Explainable Artificial Intelligence with Hybrid Ensemble Learning-Based Automated Code Comprehension Prediction

Bharat Babaso Mane
Department of Computer Science and Engineering, Alliance University, Bengaluru, Karnataka, India
bharat.mane@gmail.com (corresponding author)

Rathnakar Achary
Department of Computer Science and Engineering, Alliance University, Bengaluru, Karnataka, India
rathnakar.achary@alliance.edu.in

Received: 3 April 2026 | Revised: 11 May 2026, 21 May 2026, and 25 May 2026 | Accepted: 26 May 2026

 DOI: https://doi.org/10.48084/etasr.19084

ABSTRACT

Code comprehension prediction is an interesting research area in software engineering, which employs Artificial Intelligence (AI) and Machine Learning (ML) algorithms to assess how easily a programmer can understand a piece of code. To ensure precise classification models, the preceding analysis mainly relies on handcrafted features. However, manual feature engineering is labor-intensive and can acquire only partial information about the source code, limiting model performance. Recently, many Deep Learning (DL)–based code readability classification approaches have been presented. This paper presents an Explainable Code Readability Classification using Vector Representations and Majority Voting-Based Ensemble Learning (ECRVR-MVEL) approach. The model initially preprocesses the input code and then uses CodeBERT to transform it into vector representations. For classification, a Weighted Majority Voting Ensemble (WMVE) integrates a Graph Convolutional Network (GCN), a Deep Belief Network (DBN), and a Bidirectional Temporal Convolutional Network (Bi-TCN). In addition, Nadam is applied to optimize the model and improve performance. Finally, Local Interpretable Model-agnostic Explanations (LIME) is utilized to visualize the interpretability and ensure transparency. An extensive evaluation to determine the performance of the ECRVR-MVEL model on Python and C++ datasets demonstrates its promising results over existing methods.

Keywords-software quality; explainable artificial intelligence; deep learning; code comprehension; hyperparameter tuning; ensemble classification

I. INTRODUCTION

Understanding source code is one of the most frequent and critical activities in software engineering. Around 70% of the life-cycle cost of a software project is in the maintenance stage. The high cost of maintenance relates to the complexity of understanding and reading the source code, especially that authored by others, which underlines the significance of code readability [1]. Code readability recommends a human decision on the readability and comprehensibility of the program's source code [2]. Evaluating code readability snippets has been mandatory to assess quality. Developers assume code readability to be a key aspect in source code development activities, as they frequently consume a lot of time for reading and understanding source code [3]. To simplify recycling and generate maintainable and readable code, it is significant to consider specific features. Code developers utilize the readability score generated by models as a feature in the code snippet grading procedure [4]. Another method to measure the readability of code snippets is by the usage of ASATs like Sonar-Qube2, which provides a set of rules that can determine concerns on code readability violations.

The most critical challenge is the applied feature engineering approach. Previous research used—but was not restricted to—commonly hand-made surface-level features, such as line lengths or operator counts, to indicate code readability. Rather than manual design and feature extraction, DL-driven methods are used to automatically learn the most complex underlying features from the source code. More advanced methods are based on Machine Learning (ML) or Deep Learning (DL) methodologies, which must be trained on a labeled dataset. ML-driven models depend on feature engineering to determine textual, structural, and visual aspects and classify the source code. Datasets have been developed to train a binary classifier that could classify code snippets as "readable" or "unreadable." The latest literature analyzed the probability of developing DL-driven methods to enhance the accuracy of current models.

In [5], the readability of code was gauged according to subjective perceptions of developers, modifying the code readability methodology. This study adopted an LLM with less learning to attain the objective. CLAVE [6] is a DL method for source code verification of authorship that exploits a different approach to learning. CLAVE is fine-tuned for authorship verification utilizing DL on Python-based submissions by unique developers. In [7], the benefits and drawbacks of three C-to-Rust translators, namely C2Rust (a transpiler), C2SaferRust (an LLM-guided transpiler), and TranslationGym (an LLM-based direct translation), were examined. This study performed an extensive assessment of various essential features for the translated Rust code of the popular GNU coreutils, using human-based translation as a baseline. In [8], the effect of AI-created code on software quality was examined, focusing on readability, code complexity, quality, and documentation. CSGVD [9] is a DL-based method that denotes vulnerability at the function-level identification as a binary graph classification activity. Initially, a PE-BL element obtains and improves the information from the existing language model. Finally, Mean Biaffine-based Attention pooling (M-BFA) represents node data as a graph's feature illustration.

