Source Code Analysis: Exploring Solutions for Software Quality

1st Sakshi Kusmude
Department of Information Technology
Vishwakarma Institute of Information Technology, Pune
sakshi.22210253@viit.ac.in

2nd Utkarsha Baraskar
Department of Information Technology
Vishwakarma Institute of Information Technology, Pune
utkarsha.22211455@viit.ac.in

3rd Ratnmala N. Bhimanpallewar
Department of Information Technology
Vishwakarma Institute of Information Technology, Pune
ratnmala.bhimanpallewar@viit.ac.in

4th Purvesh Bhole
Department of Information Technology
Vishwakarma Institute of Information Technology, Pune
purvesh.22211425@viit.ac.in

5th Shravani Padwal
Department of Information Technology
Vishwakarma Institute of Information Technology, Pune
shravani.22211302@viit.ac.in

Abstract

Although there is a wide range of software applications, it is imperative to implement the correct quality to enhance the efficiency, security, and stability of the applications. Problems need to be found in source code and the code analysis becomes primary instruments for this, in the hands of the developers. It focuses on two main techniques: White box testing, which involves thoroughly inspecting the internal structure and logic of the code without actually running it, ensuring everything works as intended; and black box testing, which analyzes the code during execution to check how it performs in real-time. The paper also looks at the use of auto tools and machine learning techniques in enhancing code analysis. These solutions assist the developers to turn their created software into more stable, secure, and performing applications. Furthermore, the paper focuses on such aspects as code reviews and training of developers to increase their awareness of what practices should be followed. Thus, through static code analysis, the level of code security, as well as the overall maintainability and functionality of software, may be boosted. Thus, making software more reliable and satisfying the requirements of the existing technologies, based on this approach.

Keywords : Software Quality, Machine Learning, Developer Training, Code Functionality

Introduction

In the modern world where software is being developed constantly, there is increasing importance of ensuring that the quality of the software is as high as possible and in order to do that, source code analysis has become crucial. Code analysis involves a range of methods that is used to effectively locate problems, weaknesses or potential problems in code so as to make the code more reliable secure and easy to evolve. Since software systems have become more intricate, it is very hard to conduct manual scrutiny of codes; thus, the creation of automated tools that make use of static and dynamic analyses. They include a new ProgQuery tool for Java that is a graph-based technique that improves the speed and scalability of source code analysis to 245 times compared with other methods. This points out the fact that it is possible to analyze a large scale complex software systems much more effectively and efficiently.

Another important contribution proposed in the paper is called FineCodeAnalyzer intended for the method-level bug localization and since the cognitive cost is in the focus of the improvement, it helps developers to gain more efficiency in order to make proper changes. It was demonstrated that a manual bug localization using this tool is faster and more accurate in detecting features in the source code than traditional methods [1]. In addition, studies that perform a comparison of the above-mentioned analysis tools such as SonarQube, Coverity as well as CodeSonar have shown that each tool has been found to have certain merits on their own. For example, SonarQube is best suited for detecting faults in code sourced from different languages while Coverity is best suited for detecting security issues making them invaluable in improving on the quality of software in various ways [2].

However, it is for this reason that such problems can still be observed in the contemporary architectures, for instance, in the distributed systems, information restatement or a possible inconsistency of security policies. Scholars have observed that there is a need to come up with better solutions as the complexity that arises from working with distributed systems may prove hard to manage [3]. Also, most of the open-source static code analysis tools are Semgrep and Klocwork, and they are also good for defining vulnerabilities and they are multi-language, cross-platform, and this may force the developer to choose the right tool for the given project according to the needs [4]. This paper aims at presenting the existing tools and methods regarding source code analysis as well as the current concerns and future prospects towards the enhancement of the quality of the developed software.

Literature Survey

As it can be seen, the process of using various approaches and methods of code reviewing as a process of analysis of a definite source code has significantly enhanced the quality and effectiveness of the programs. Looking from the viewpoint of computer science, one would like large scale applications to have efficiency and expressiveness; these are exactly the characteristics that ProgQuery, a graph-based system written into Java, has the ability to do well in, the possibly large scale of the applications I was referring to. This has been backed by the versatility of this tool in handling huge codes as well as scaling and the reduce in the time required for analyze codes. In the same way, FineCodeAnalyzer integrates high accuracy of bug localization on the method level that minimizes the load on the developers and accelerates the identification process.

