UDC 004.8+004.41
DOI: 10.15587/2706-5448.2024.310822

Illia Vokhranov,
Bogdan Bulakh

TRANSFORMER-BASED MODELS APPLICATION FOR BUG DETECTION IN SOURCE CODE

This paper explores the use of transformer-based models for bug detection in source code, aiming to better understand the capacity of these models to learn complex patterns and relationships within the code. Traditional static analysis tools are highly limited in their ability to detect semantic errors, resulting in numerous defects passing through to the code execution stage. This research represents a step towards enhancing static code analysis using neural networks.

The experiments were designed as binary classification tasks to detect buggy code snippets, each targeting a specific defect type such as NameError, TypeError, IndexError, AttributeError, ValueError, EOFError, SyntaxError, and ModuleNotFoundError. Utilizing the «RunBugRun» dataset, which relies on code execution results, the models – BERT, CodeBERT, GPT-2, and CodeT5 – were fine-tuned and compared under identical conditions and hyperparameters. Performance was evaluated using F1-Score, Precision, and Recall.

The results indicated that transformer-based models, especially CodeT5 and CodeBERT, were effective in identifying various defects, demonstrating their ability to learn complex code patterns. However, performance varied by defect type, with some defects like IndexError and TypeError being more challenging to detect. The outcomes underscore the importance of high-quality, diverse training data and highlight the potential of transformer-based models to achieve more accurate early defect detection.

Future research should further explore advanced transformer architectures for detecting complicated defects, and investigate the integration of additional contextual information to the detection process. This study highlights the potential of modern machine learning architectures to advance software engineering practices, leading to more efficient and reliable software development.

Keywords: *transformers, large language models, bug detection, defect detection, static code analysis, neural networks.*

Received date: 22.06.2024
Accepted date: 30.08.2024
Published date: 31.08.2024

How to cite
Vokhranov, I., Bulakh, B. (2024). Transformer-based models application for bug detection in source code. Technology Audit and Production Reserves, 5 (2 (79)), 00–00. https://doi.org/10.15587/2706-5448.2024.310822

1. Introduction

In the realm of commercial software development, the capability to deliver high-quality source code within shorter time frames is essential to meeting contemporary demands. Development assistance tools play a pivotal role in this context, particularly in the stage of static code analysis. Static code analysis, which does not require code execution, provides automatic analysis at the initial stages of code writing, thereby enhancing resource efficiency [1]. However, the effectiveness of this process depends on the quality of the alerts it generates, which developers must manage manually.

Traditional rule-based tools for static code analysis usually suffer from a high false-positive rate in alerts they raise [2]. Furthermore, these tools are limited in their ability to identify complex patterns within program code, resulting in numerous defects passing through to the code execution stage. To address these limitations, there is ongoing research in enhancing static code analysis techniques, encompassing various areas such as defect detection, defect correction, code review assistance, code smell detection, and code explanation and documentation.

The emergence of Large Language Models (LLMs) based on transformer architectures has brought significant advancements in various fields. LLMs represent a substantial breakthrough in artificial intelligence, marking a notable milestone in natural language processing tasks. Typically built on transformer architectures [3] and self-attention mechanisms [3, 4], these models are trained on extensive datasets comprising vast collections of texts from the internet, books, and code repositories. This has allowed transformer-based models to establish a robust foundation for their application in various software engineering tasks [5].

Research papers in the field of static code analysis substantially focus on enhancing tools to reduce false positive alerts. Another prominent area of interest is the development of code writing and code completion assistance tools. These research directions are undeniably crucial and essential for the software development industry. However, it is possible to notice a slight lack of focus in existing studies regarding the detection of more complex code issues, particularly those related to the runtime stage, during the static code analysis phase using machine learning techniques.

Bugs that manifest during the runtime stage are typically not associated with static analysis. These bugs often involve non-trivial patterns and relationships within the code that are not easily detectable by traditional static analysis methods. However, it is possible to suggest that modern machine learning architectures have the capacity to learn and recognize these complex patterns, thereby potentially identifying such sophisticated issues during the static analysis phase.

In light of this, let's aim to further explore the capabilities of modern transformer-based models in detecting bug-related patterns in code retrieved from code execution. Specifically, this paper evaluates the ability of these models to learn and identify various patterns of defects in real-world code snippets. By fine-tuning pre-trained transformer models and comparing their performance, let's seek to better understand their effectiveness in predicting errors that occur at the run-time stage. *The aim of this research* is to provide insights into the potential and limitations of transformer-based models in enhancing the accuracy of bug detection in software development processes.

2. Materials and Methods

2.1. Dataset selection. In training neural networks for source code processing tasks, data plays a critical role in determining the generalization ability, effectiveness, and performance of the resulting models [6]. In recent years, the demand for source code datasets has increased significantly. Today, there are various datasets collected using different methods and obtained from different sources. From the perspective of data availability, we have reached a point where the quantity of data is no longer a pressing issue. However, aside from quantity, the quality of data is equally important. Considerable effort is being made to obtain high-quality datasets for various software engineering tasks. Researchers and commercial companies employ diverse approaches to build and annotate source code datasets.

For instance, some datasets are based on manually crafted and annotated data, while others are annotated by scraping and categorizing information from sources such as Stack Overflow and GitHub. The latter ones are often enhanced by researchers who manually or automatically add specific labels, such as bug types. Industrial datasets may also include proprietary annotations derived from user behavior logs or business data, frequently requiring strict confidentiality agreements. Each dataset possesses distinct advantages and disadvantages that must be carefully considered when selecting data for a specific task [5].

As previously stated, our current research is focused on data labeled based on code executions. At the same time, let's aim to avoid synthetic data and focus on code from real-world projects. A dataset that meets these requirements is the *«RunBugRun» dataset*, created by Julian Aron Prenner and Romain Robbes in their work titled «RunBugRun – An Executable Dataset for Automated Program Repair» [7]. In their work, the authors highlight that incorporating real code executions in datasets bridges the gap between theoretical models and practical applications. This approach enables semantic-level evaluation, enhances fault localization, improves data quality and reliability, and provides much more realistic benchmarking [7].

It is important to mention that the appropriate use of synthetic data can be a powerful tool for model training and development. However, the availability of real and reliable data is essential for achieving high-quality validation in software engineering tasks.

