental setup using Defects4J or BugsInPy could be a potential solution to ease this comparison. Nonetheless, at present, there are no established methods to apply the Top-N accuracy metric in a multi-line classification setting. In both datasets, defects are not confined to a single line but may span multiple lines, which makes it infeasible to apply Top-N accuracy in the same way as in our approach.

In their study, all classifiers trained on Defects4J or BugsInPy reported ROC AUC values below 0.7 (0.677 for Defects4J and 0.571 for BugsInPy), whereas our best classifier achieves a mean ROC AUC of 0.8400 ± 0.0123 , which suggests a higher suitability for this task. Focusing on the Python scenario, although a different dataset is used, our approach yields consistent higher values across all evaluated metrics, with absolute gains of 0.0822 for Top-1, 0.3100 for Top-3, 0.4103 for Top-5, and 0.2690 in ROC AUC. When they train and evaluate their workflow on the Devign dataset, a considerable larger collection of C programs [37], the authors report a ROC AUC of 0.855 using Codegen-16B, which represents a significant improvement over the other scenarios. This performance is achieved with a model containing approximately 16 billion parameters, which is nearly 200 times larger

than DeBERTa-v3-base, the model used in our experiments. From an efficiency perspective, our approach attains comparable results, and even superior ones in some scenarios, while using a smaller model. However, direct comparisons on the same benchmark are needed to fully validate these advantages.

Error Analysis

To better understand when and why the model fails, we analyzed the predictions of a representative 2:1 model (seed 0, whose aggregate metrics closely match the means in Tables 2 and 5) on the Python test set and the Java set. For every program, we computed the rank of the defective line within the suspiciousness ranking, grouped programs by length, and assigned each defective line and each Top-1 prediction a broad syntactic category derived from lexical rules applicable to both languages.

Table 6 shows that performance consistently degrades with program length. The Top-5 hit rate drops from 0.628 for programs under 50 lines to 0.312 for programs between 100 and 200 lines in Python, and from 0.675 to 0.031 for programs exceeding 200 lines in Java. The median rank grows from 3 to 226. This degradation mainly comes from the segmentation strategy since programs exceeding 200 lines are split into chunks that are scored in independent forward passes. Thus, the resulting scores are not calibrated against each other, and each chunk lacks the context of the remaining code. In contrast, the case in which the defective line receives no score because its chunk exceeds the 4096-token limit, resulting in the line being cut off by the tokenizer, is marginal. This occurs in only three programs across both sets. This analysis also clarifies the Python-to-Java drop reported in Table 5. For programs under 50 lines, Java (0.675) performs similarly to Python (0.628), even though the model was fine-tuned exclusively on Python. However, 23% of Java programs exceed 200 lines, compared to only 3% of Python programs. Therefore, the cross-language degradation is mainly attributable to program length and the associated segmentation penalty rather than the language change itself.

Table 6. Top-5 hit rate and median rank of the defective line by program length (2:1 configuration). Columns “n” report the number of programs in each bucket.

Length (Lines)	Python Top-5	Python Median Rank	Python n	Java Top-5	Java Median Rank	Java n
0–50	0.628	3.0	446	0.675	3.0	197
50–100	0.506	5.0	77	0.353	8.5	68
100–200	0.312	12.0	32	0.104	42.0	48
>200	0.389	9.5	18	0.031	226.0	96

The syntactic profile of defects reveals that faults in conditional statements are the easiest to localize (Top-5 hit rate of 0.770 in Python), followed by output statements (0.738) and loop headers (0.675). Faulty assignments (0.539) and general expression statements (0.214), however, are more difficult to locate. Notably, none of the 13 Python defects located in lines that read program input were ranked within the top 5. The predictions also reveal a consistent bias. In Python, 47.6% of the Top-1 predictions are conditional statements, though they account for only 19.7% of the actual defects. Manual inspection of the worst-ranked cases confirms this pattern. The model focuses on algorithmically dense lines, such as nested conditions or expressions with heavy arithmetic and indexing. However, many real defects reside in seemingly innocuous lines, such as those that read input or set loop bounds. The correctness of these lines depends on the problem specification, which the model never observes. In very short programs, an additional pattern emerges: all scores are close to the decision boundary, and the defective line is often ranked second.