The comparative analysis of tools, used for the source code analysis, states about the essence of such tools as SonarQube, Coverity, CodeSonar and underlines the peculiarities of their activities. Both SonarQube and Coverity are equally good tools in their own right; while SonarQube has the ability to find flaws in multiple languages, Coverity is particularly helpful in finding security flaws. They have their specific uses, and this is why the developers must choose the appropriate one depending on the project they are working on. However, as mentioned before, the analysis of distributed systems still remains a problem and a number of issues such as information restatement and security inconsistencies need more profound solutions.

The analysis also reveals that static open source tools are also useful in the identification of security weaknesses across different programming languages. Specific tools such as Semgrep do stand out in JavaScript and use of SonarQube can be more general with an organized method of defective code identification. However, these tools also show that there is continuous improvement is still required in dealing with scalability, precision, and adaptability issues CACM in distributed architecture and projects.

Methodology

The process of making our machine learning model that would classify code snippets into certain categories was well structured. In this section, a detailed methodological approach – data preprocessing, model architecture, training and validation – is described.

Data Collection and Preprocessing

The initial part of the task included the generation of a labelled dataset which includes code snippets and that data was collected and placed into a CSV file format. Every snippet was labelled and put into one of the categories that were used to build the model on.
Tokenization was applied on the textual data to which it was turned so that it can be manipulated by the model. Each code snippet was essentially converted into a sequence of integers, with each number representing a specific token in the given programming language: it could be a word or a character. Mathematically, if S means the available of code snippets and T is the tokenizer, each of the code snippets $s_i \in S$ is converted into an integer sequence

$$T(s_i)=[t_1,t_2,\dots,tn] \text{ , where } t_i \text{ are token indices}$$

Due to the possibility of sequences in the input data having different lengths the used sequences were padded. All sequences were aligned to 500 tokens, these exercises were to make sure that the input dimensions in the neural network are consistent. This was important because unlike traditional Machine learning algorithms, neural networks require the inputs to be of the same size.

Model Architecture

The architecture of the model consisted of three key layers: the embedding layer that is followed by LSTM layer and finally the last step in dense layer with softmax activation at the end. The embedding layer converted tokens into meaningful vectors, the LSTM captured the sequence patterns, and the dense layer made final predictions. Together, these layers worked in harmony to analyze the code snippets and make accurate classification decisions.

The embedding layer was the initial layer of the model and it required transforming the tokenized integer sequences to dense vectors. Let the input sequence be $z = [z_1 , z_2, \dots, z_n]$, where z_i is token index. The first encoder layer called the embedding layer maps each token z_i to a corresponding vector $v_i \in R^d$, where d is the embedding dimension, which in this work was set to 128. These vector representations that the layer learns during training are capable of capturing semantic relationship of the tokens present in it.

Then, we added an LSTM layer to extract sequential information of tokens in the code snippets. There is also the LSTMs which are efficient in handling sequence data since it has a

memory which holds information from previous tokens. The LSTM computes for the hidden state n_t and cell state
m_t at each time step t, that has been entered as an input, the previous state n_{t-1} and previous cell state m_{t-1}.

$$p_t = \sigma(W_p \cdot [n_{t-1}, z_t] + b_p)$$

$$q_t = \sigma(W_q \cdot [n_{t-1}, z_t] + b_q)$$

$$r_t = \sigma(Wr \cdot [n_{t-1}, z_t] + b_r)$$

$$m_t = q_t * m_{t-1} + q_t * \tanh(W_m \cdot [n_{t-1}, z_t] + b_m)$$

$$n_t = r_t * \tanh(m_t)$$

In the above equations, p_t, q_t, r_t these are the forget, input, and output gates manage the flow of information in the model. In this context, σ (sigma), which is sigmoid activation function applied element-wise and tanh denotes the element-wise hyperbolic tangent function, each playing a key role in processing and transforming the data. These long-term dependencies that LSTM can capture make it a perfect candidate for processing code snippets, where understanding the context over multiple tokens becomes essential.

Lastly, the last layer of the model which is the output layer was of dense type with softmax activation function. The activation of softmax makes the LSTM output be a probability distribution over a certain set of predetermined classes. Its mathematical definition is as follows:

softmax() =

where is the output for class i, while the denominator sums up to the exponential of all class outputs. The result is a probability distribution, that is where the class with the highest probability is chosen as the model's prediction is made.