2.2. Experiment's structure. Our experiments were designed as binary classification tasks to evaluate the performance of a few basic state-of-the-art models in detecting buggy code snippets. For each type of defect, we fine-tuned the selected models for 10 epochs under identical conditions. Each run was executed in the same environment, using the same hyperparameters, and on the same data specific to each defect type.

Fine-tuning of the models was performed using LoRA [8], a technique that accelerates the fine-tuning of large models while consuming less memory, thereby enhancing resource efficiency. The selected LoRA hyperparameters were: $rank = 16$, $alpha = 32$, and $dropout = 0.1$.

The experiments were conducted in the Kaggle environment, utilizing GPU P100 computing units.

Evaluation metrics were tracked for both training and test datasets to ensure a better understanding of the results. This was particularly important for experiments with relatively low validation data.

2.3. Models selection. To study the capabilities of transformer-based models for the given task, the *BERT*, *CodeBERT*, *GPT-2*, and *CodeT5* models were chosen as representatives of this architecture. These classic models were selected in their lighter versions to correspond with task complexity and data size.

For this study, two text-based models (BERT and GPT-2) and two models specifically trained on programming language data (CodeBERT and CodeT5) were selected.

The BERT and CodeBERT models represent a simpler architecture, consisting solely of the encoder component of the transformer architecture, whereas GPT-2 and CodeT5 feature more complex, generation-oriented encoder-decoder architectures. While the bug detection tasks do not necessarily require more complex generative architectures, understanding the potential of these models in bug detection remains of similar interest, since error detection is frequently accompanied by error correction.

2.4. Data preprocessing

2.4.1. Programming Language Focus. It was decided to conduct our experiments on Python code snippets. Python submissions are among the most prevalent in the dataset, alongside C++.

2.4.2. Exceptions filtering. For our experiments, it is necessary only records that represent bug fixes of a single defect in the code, marked as one of the Python Exceptions [9]. To retrieve this information, let's read the «exceptions» field in the execution results. Entries lacking specified exceptions or containing multiple exceptions are excluded.

2.4.3. Filtering by tokenized length. To maintain consistency across different models, let's exclude records with code snippets exceeding 512 input tokens, tokenized for each selected model (BERT, CodeBERT, GPT-2, CodeT5). This restriction allows to run experiments under the same conditions without having to implement a Sliding Window approach and ensures that none of our code snippets are truncated for input. The vast majority of code snippets are less in size, so we are not losing much data.

ISSN 2664-9969

2.4.4. Data labeling. Let's construct a balanced binary classification dataset with a 50/50 distribution of correct and buggy code snippets. This formation is performed as follows: each entry that has passed the selection criteria from previous steps is divided into two records. The initial buggy code fragment is marked to have defects with a label of 1, whereas the corrected code fragment is labeled as 0. Since the «RunBugRun» dataset contains only those submissions where the patched code was executed without errors, it is reasonably possible to assume that these code snippets are free of defects (at least in the known context). By providing such data, let's expect models to learn about the exact differences between code snippets that lead to particular defects.

2.4.5. Training samples selection. Certain types of defects are more prevalent in the dataset than others. To ensure that the fine-tuning process yields representative results, it is not possible to run our experiments for defects that are not sufficiently represented in the dataset. The starting point is with at least 200 buggy examples in the training data (which results in 400 records in the training set). Also, let's limit training data to 4000 records for each defect type. This threshold, determined through practical experiments, balances the need for sufficient data for fine-tuning and the execution time within our available resources.

2.5. Evaluation metrics. In source code bug detection tasks, selecting appropriate evaluation metrics is crucial. For example, the metric of Accuracy can be misleading in cases where defects are much rarer than correct code. A model that consistently predicts no defects may achieve high accuracy, yet fail to identify any actual defects.

One of the most convenient options for monitoring the performance of models in bug detection is to rely on the *Precision* and *Recall* metrics. While Precision indicates the model's ability to correctly identify true positives, Recall reflects the model's ability to capture the majority of existing defects. In addition to applying these metrics, let's employ the *F1-Score*. The F1-Score is the harmonic mean of Precision and Recall, providing a single, balanced measure that accounts for both false positives and false negatives. This makes it particularly suitable for defect detection scenarios.

3. Results and Discussions

3.1. Experiment 1: NameError detection. *NameError exception meaning:* The exception is raised when the identifier being accessed is not found in the local or global scope [9].

Code snippet example representing the NameError defect and its correction:

```
n = int(input())

res = []
for i in range(n):
  city, value = input().split()
  value = int(value)
  res.append([city, value, i+1])
res_s = sorted(res, key = lambda x: (x[0], -x[1]))
for i in range(n):
  print(ans[i][2])
  print(res_s[i][2])
```

Train data samples: 4000.
Test data samples: 1278.
Fine-tuning results are shown in Fig. 1–3.

The best epoch selection based on the models' performance is represented in Table 1.