5. Conclusions

This study addressed the problem of FL through the application of encoder-based LMs. The adoption of the Transformer architecture has led to innovative solutions in this research area, with LLMAO being the main reference due to its ability to identify faults without the need for test cases, unlike most traditional approaches. Following that research line, this work has focused on adapting and extending this solution, originally implemented with decoders, to encode architectures that are more suitable for classification tasks.

As a result, we fine-tuned DeBERTa-v3-base on the ConDefects dataset for line-level FL to enhance classification performance. During the training process, a challenge related to class imbalance was identified, for which a downsampling strategy based on the exclusion of code lines during loss calculation was proposed. This technique has proven to be effective and improved the performance of our classifier, achieving, over ten random seeds, a mean Top-1 accuracy of 0.1852, Top-3 accuracy of 0.4300, Top-5 accuracy of 0.5623, and ROC AUC score of 0.8400 on the Python test partition.

The benefit of downsampling generalizes to CodeBERT, GraphCodeBERT and RoBERTa, with DeBERTa-v3-base remaining the strongest classifier under identical conditions. In addition, downsampling performs on par with class-weighted loss and outperforms focal loss, confirming it as a competitive and efficient choice for handling class imbalance in this task.

This experimental setup suggests competitive results against other approaches when performing cross-dataset comparisons. However, ConDefects is primarily composed of defective solutions to competitive programming problems written in Java and Python; hence, the code samples are generally smaller, more self-contained, and less complex than those found in real-world software systems. In addition, the benchmark size may limit the diversity of defect patterns available during fine-tuning, potentially affecting the robustness and generalization capabilities of LLMs. Therefore, while the reported results demonstrate the effectiveness of the proposed approach on ConDefects, its performance on other fault localization benchmarks, additional programming languages, and industrial-scale software projects remains an open question. Nevertheless, because our approach relies on contextual line-level embeddings and a test-suite-free training protocol, it holds strong potential for cross-domain transfer. Future work will focus on validating the method on diverse open-source repositories and exploring lightweight domain adaptation strategies to ensure robustness across broader software ecosystems. This validation would enable using smaller, encoder-based LMs to deliver robust fault localization without the computational cost overhead of larger LMs and closed-source inference.

Two additional areas for improvement have been identified. First, FL using neural networks is limited by the amount of context a language model can process. This restriction imposes a constraint on the length of code snippets that can be analyzed, which may affect the quality of predictions. Although decomposing lengthy source codes in blocks of 200 lines offers a viable solution for extended contexts, it would be ideal to analyze entire programs, which would not only improve results but also facilitate the exploration of wider granularities. Finally, Top-N accuracy, as implemented in this work, is restricted to single-line FL scenarios. To extend the use of neural networks to defects spanning multiple lines, it is necessary to develop an analogous Top-N metric capable of evaluating the performance of the model for multiple targets. This new metric would enable a direct and rigorous comparison with previous research that employs the original Top-N accuracy metric.

Author Contributions: J.-A.d.-H.-G.: Writing—review & editing, Writing—original draft, Validation, Methodology, Investigation, Conceptualization. A.G.-C.: Writing—review & editing, Writing—original draft, Funding acquisition, Project Administration, Supervision, Methodology,

Investigation, Conceptualization. E.G.-L.: Writing—review & editing, Writing—original draft, Funding acquisition, Project Administration, Supervision, Methodology, Investigation, Conceptualization. D.d.-F.-D.: Writing—review & editing, Writing—original draft, Validation, Methodology, Investigation, Conceptualization. All authors have read and agreed to the published version of the manuscript.

Funding: The APC was partially funded by the project “Reparación automática de código fuente mediante modelos generativos de Procesamiento de Lenguaje Natural” (SBPLY/23/180225/000063, cofunded by Junta de Comunidades de Castilla-La Mancha and Programa Operativo Feder de Castilla-La Mancha).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: The original contributions presented in the study are included in the article, further inquiries can be directed to the corresponding author.