Model Training

For optimization the model was trained using the Adam optimizer, one of the preferred optimization algorithms because it adapts the learning rate for every parameter based on their pace. This helps Adam combine the benefits of both momentum and RMSprop, allowing faster convergence and better performance.

Categorical cross entropy is used as the loss function for multi class classification since it is most common for this type of problem and is expressed as:

Loss =

where is the predicted probability for class i. Training was done on 32 for total of 10 epochs, including 20% of the data to monitor overfitting.

Evaluation and Visualization

After the training was over, the model was tested on full data and we got a test accuracy of 53. 9%. These scores represent accuracy of the model in identifying the code snippets to the right categories.

To have more insight about the performance of this model, we equally illustrated the training and the validation accuracy as well as the loss after each epoch of 10. This showed that the accuracy was increasing on training as the epochs passed by and suggested that the model is learning. However, the validation accuracy showed signs of convergence, which suggests that perhaps the model needs a little bit more tuning to increase the generalization of data of unseen datasets. The loss curves shown below described the model's progression and validation while achieving higher validation accuracy, which was somewhat tricky.

Results

The model which was trained over 10 epochs had the following results:

Training Performance: The training process indeed showed a fair improvement, starting from 46.68% in the first epoch to a gradual increase of 57.98% by the tenth epoch.

The loss during training decreased from 0.6957 in the first epoch to 0.6746 by the final epoch, showing that at least the model was learning and minimizing errors as time progressed.

However, the accuracy fluctuated a bit in the middle epochs, which may indicate that this model had some difficulties in smoothly enhancing its performance during this time.

Validation Performance: The validation accuracy started at 52.50%, and interestingly hit its peak at 53.00% in the third epoch, but eventually fell to 44.50% at the last epoch.

By contrast, the validation loss was on an upward trend from 0.6918 in the first epoch to 0.7059 by the tenth. This pattern essentially may indicate that it might have started overfitting on the training data, given its poorer performance on unseen validation data.

Test Performance: By applying it to the whole dataset, the model was eventually able to reach a test accuracy of 53.90%; it is doing better than random guessing, but it certainly leaves a lot of room for improvement.

From the initial sets of training, the model showed some potential; in further training, however, the model continued to struggle to yield consistent high validation accuracy levels.

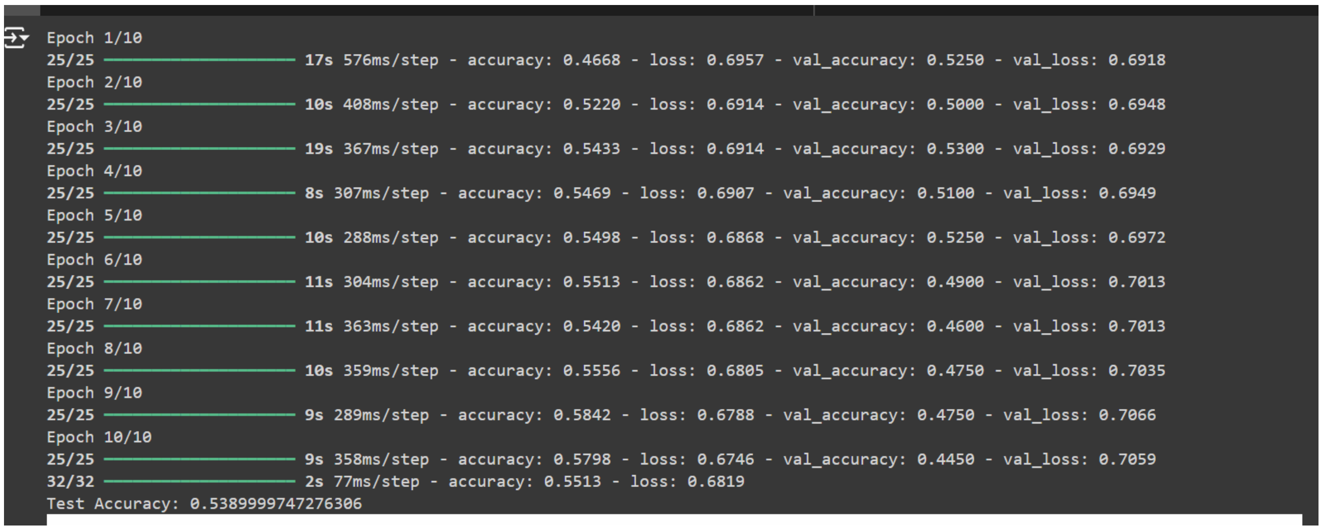
The increasing validation loss is indicative that overfitting may have occurred, and the model has learned specific patterns in the training data which do not generalize well on unseen data. The test accuracy of 53.90% reflects that this model is slightly above random guesses but may need some fine-tuning, with the probability of resorting to more sophisticated methods for improving its predictive capability.