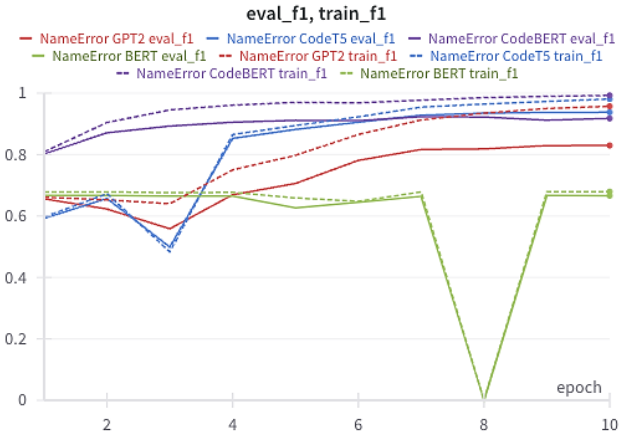

Fig. 1. F1-score fine-tuning results for NameError detection

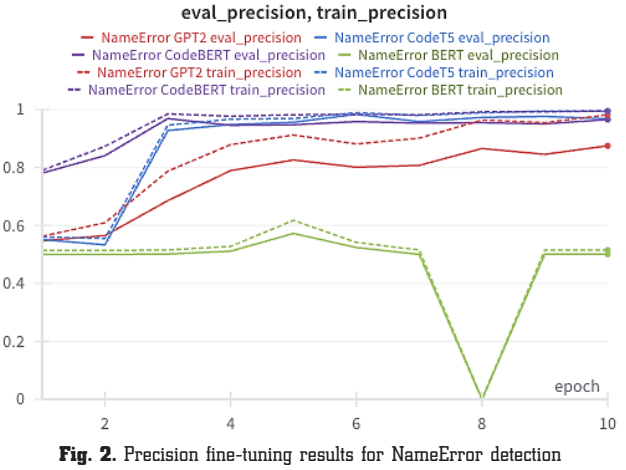

Fig. 2. Precision fine-tuning results for NameError detection

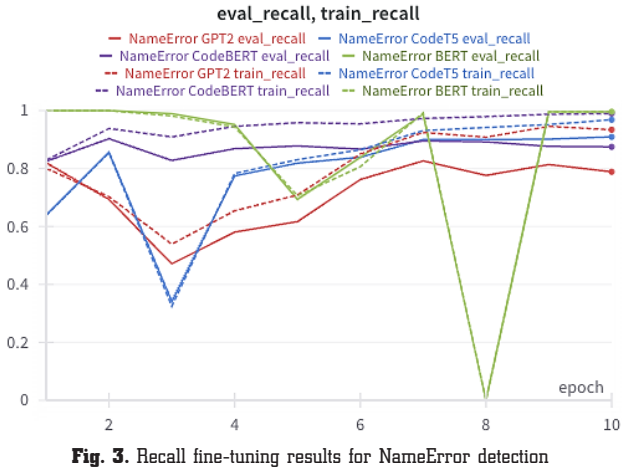

Fig. 3. Recall fine-tuning results for NameError detection

Table 1

Best epochs metric values for NameError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.94	0.98	0.97	0.99	0.89	0.98
CodeBERT	0.92	0.99	0.95	0.99	0.91	0.97
GPT2	0.83	0.96	0.88	0.98	0.79	0.93
BERT	0.67	0.68	0.50	0.52	1.00	1.00

Discussion: Upon detecting a NameError, only the BERT model failed to produce acceptable results. Its Precision score did not rise significantly above 0.5, indicating the model's inability to learn the error patterns. In contrast, the CodeT5 and CodeBERT models demonstrated excellent performance.

Code defects that lead to a NameError exception can often be successfully identified by classical static analysis tools. Therefore, the strong results observed in this experiment are, to some extent, expected.

3.2. Experiment 2: TypeError detection. *TypeError exception meaning:* The exception is raised when an operation or function is applied to an incorrect/unsupported object type [9].

Code snippet example representing the TypeError defect and its correction:

```
list = map(int, input().split())
list = list(map(int, input().split()))
S = list[0]
W = list[1]
if W >= S: print("unsafe")
else: print("safe")
```

Train data samples: 4000.

Test data samples: 1134.

Fine-tuning results are shown in Fig. 4–6.

The best epoch selection based on the models' performance is represented in Table 2.

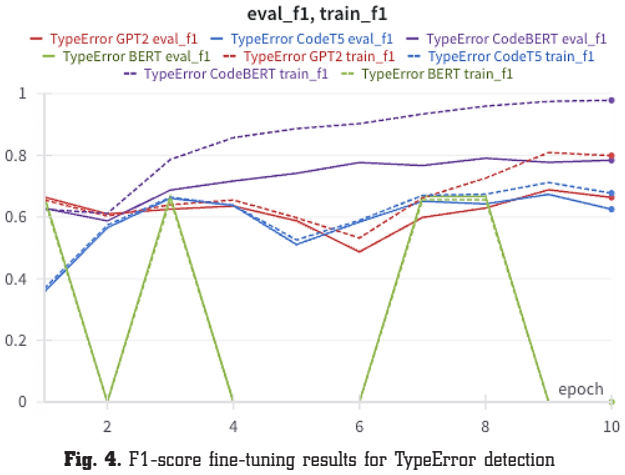

Fig. 4. F1-score fine-tuning results for TypeError detection

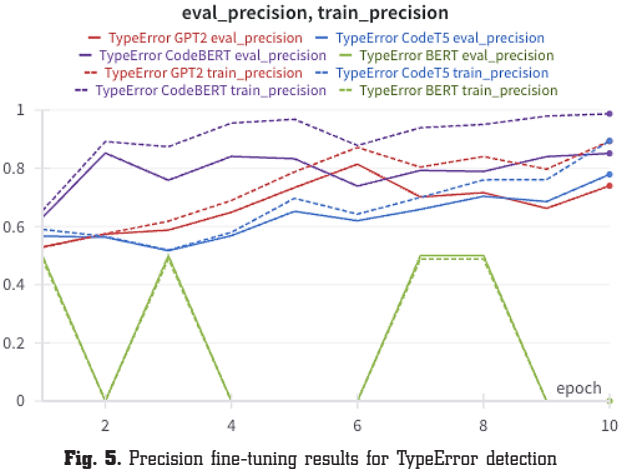

Fig. 5. Precision fine-tuning results for TypeError detection

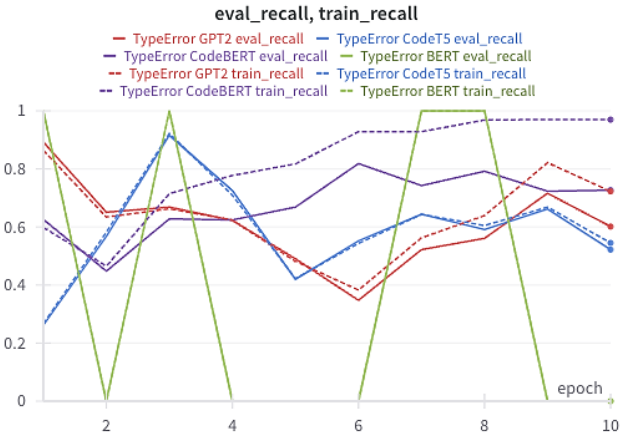

Fig. 6. Recall fine-tuning results for TypeError detection

Table 2

Best epochs metric values for TypeError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.67	0.71	0.69	0.76	0.66	0.67
CodeBERT	0.79	0.96	0.79	0.95	0.79	0.97
GPT2	0.69	0.81	0.66	0.80	0.72	0.82
BERT	0.67	0.66	0.50	0.49	1.00	1.00

Discussion: The results indicated that detecting TypeError was somewhat challenging. The models did not perform exceptionally well when evaluated on the test set; however, they showed some ability to identify patterns of such defects, as evidenced by their reasonable performance on the training set. This suggests that under better conditions, (e. g., slightly different data in the dataset) the models might have achieved better results. Nevertheless, this type of error is undoubtedly difficult to detect during simple code analysis due to its complex dependencies within the code. To reliably detect such errors in various conditions and contexts, models need to be trained on a very large and diverse set of code snippets.