Acknowledgments: The authors want to thank the support provided by the projects “Reparación automática de código fuente mediante modelos generativos de Procesamiento de Lenguaje Natural” (SBPLY/23/180225/000063, cofunded by Junta de Comunidades de Castilla-La Mancha and Programa Operativo Feder de Castilla-La Mancha); “Tecnologías Inteligentes para la Fabricación, el diseño y las Operaciones en entornos iNdustriales” (TIFON, PLEC2023-010251), and “Detección de Errores y Vulnerabilidades en el Software con Autoencoders Dispersos” (PIUAH25/IA-026). The first author and the last author also want to thank the support received by the Ministerio de Ciencia, Innovación y Universidades para la Formación de Profesorado Universitario (FPU24/03370 and FPU23/01120, respectively).

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Ji, S.; Lee, S.; Lee, C.; Han, Y.-S.; Im, H. Impact of large language models of code on fault localization. In *Proceedings of the 2025 IEEE Conference on Software Testing, Verification and Validation (ICST), Napoli, Italy, 31 March–4 April 2025*; IEEE: Piscataway, NJ, USA, 2025; pp. 302–313. [\[CrossRef\]](#)
2. Kochhar, P.S.; Xia, X.; Lo, D.; Li, S. Practitioners’ expectations on automated fault localization. In *Proceedings of the 25th International Symposium on Software Testing and Analysis, Saarbrücken, Germany, 18–20 July 2016*; ACM: New York, NY, USA, 2016; pp. 165–176. [\[CrossRef\]](#)
3. Abreu, R.; Zoetewij, P.; Van Gemund, A. An evaluation of similarity coefficients for software fault localization. In *Proceedings of the 2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC’06), Riverside, CA, USA, 18–20 December 2006*; IEEE: Piscataway, NJ, USA, 2006; pp. 39–46. [\[CrossRef\]](#)
4. Yang, A.Z.H.; Le Goues, C.; Martins, R.; Hellendoorn, V. Large language models for test-free fault localization. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering, Lisbon, Portugal, 14–20 April 2024*; ACM: New York, NY, USA, 2024; pp. 1–12. [\[CrossRef\]](#)
5. Jones, J.A.; Harrold, M.J. Empirical evaluation of the tarantula automatic fault-localization technique. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE ’05), Long Beach, CA, USA, 7–11 November 2005*; Association for Computing Machinery: New York, NY, USA, 2005; pp. 273–282. [\[CrossRef\]](#)
6. Wong, W.E.; Gao, R.; Li, Y.; Abreu, R.; Wotawa, F. A Survey on Software Fault Localization. *IEEE Trans. Softw. Eng.* **2016**, *42*, 707–740. [\[CrossRef\]](#)
7. Liu, H.; Li, Z.; Han, B.; Liu, Y.; Chen, X.; Liu, Y. Delta4Ms: Improving mutation-based fault localization by eliminating mutant bias. *Softw. Test. Verif. Reliab.* **2024**, *34*, e1872. [\[CrossRef\]](#)
8. Vaswani, A.; Shazeer, N.; Parmar, N.; Uszkoreit, J.; Jones, L.; Gomez, A.N.; Kaiser, Ł.; Polosukhin, I. Attention is All you Need. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2017. Available online: <https://proceedings.neurips.cc/paper/2017/hash/3f5ee243547dee91fbd053c1c4a845aa-Abstract.html> (accessed on 25 March 2025).
9. He, P.; Liu, X.; Gao, J.; Chen, W. DeBERTa: Decoding-enhanced BERT with disentangled attention. *arXiv* **2021**. [\[CrossRef\]](#)
10. He, P.; Gao, J.; Chen, W. DeBERTaV3: Improving DeBERTa using ELECTRA-style pre-training with gradient-disentangled embedding sharing. *arXiv* **2021**. [\[CrossRef\]](#)
11. Lin, T.-Y.; Goyal, P.; Girshick, R.; He, K.; Dollár, P. Focal Loss for Dense Object Detection. *arXiv* **2018**, arXiv:1708.02002. [\[CrossRef\]](#)