Performance Metrics

Epoch	Accuracy During Training	Loss During Training	Accuracy During Validation	Loss During Validation
1	46.68%	0.6957	52.50%	0.6918

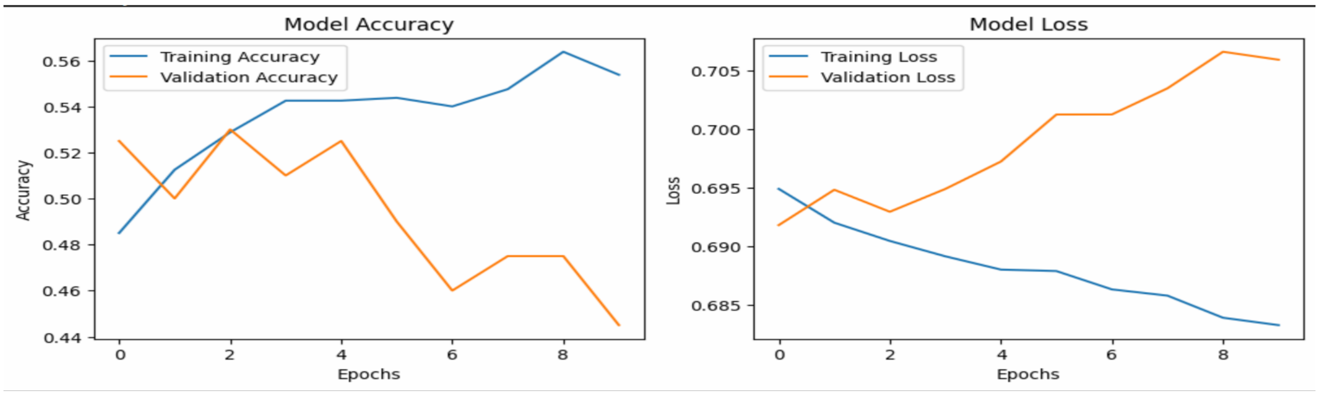
2	52.20%	0.6914	50.00%	0.6948
3	54.33%	0.6914	53.00%	0.6929
4	54.69%	0.6907	51.00%	0.6949
5	54.98%	0.6868	52.50%	0.6972
6	55.13%	0.6862	49.00%	0.7013
7	54.20%	0.6862	46.00%	0.7013
8	55.56%	0.6805	47.50%	0.7035
9	58.42%	0.6788	47.50%	0.7066
10	57.98%	0.6746	44.50%	0.7059

Below is the output generated by the model:



```
Epoch 1/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 17s 576ms/step - accuracy: 0.4668 - loss: 0.6957 - val_accuracy: 0.5250 - val_loss: 0.6918
Epoch 2/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 10s 408ms/step - accuracy: 0.5220 - loss: 0.6914 - val_accuracy: 0.5000 - val_loss: 0.6948
Epoch 3/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 19s 367ms/step - accuracy: 0.5433 - loss: 0.6914 - val_accuracy: 0.5300 - val_loss: 0.6929
Epoch 4/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 8s 307ms/step - accuracy: 0.5469 - loss: 0.6907 - val_accuracy: 0.5100 - val_loss: 0.6949
Epoch 5/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 10s 288ms/step - accuracy: 0.5498 - loss: 0.6868 - val_accuracy: 0.5250 - val_loss: 0.6972
Epoch 6/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 11s 304ms/step - accuracy: 0.5513 - loss: 0.6862 - val_accuracy: 0.4900 - val_loss: 0.7013
Epoch 7/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 11s 363ms/step - accuracy: 0.5420 - loss: 0.6862 - val_accuracy: 0.4600 - val_loss: 0.7013
Epoch 8/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 10s 359ms/step - accuracy: 0.5556 - loss: 0.6805 - val_accuracy: 0.4750 - val_loss: 0.7035
Epoch 9/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 9s 289ms/step - accuracy: 0.5842 - loss: 0.6788 - val_accuracy: 0.4750 - val_loss: 0.7066
Epoch 10/10
25/25 ━━━━━━━━━━━━━━━━━━━━ 9s 358ms/step - accuracy: 0.5798 - loss: 0.6746 - val_accuracy: 0.4450 - val_loss: 0.7059
32/32 ━━━━━━━━━━━━━━━━━━━━ 2s 77ms/step - accuracy: 0.5513 - loss: 0.6819
Test Accuracy: 0.5389999747276306
```