3.3. Experiment 3: IndexError detection. *IndexError exception meaning:* The exception is raised when attempting to access an element in a list, tuple, or any other sequence with an index that is out of the range of available indices [9].

Code snippet example representing the IndexError defect and its correction:

```
n = int(input())
s = input()
a = [str(c) for c in s]
c = 0

for i in range(n):
for i in range(n-2):
  if a[i] == 'A' and a[i+1] == 'B' and a[i+2] == 'C':
    c +=1
  else:
    continue
print(c)
```

Train data samples: 4000.

Test data samples: 588.

ISSN 2664-9969

Fine-tuning results are shown in Fig. 7–9.

The best epoch selection based on the models' performance is represented in Table 3.

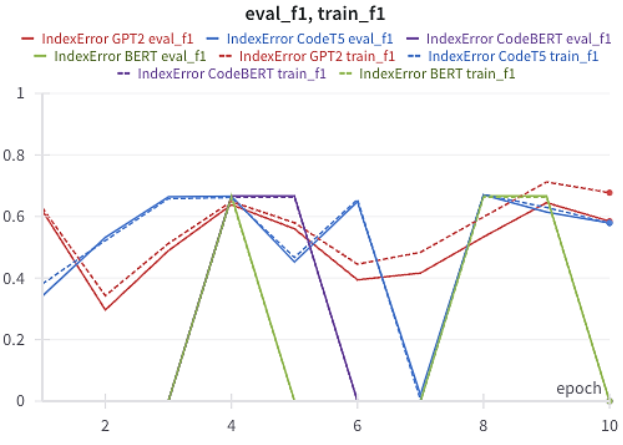

Fig. 7. F1-score fine-tuning results for IndexError detection

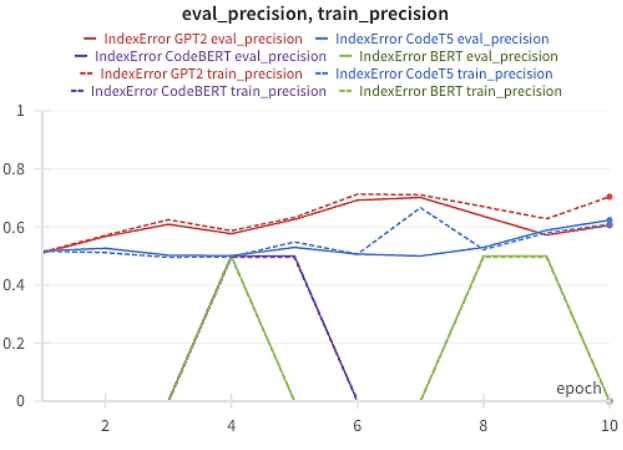

Fig. 8. Precision fine-tuning results for IndexError detection

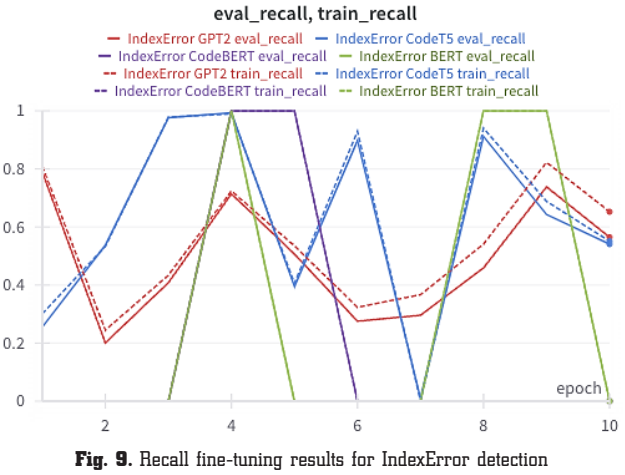

Fig. 9. Recall fine-tuning results for IndexError detection

Table 3

Best epochs metric values for IndexError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.58	0.58	0.62	0.61	0.55	0.54
CodeBERT	0.66	0.66	0.5	0.5	1	1
GPT2	0.64	0.71	0.57	0.63	0.74	0.82
BERT	0.66	0.66	0.5	0.5	1	1

Discussion: Detecting IndexError proved to be a particularly challenging task. For this defect, the difference between buggy and correct code snippets often comes down to small details like «+1» and the results of arithmetic calculations, making such defects highly challenging to identify. The CodeT5 and GPT models performed slightly better than random guessing. Given this and the nature of these defects, it is possible to infer that models with greater capacity and potentially newer architectures are needed to capture such complex dependencies in the code.

3.4. Experiment 4: AttributeError detection. *AttributeError exception meaning:* The exception is raised when an attribute reference or assignment fails [9].

Code snippet example representing the AttributeError defect and its correction:

```
a, b = input().strip().Split()
a, b = input().strip().split()

print(int(a)/int(b))
```

Train data samples: 3114.

Test data samples: 214.

Fine-tuning results are shown in Fig. 10–12.

The best epoch selection based on the models' performance is represented in Table 4.