12. Feng, Z.; Guo, D.; Tang, D.; Duan, N.; Feng, X.; Gong, M.; Shou, L.; Qin, B.; Liu, T.; Jiang, D.; et al. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*; Association for Computational Linguistics: Stroudsburg, PA, USA, 2020; pp. 1536–1547. [\[CrossRef\]](#)
13. Guo, D.; Ren, S.; Lu, S.; Feng, Z.; Tang, D.; Liu, S.; Zhou, L.; Duan, N.; Svyatkovskiy, A.; Fu, S.; et al. GraphCodeBERT: Pre-training Code Representations with Data Flow. *arXiv* **2021**, arXiv:2009.08366. [\[CrossRef\]](#)
14. Liu, Y.; Ott, M.; Goyal, N.; Du, J.; Joshi, M.; Chen, D.; Levy, O.; Lewis, M.; Zettlemoyer, L.; Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. *arXiv* **2019**, arXiv:1907.11692. [\[CrossRef\]](#)
15. Chen, M.Y.; Kiciman, E.; Fratkin, E.; Fox, A.; Brewer, E. Pinpoint: Problem determination in large, dynamic Internet services. In *Proceedings of the International Conference on Dependable Systems and Networks, Washington, DC, USA, 23–26 June 2002*; IEEE: Piscataway, NJ, USA, 2002; pp. 595–604. [\[CrossRef\]](#)
16. Dallmeier, V.; Lindig, C.; Zeller, A. Lightweight Defect Localization for Java. In *ECOOP 2005—Object-Oriented Programming*; Black, A.P., Ed.; Springer: Berlin/Heidelberg, Germany, 2005; pp. 528–550. [\[CrossRef\]](#)
17. Dallmeier, V.; Lindig, C.; Zeller, A. Lightweight bug localization with AMPLE. In *Proceedings of the Sixth International Symposium on Automated Analysis-Driven Debugging (AADEB'05), Monterey, CA, USA, 19–21 September 2005*; Association for Computing Machinery: New York, NY, USA, 2005; pp. 99–104. [\[CrossRef\]](#)
18. Weiser, M. Program Slicing. *IEEE Trans. Softw. Eng.* **1984**, SE-10, 352–357. [\[CrossRef\]](#)
19. Weiser, M.D. Program Slices: Formal, Psychological, and Practical Investigations of an Automatic Program Abstraction Method. Ph.D. Thesis, University of Michigan, Ann Arbor, MI, USA, 1979.
20. Agrawal, H.; Horgan, J.R. Dynamic program slicing. *ACM SIGPLAN Not.* **1990**, 25, 246–256. [\[CrossRef\]](#)
21. Agrawal, H.; Horgan, J.R.; London, S.; Wong, W.E. Fault localization using execution slices and dataflow tests. In *Proceedings of Sixth International Symposium on Software Reliability Engineering. ISSRE'95, Toulouse, France, 24–27 October 1995*; IEEE: Piscataway, NJ, USA, 1995; pp. 143–151. [\[CrossRef\]](#)
22. Binkley, D.; Gold, N.; Harman, M.; Islam, S.; Krinke, J.; Yoo, S. ORBS: Language-independent program slicing. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014), Hong Kong, China, 16–21 November 2014*; Association for Computing Machinery: New York, NY, USA, 2014; pp. 109–120. [\[CrossRef\]](#)
23. Papadakis, M.; Traon, Y.L. Metallaxis-FL: Mutation-based fault localization. *Softw. Test. Verif. Reliab.* **2015**, 25, 605–628. [\[CrossRef\]](#)
24. Moon, S.; Kim, Y.; Kim, M.; Yoo, S. Ask the Mutants: Mutating Faulty Programs for Fault Localization. In *Proceedings of the 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, OH, USA, 31 March–4 April 2014*; IEEE: Piscataway, NJ, USA, 2014; pp. 153–162. [\[CrossRef\]](#)
25. Wang, H.; Li, Z.; Liu, Y.; Chen, X.; Paul, D.; Cai, Y.; Fan, L. Can Higher-Order Mutants Improve the Performance of Mutation-Based Fault Localization? *IEEE Trans. Reliab.* **2022**, 71, 1157–1173. [\[CrossRef\]](#)