Below, you'll find visual representations of the model's accuracy and loss, created using Matplotlib. These plots illustrate how the model's performance evolved over time, showing both its accuracy improvements and loss reductions. This helps us better understand how well the model is learning and where it might need adjustments :



Comparison with Other Approaches

For the purpose of classifying code snippets in this research, we used a simple LSTM model which can be compared to other methods that have been used in earlier studies done in this area of research. Algorithm-based approaches like rule-based engine which scans through the code snippets using pre-defined patterns and rules about the code; although these systems are highly accurate in particular cases they are not efficient when it comes to the case of recognizing code not previously seen by the engine. As seen earlier, our LSTM model our LSTM model provided us with a moderate accuracy of 53. New coders, here ranked 90% better in terms of ability to cope with new code. However, it also has some phenomena of over-learning, that is, it works well on training sets and does not work so well on the new data set. As for the scalability, the size of a rule based system is becoming unmanageable as the rule base increases whereas the LSTM model, although would require some fine tuning, can be scaled more easily by introducing more data since the model does not rely on rules.

Compared with other traditional machine learning models, our LSTM model has its advantages and disadvantages while used in classifying codes, the strengths of which far outweighs that of Random Forests or Support Vector Machines (SVMs), which are typically applied to this area. Although these traditional models can result in high accurate result, a lot of effort needs to be spent on feature engineering to find out which features of the code they should use. In contrast, the LSTM proposed in our research learnt these features automatically, which makes the development process easier. But all the same, what it poorly offers in terms of accuracy is 53. 90% however, the effectiveness of our proposed model is still lower and needs to be tweaked to offer the efficiency of these conventional approaches. Further, traditional models could be rigid when used in various codes whereas the LSTM model does not require much feature engineering and can easily be generalized to different coding styles. State-of-the-art methods given concerns more challenging code patterns involve the utilization of deep learning techniques such as the transformers by applying BERT or GPT. Such models usually are more accurate and perform better than LSTMs, however, demands much more computational power. Despite the fact that our LSTM model was a bit less accurate, it is much simpler to implement and use less resources. Transformers, for instance, have been fine-tuned where they receive a huge number of computations and big quantities of data. However, our LSTM model is less powerful but more realistic to the projects which is having limited funding for implementation.

All in all, it is the simplicity and being free from manual feature engineering, and it is significantly less resource-demanding compared to the models like transformers. This is because our proposed LSTM model does not require any form of custom rule to be imposed on them due to flexibility of the model. However, there are a few short-coming that need to be hit; for example, overfitting in the model and further optimization of the model for the better performance. Even though our LSTM model has several benefits, there is a potential for further studies with the development of other methods that can increase the model’s accuracy and efficiency.

Conclusion

We present in this research paper the study done for classification of code snippets, which is an essential task in software quality enhancement and keeping coding standards. From our results, we show that our LSTM model produced an accuracy of 53.90% over the test data. This accuracy is not particularly high but does illustrate the inherent difficulty of trying to use a relatively simple deep learning model for the task of code classification. Actually, the model showed some sign of overfitting as it did way better with the training data in comparison to the out-of-sample data; therefore, it required improvement in refinement. So, use of an LSTM model has an added advantage as it can automatically learn the features inherent in the input data; hence, the task of feature engineering is greatly minimized, making the job simple for dealing with different code types compared to traditional machine learning methods.

In comparative analysis, our LSTM approach struck a good balance between complexity and performance in relation to rule-based systems and classical machine learning models. Whereas more advanced models, for example, Transformers, reach higher accuracy and deeper insight into complex structures, they also bring with them a massive computational cost and, therefore, usually need much more data, which makes these models less feasible for smaller projects. Although the challenges faced, our results suggest that LSTM models can be a decent choice for code classification, mostly when resources are scarce. There is certainly room for further improvement with model architecture optimization and dataset expansion or exploration of advanced model designs to transformer models. We will experiment more and update our iterative approach in this paper for the generalization of the model to handle the expanded range of coding scenarios with a higher level of precision.