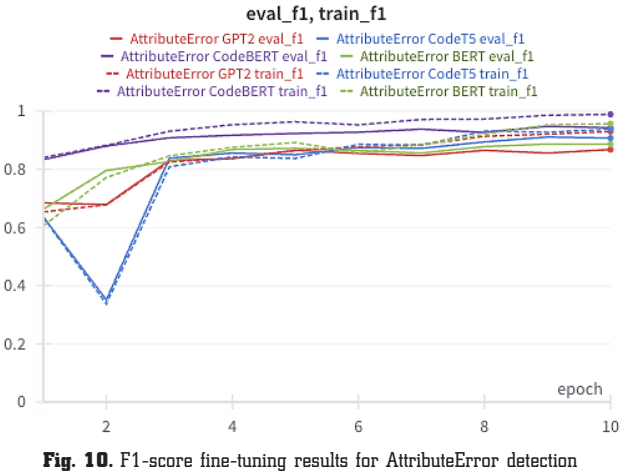

Fig. 10. F1-score fine-tuning results for AttributeError detection

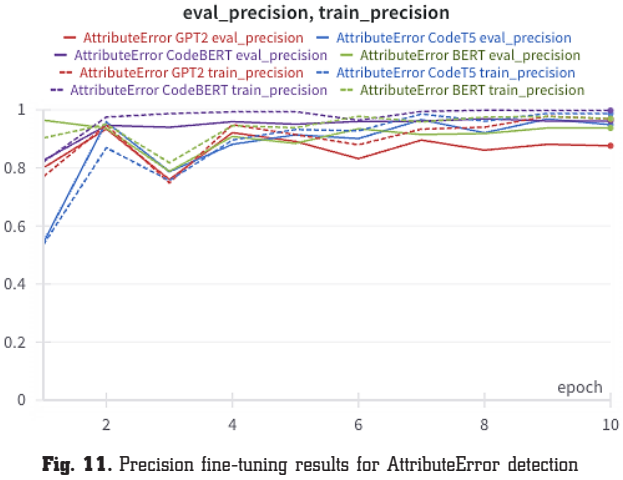

Fig. 11. Precision fine-tuning results for AttributeError detection

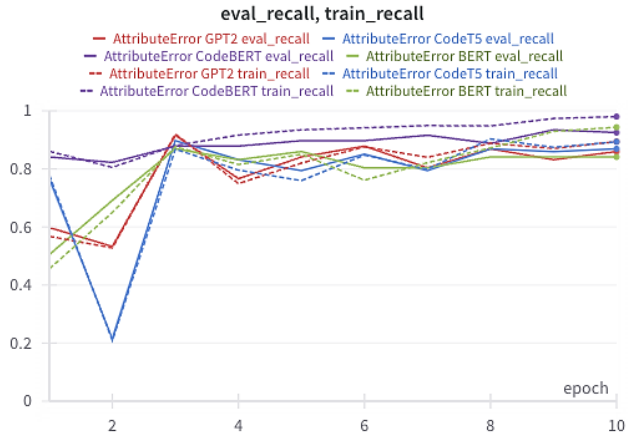

Fig. 12. Recall fine-tuning results for AttributeError detection

Table 4

Best epochs metric values for AttributeError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.91	0.93	0.97	0.99	0.86	0.87
CodeBERT	0.95	0.99	0.96	1	0.93	0.97
GPT2	0.87	0.93	0.88	0.97	0.86	0.9
BERT	0.89	0.96	0.94	0.97	0.84	0.94

Discussion: In AttributeError detection, all models demonstrated excellent results. This suggests that this type of defect requires only a sufficient amount of suitable training data to enable models to learn which attributes and references are correct for specific modules.

It is important to understand that such straightforward detection is feasible only with widely used modules and libraries, as the models need to «learn how to use them correctly». But at the same time, this definitely can be enhanced with additional context for models (like lists of available attributes for particular modules or other essential information retrieved with AST parsing).

3.5. Experiment 5: ValueError detection. *ValueError exception meaning:* The exception is raised when an operation or function receives an argument that has an inappropriate/invalid value, and the IndexError was not raised [9].

Code snippet example representing the ValueError defect and its correction:

```
a, b = input()
a, b = input().split()
num_a = int(a)
num_b = int(b)
dif = num_a - num_b*2

if dif >= 0:
  print(dif)
else:
  print(0)
```

Train data samples: 2913.

Test data samples: 210.

Fine-tuning results are shown in Fig. 13–15.

The best epoch selection based on the models' performance is represented in Table 5.

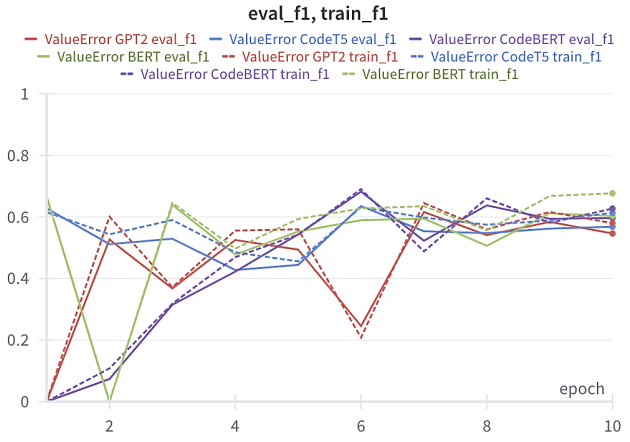

Fig. 13. F1-score fine-tuning results for ValueError detection

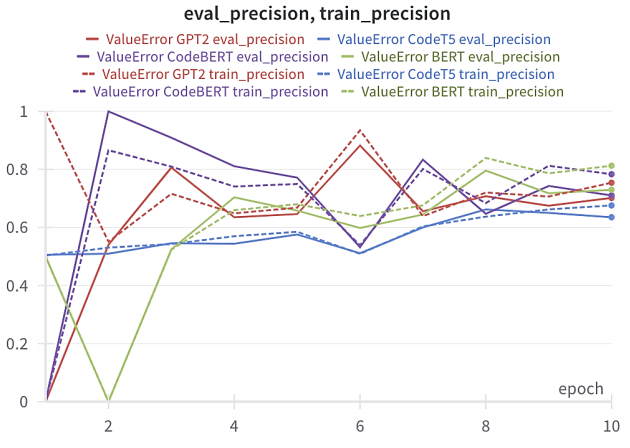

Fig. 14. Precision fine-tuning results for ValueError detection

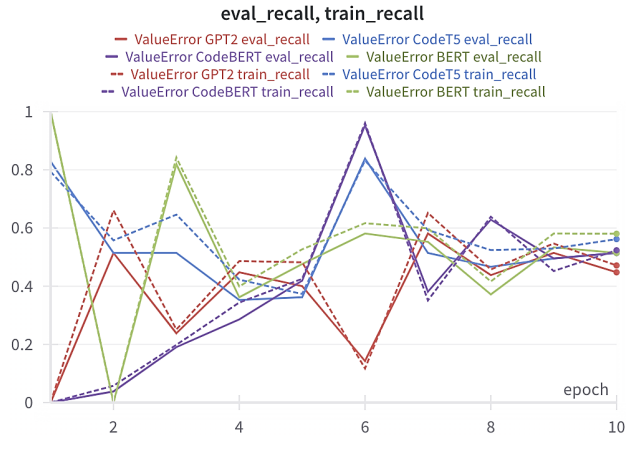

Fig. 15. Recall fine-tuning results for ValueError detection

Table 5

Best epochs metric values for ValueError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.57	0.61	0.64	0.68	0.51	0.56
CodeBERT	0.6	0.63	0.71	0.7	0.51	0.52
GPT2	0.58	0.62	0.68	0.7	0.51	0.55
BERT	0.61	0.67	0.72	0.79	0.53	0.58

Discussion: Detecting the ValueError exception is quite challenging, as it requires both an understanding of the argument's value and knowledge about the functions and

ISSN 2664-9969

operations to which it is passed. Surprisingly, the models demonstrated some ability to handle this detection. In this case, it's primarily attributed to the data consistency, which enabled the models to effectively learn the context of provided code snippets. For reliable detection of such defects in practice, a large amount of diverse data is needed, as well as additional context about the functions and operations being used.