26. Watson, A.; Ufuktepe, E.; Palaniappan, K. Detecting Software Code Vulnerabilities Using 2D Convolutional Neural Networks with Program Slicing Feature Maps. In *Proceedings of the 2022 IEEE Applied Imagery Pattern Recognition Workshop (AIPR), Washington, DC, USA, 11–13 October 2022*; IEEE: Piscataway, NJ, USA, 2022; pp. 1–9. [\[CrossRef\]](#)
27. Ziems, N.; Wu, S. Security Vulnerability Detection Using Deep Learning Natural Language Processing. In *Proceedings of the IEEE INFOCOM 2021—IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), Vancouver, BC, Canada, 10–13 May 2021*; IEEE: Piscataway, NJ, USA, 2021; pp. 1–6. [\[CrossRef\]](#)
28. Meng, X.; Wang, X.; Zhang, H.; Sun, H.; Liu, X. Improving fault localization and program repair with deep semantic features and transferred knowledge. In *Proceedings of the 44th International Conference on Software Engineering (ICSE '22), Pittsburgh, PA, USA, 21–29 May 2022*; Association for Computing Machinery: New York, NY, USA, 2022; pp. 1169–1180. [\[CrossRef\]](#)
29. Li, X.; Geng, Q.; Huang, Q.; Li, X.; Tang, J.; Ye, Q. Spatial–Temporal Self-Compensating Graph Convolutional Network for Skeleton-Based Action Recognition Under Data Constraints. *IEEE Trans. Image Process.* **2026**, 35, 5818–5833. [\[CrossRef\]](#) [\[PubMed\]](#)
30. Wu, Y.; Li, Z.; Zhang, J.M.; Papadakis, M.; Harman, M.; Liu, Y. Large language models in fault localisation. *arXiv* **2023**. [\[CrossRef\]](#)
31. Li, J.; Tang, T.; Zhao, W.X.; Nie, J.-Y.; Wen, J.-R. Pre-Trained Language Models for Text Generation: A Survey. *ACM Comput. Surv.* **2024**, 56, 230. [\[CrossRef\]](#)
32. Wu, Y.; Li, Z.; Zhang, J.M.; Liu, Y. ConDefects: A New Dataset to Address the Data Leakage Concern for LLM-based Fault Localization and Program Repair. *arXiv* **2023**. [\[CrossRef\]](#)
33. Hosmer, D.W., Jr.; Lemeshow, S.; Sturdivant, R.X. Assessing the Fit of the Model. In *Applied Logistic Regression*; John Wiley & Sons: Hoboken, NJ, USA, 2013. [\[CrossRef\]](#)
34. Wilcoxon, F. Individual Comparisons by Ranking Methods. *Biom. Bull.* **1945**, 1, 80–83. [\[CrossRef\]](#)
35. Demšar, J. Statistical Comparisons of Classifiers over Multiple Data Sets. *J. Mach. Learn. Res.* **2006**, 7, 1–30.

36. Wu, Y.; Li, Z.; Zhang, J.M.; Liu, Y. ConDefects: A Complementary Dataset to Address the Data Leakage Concern for LLM-Based Fault Localization and Program Repair. In *FSE 2024: Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering*; Association for Computing Machinery: New York, NY, USA, 2024; pp. 642–646. [[CrossRef](#)]
37. Zhou, Y.; Liu, S.; Siow, J.; Du, X.; Liu, Y. Devign: Effective Vulnerability Identification by Learning Comprehensive Program Semantics via Graph Neural Networks. In *Advances in Neural Information Processing Systems*; Curran Associates, Inc.: Red Hook, NY, USA, 2019. Available online: https://proceedings.neurips.cc/paper_files/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html (accessed on 29 June 2025).

Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.