Future work should be directed at regularization strategies that will take care of overfitting or on increasing the size of the training data and making it more diverse. Hybrid models, combining LSTM with transformer techniques, might provide a more comprehensive solution for code classification. Transfer learning may aid both accuracy and reduction in training time by pre-training on large quantities of code before fine-tuning for any specific task. In general, this research highlights that the high potential of using LSTM in source code analysis points to further development on how software quality can be maintained with respect to the coding standards within the domain of software engineering.

References

[1] Qayum, S. U. R. Khan, I. U. Rehman, and A. Akhunzada, "FineCodeAnalyzer: Multi-Perspective Source Code Analysis Support for Software Developer Through Fine-Granular Level Interactive Code Visualization," IEEE Access, vol. 10, pp. 20498-20513, 2022. DOI: 10.1109/ACCESS.2022.3151395.

[2] V. Bhutani, F. G. Toosi, and J. Buckley, "Analysing the Analysers: An Investigation of Source Code Analysis Tools," Applied Computer Systems, vol. 29, no. 1, pp. 98–111, June 2024. DOI: 10.2478/acss-2024-0013.

[3] T. Cerny, J. Huang, and others, "Code Analysis Opportunities and Challenges for Enterprise Systems and Microservices," IEEE Access, vol. 8, pp. 159464-159470, 2020. DOI: 10.1109/ACCESS.2020.3019985.

[4] K. Kuszczyński and M. Walkowski, "Comparative Analysis of Open-Source Tools for Conducting Static Code Analysis," Sensors, vol. 23, no. 7978, 2023.

[5] M. Hoq, P. Brusilovsky, and B. Akram, "SANN: A Subtree-based Attention Neural Network Model for Student Success Prediction Through Source Code Analysis," in Proc. 6th Educational

Data Mining in Computer Science Education (CSEDM) Workshop, Durham, United Kingdom, 2022. DOI: 10.5281/zenodo.6983496.

[6] Sotirov, "Automatic Vulnerability Detection Using Static Source Code Analysis," M.S. thesis, Dept. of Computer Science, Univ. of Alabama, Tuscaloosa, AL, 2005.

[7] F. Logozzo and M. Fähndrich, "On the Relative Completeness of Bytecode Analysis versus Source Code Analysis," Microsoft Research.

[8] N. Stulova, A. Blasi, A. Gorla, and O. Nierstrasz, "Towards Detecting Inconsistent Comments in Java Source Code Automatically," in SCAM'20: 20th IEEE International Working Conference on Source Code Analysis and Manipulation, 27 Sept. 2020.

[9] F. Schuckert, B. Katt, and H. Langweg, "Difficult SQLi Code Patterns for Static Code Analysis Tools," in Proc. 1st Model-Driven Simulation and Training Environment for Cybersecurity, 2020.

[10] R. L. de Souza, F. Z. Ferreira, and S. S. Botelho, "A Proposal for Source Code Assessment Through Static Analysis," PPG Modelagem Computational, Universidade Federal do Rio Grande, Rio Grande - RS, Brazil 2023.

[11] R. Sadik, A. Ceravola, F. Joublin, and J. Patra, "Analysis of ChatGPT on Source Code," Honda Research Institute Europe, 2023.

[12] M. Iammarino, L. Aversano, M. L. Bernardi, and M. Cimitile, "A Topic Modeling Approach to Evaluate the Comments Consistency to Source Code," 2020 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2020. DOI: 10.1109/ICSME.2020.0010.

[13] Md. Mostafizer Rahman, Yutaka Watanobe, Atsushi Shirafuji, and Mohamed Hamada, "Exploring Automated Code Evaluation Systems and Resources for Code Analysis: A Comprehensive Survey," J. ACM, vol. 37, no. 4, Art. 111, Aug. 2018.

[14] J. H. Suk, Y. B. Lee, and D. H. Lee, "SCORE: Source Code Optimization & REconstruction," IEEE Access, vol. 8, pp. 129478-129495, 2020, DOI: 10.1109/ACCESS.2020.30089051.

[15] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A Transformer-based Approach for Source Code Summarization", May 2020.