3.6. Experiment 6: EOFError detection. *EOFError exception meaning:* The exception is raised when the input() function hits an end-of-file condition (EOF) without reading any data [9].

Code snippet example representing the EOFError defect and its correction:

```
while 1:
  x = input()
  if x[0] == "0":
  if x[0] == 0:
    break
  print (sum(int(i)for i in x))
```

Train data samples: 1014.
Test data samples: 76.
Fine-tuning results are shown in Fig. 16–18.
The best epoch selection based on the models' performance is represented in Table 6.

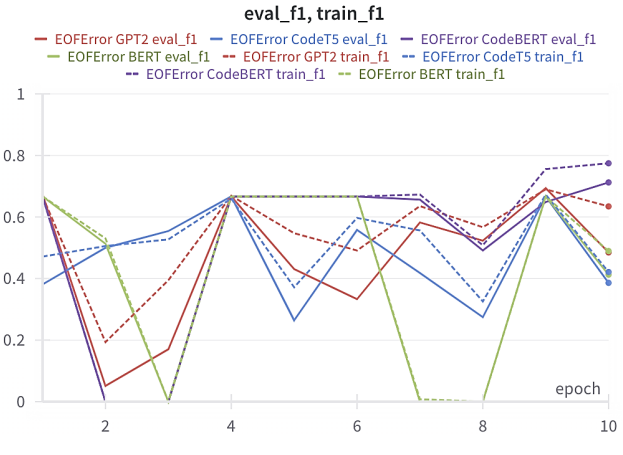

Fig. 16. F1-score fine-tuning results for EOFError detection

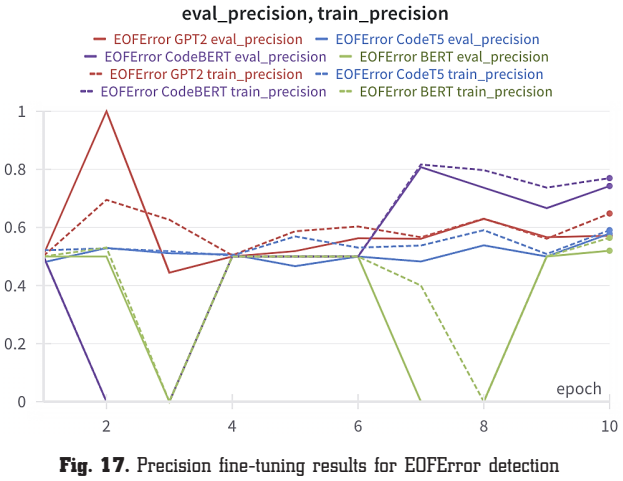

Fig. 17. Precision fine-tuning results for EOFError detection

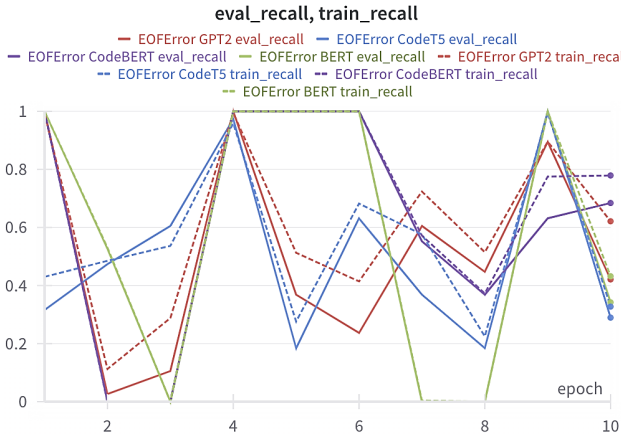

Fig. 18. Recall fine-tuning results for EOFError detection

Table 6

Best epochs metric values for EOFError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.39	0.42	0.58	0.6	0.29	0.33
CodeBERT	0.71	0.77	0.74	0.77	0.68	0.78
GPT2	0.69	0.69	0.56	0.57	0.89	0.9
BERT	0.41	0.49	0.52	0.56	0.34	0.43

Discussion: The EOFError exception is highly related to the data passed into the input() function by users, making direct detection of the issue itself unavailable. However, similar to many other tasks in static code analysis, it is possible to identify the presence or absence of certain structures for handling input data in the code. In our experiment, the CodeBERT model demonstrated a slightly better performance in detecting such dependencies. However, it is possible to believe that addressing such defects requires more focused data that can highlight specific patterns in source code. Approaches utilizing synthetic training data may be particularly useful in such cases.

3.7. Experiment 7: SyntaxError detection. *SyntaxError exception meaning:* The exception is raised when the interpreter encounters a syntax error in the code. This may also occur when certain identifiers are used in unexpected contexts, such as calling 'return' outside a function or 'break' outside a loop.

Code snippet example representing the SyntaxError defect and its correction:

```
n = int(input())
x = list(map(int, input().split()))
ans = x[0]
if 0 in x:
  ans = 0
  break

else:
  for i in range(1,n):
    ans = ans * x[i]
    if ans > 10 ** 18:
      ans = -1
      break
print(ans)
```

Train data samples: 588.
Test data samples: 42.
Fine-tuning results are shown in Fig. 19–21.

The best epoch selection based on the models' performance is represented in Table 7.

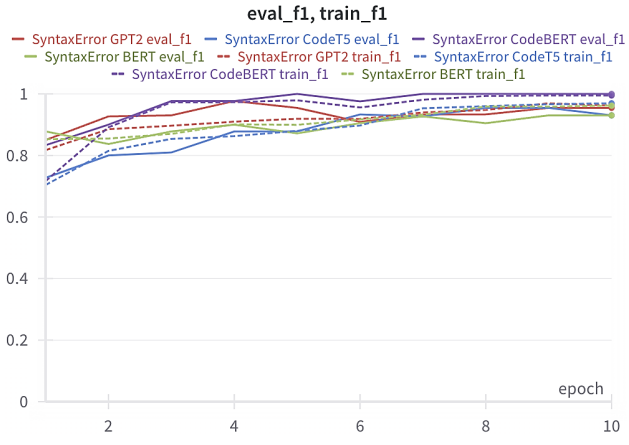

Fig. 19. F1-score fine-tuning results for SyntaxError detection

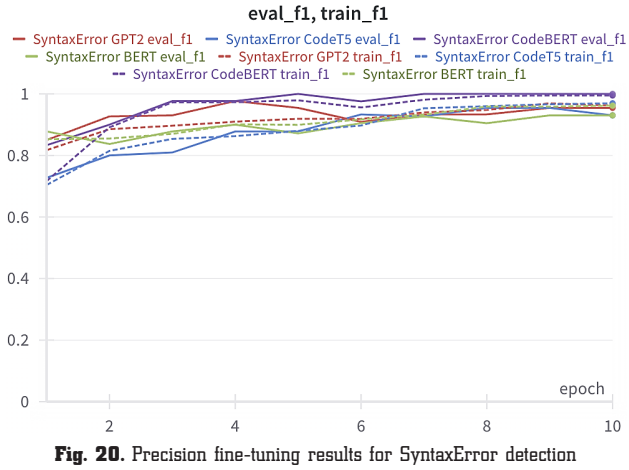

Fig. 20. Precision fine-tuning results for SyntaxError detection

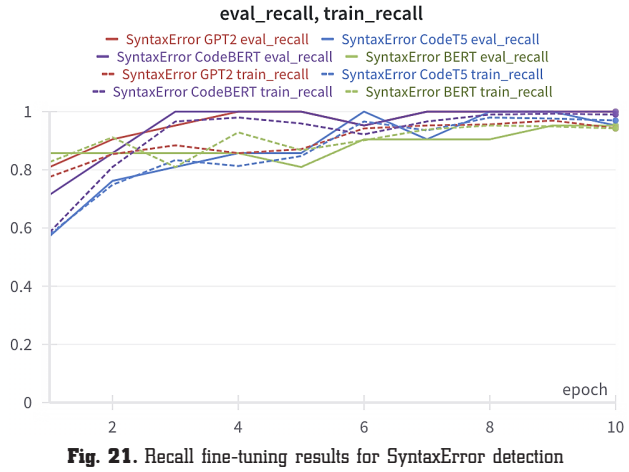

Fig. 21. Recall fine-tuning results for SyntaxError detection

Table 7

Best epochs metric values for SyntaxError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	0.95	0.97	0.91	0.96	0.98	1
CodeBERT	1	1	1	1	1	1
GPT2	0.95	0.97	0.91	0.97	1	0.97
BERT	0.93	0.96	0.9	0.97	0.95	0.95

Discussion: Syntax errors are a primary focus in static code analysis. Most of these errors are effectively detected by traditional static analysis tools. However, identifying more complex dependencies requires more sophisticated rules. As a result, there is no single advanced analyzer capable of detecting all possible syntax errors.

In our experiment, all models demonstrated strong performance despite the limited amount of training data. Given the small size of the test set for this type of error, it is important to also consider the evaluation of the training set while analyzing results.

3.8. Experiment 8: ModuleNotFoundError detection. *ModuleNotFoundError exception meaning:* The exception is a subclass of ImportError and is raised by import when a module cannot be located [9].

Code snippet example representing the ModuleNotFoundError defect and its correction:

```
from functols import reduce
from functools import reduce
from math import gcd
N,X = map(int, input().split())
x =[abs(X-int(i)) for i in input().split()]
print(reduce(gcd,x))
```

Train data samples: 489.
Test data samples: 22.
Fine-tuning results are shown in Fig. 22–24.

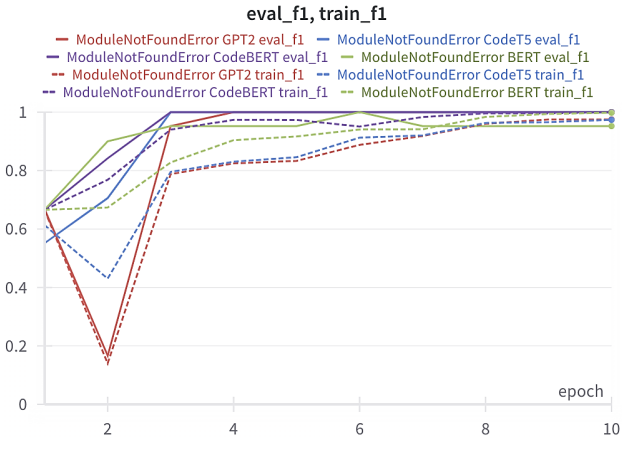

Fig. 22. F1-score fine-tuning results for ModuleNotFoundError detection

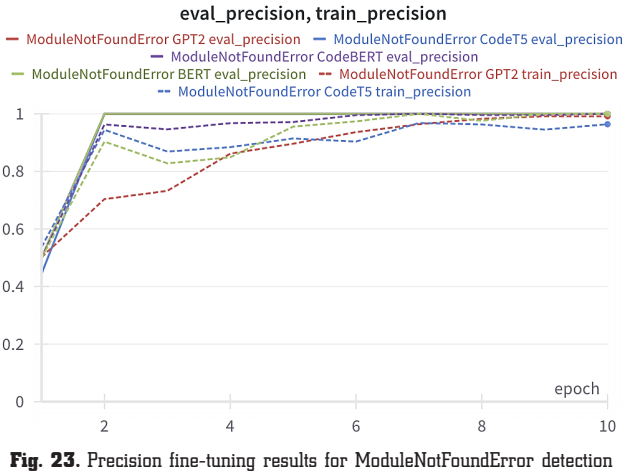

Fig. 23. Precision fine-tuning results for ModuleNotFoundError detection

ISSN 2664-9969

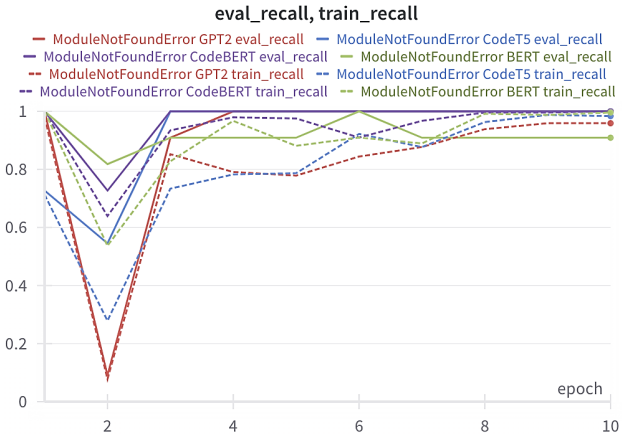

Fig. 24. Recall fine-tuning results for ModuleNotFoundError detection

The best epoch selection based on the models' performance is represented in Table 8.

Table 8

Best epochs metric values for ModuleNotFoundError detection

Model	F1		Precision		Recall	
	Eval	Train	Eval	Train	Eval	Train
CodeT5	1	0.97	1	0.96	1	0.98
CodeBERT	1	1	1	1	1	1
GPT2	1	0.98	1	0.99	1	0.96
BERT	0.95	1	1	1	0.9	0.99

ModuleNotFoundError detection has shown good results with all models. Despite the relatively small amount of data, the models were able to identify patterns of the existing errors. This type of defect, similar to some of the previous ones, also requires knowledge about the possible modules. Therefore, to detect such defects in practice, it is desirable to have additional input information. For example, such information could be obtained from package managers like «pip», or similar sources.

This study confirms that transformer-based models, with their ability to capture and understand intricate code patterns, offer promising solutions for static code analysis and bug detection. Nevertheless, achieving reliable results in practical applications necessitates not only high-quality and diverse datasets but also careful consideration of the types of defects targeted for detection. Collectively, these factors underscore the importance of data quality and diversity in enhancing the effectiveness of software engineering tasks.

This research represents a preliminary step towards developing a method to enhance static code analysis using neural networks in the future. Our future work will include exploring advanced transformer architectures for detecting defects that rely on complicated dependencies, as well as investigating the integration of additional contextual information for improved code understanding.

4. Conclusions

The experiments conducted on detecting various types of defects using transformer-based models demonstrate their robust capabilities in learning complex patterns within source code. Achieving effective detection results depends on two key factors:

1) performing detection on defect types that have a genuine reflection in code dependencies (even if they are complex);

2) having a sufficient quantity of representative data in the training set.

Furthermore, while it may seem evident, we want to highlight once again the paramount importance of data quality. Working with source code demands high precision, and insufficient quality of training data can significantly impair model performance. Simultaneously, if we aim to get closer to real-world tasks, diversity in training examples is critical to ensuring good model generalization in practice. Collectively, these factors make software engineering tasks highly sensible to data quality.

In the experiments, models like CodeT5 and CodeBERT generally outperformed others. However, complex defects like IndexError and TypeError presented significant challenges, suggesting the need for more sophisticated models or additional contextual information for reliable detection.

Conflict of interest

The authors declare that they have no conflict of interest in relation to this study, including financial, personal, authorship, or any other, that could affect the study and its results presented in this article.

Financing

The study was conducted without financial support.

Data availability

The paper has no associated data.

Use of artificial intelligence

The authors confirm that they did not use artificial intelligence technologies when creating this work.

References

1. Tassey, G. (2002). *The Economic Impacts of Inadequate Infrastructure for Software Testing (NIST Planning Report 02-3)*. RTI International. National Institute of Standards and Technology. Available at: https://www.nist.gov/system/files/documents/director/planning/report02-3.pdf Last accessed: 22.07.2024.
2. Nachtigall, M., Schlichtig, M., Bodden, E. (2022). A large-scale study of usability criteria addressed by static analysis tools. *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. https://doi.org/10.1145/3533767.3534374
3. Vaswani, A., Shazeer, N. M., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N. et al. (2017). Attention is All you Need. *Neural Information Processing Systems*. https://doi.org/10.48550/arXiv.1706.03762
4. Bahdanau, D., Cho, K., Bengio, Y. (2014). *Neural Machine Translation by Jointly Learning to Align and Translate*. CoRR, abs/1409.0473. https://doi.org/10.48550/arXiv.1409.0473
5. Hou, X., Zhao, Y., Liu, Y., Yang, Z., Wang, K., Li, L. et al. (2023). *Large Language Models for Software Engineering: A Systematic Literature Review*. ArXiv, abs/2308.10620. https://doi.org/10.48550/arXiv.2308.10620
6. Sun, Z., Li, L., Liu, Y., Du, X. (2022). On the Importance of Building High-quality Training Datasets for Neural Code Search. *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, 1609–1620. https://doi.org/10.1145/3510003.3510160

7. Prenner, J. A., Robbes, R. (2023). *RunBugRun – An Executable Dataset for Automated Program Repair*. ArXiv, abs/2304.01102. https://doi.org/10.48550/arXiv.2304.01102
8. Hu, J. E., Shen, Y., Wallis, P., Allen-Zhu, Z., Li, Y., Wang, S., Chen, W. (2021). *LoRA: Low-Rank Adaptation of Large Language Models*. ArXiv, abs/2106.09685. https://doi.org/10.48550/arXiv.2106.09685
9. *Built-in exceptions*. *Python Documentation*. Python Software Foundation. Available at: https://docs.python.org/3/library/exceptions.html Last accessed: 22.07.2024
10. Marjanov, T., Pashchenko, I., Massacci, F. (2022). Machine Learning for Source Code Vulnerability Detection: What Works and What Isn't There Yet. *IEEE Security & Privacy, 20 (5)*, 60–76. https://doi.org/10.1109/msec.2022.3176058
11. Fang, C., Miao, N., Srivastav, S., Liu, J., Zhang, R., Fang, R. et al. (2023). *Large Language Models for Code Analysis: Do LLMs Really Do Their Job?* ArXiv, abs/2310.12357. https://doi.org/10.48550/arXiv.2310.12357
12. Xiao, Y., Zuo, X., Xue, L., Wang, K., Dong, J. S., Beschastnikh, I. (2023). *Empirical Study on Transformer-based Techniques for Software Engineering*. ArXiv, abs/2310.00399. https://doi.org/10.48550/arXiv.2310.00399

✉***Illia Vokhranov,*** *PhD Student, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, e-mail: vokhranov@gmail.com, ORCID: https://orcid.org/0009-0000-1702-0460*

Bogdan Bulakh, *PhD, Associate Professor, Department of System Design, National Technical University of Ukraine «Igor Sikorsky Kyiv Polytechnic Institute», Kyiv, Ukraine, ORCID: https://orcid.org/0000-0001-5880-6101*

✉*Corresponding author*