In [10], it was shown that Question-and-Answer (Q&A) websites such as Stack Overflow remain useful and vibrant, despite possible side effects. This model is moderately effective in using features to recognize question quality and identify low-quality ones that should be improved and revised. In this model, performance was improved for both low- and high-quality question classification based on a different type of data: the identification of code snippets in Stack Overflow questions. In [11], code readability and software complexity were studied as fundamental aspects of software quality. Software metrics influence factors such as maintenance and reusability. The maintenance process consumes a large portion of the software lifecycle cost and is considered a highly costly stage that must be given additional attention. For this reason, the importance of software complexity and code readability is discussed, considering maintenance as the most time-consuming phase in overall software maintenance activities.

To enhance the performance of code readability classification, this study presents a novel Explainable Code Readability Classification using Vector Representations and Majority Voting-Based Ensemble Learning (ECRVR-MVEL) approach. The contributions of this study are as follows:

- Employs CodeBERT-based embedding for transforming raw code into vectorized representations, which enhances code readability classification.
- Designs a Weighted Majority Voting Ensemble (WMVE) that incorporates three DL approaches, namely a Graph Convolutional Network (GCN), a Deep Belief Network (DBN), and a Bidirectional Temporal Network (Bi-TCN), to classify code readability into high, medium, and low levels and improve the performance of the analysis.
- Utilizes the Nadam optimizer for hyperparameter tuning, which helps to accomplish improved performance of code readability classification with limited computational complexity.
- Utilizes LIME-based explainability to highlight code elements and provide interpretable explanations.

II. THE PROPOSED MODEL

The proposed ECRVR-MVEL method follows a methodological framework for code readability classification. Figure 1 shows the overall process of the ECRVR-MVEL approach, which involves text preprocessing, embedding using CodeBERT, weighted voting ensemble classification, optimization, and explainability analysis to improve model transparency and performance.

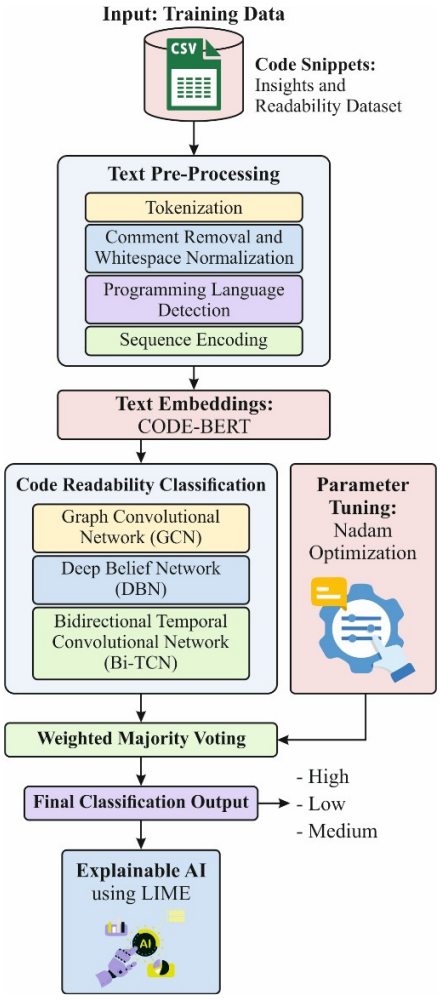

Fig. 1. Working pipeline of the proposed model.

At the initial stage, preprocessing includes tokenization, removing comments and normalizing white space, and detecting language and encoding sequences to clean and standardize raw code to increase model performance. The CodeBERT embedding model is used to convert raw source code into vector embedding representations that allow for an effective code readability assessment. The voting-based ensemble model integrates GCN, DBN, and BiTCN to classify the code into three readability classes, namely high, medium, and low. The Nadam optimizer is used to improve the

classification performance and the convergence speed of the model. Finally, a LIME-based explainable approach is used to visualize the transparency of the model.

A. Text Preprocessing

Preprocessing is a critical phase for preparing raw source code for efficient examination and subsequent application of ML and DL models. Since programming languages show structured rules and syntax, traditional NLP models can be carefully adapted. In this method, C++ and Python source codes are key inputs, and a dedicated preprocessing pipeline is used to preserve syntactic integrity while improving performance and model interpretability.

1) Tokenization

In the initial phase of preprocessing, the raw source code is tokenized to order valid tokens. In particular, keywords (for example, while, if, return, for), identifiers (function names, class names, and variable names), literals (string and numeric constants), operators (logical, relational, and arithmetic), punctuation symbols, and delimiters are obtained as separate tokens. Language-specific tokenization is used to ensure precise parsing of C++ and Python syntax, thus preserving both structural and lexical data vital for downstream learning challenges.

2) Comment Removal and Whitespace Normalization

Source code frequently covers irregular formatting and comments, which are unrelated to functional semantics. To reduce noise and enhance representation consistency, multi-line as well as single-line comments are optionally eliminated in preprocessing. Moreover, redundant line breaks, excessive whitespace, and unreliable indentation are normalized. Although indentation has been syntactically crucial in Python, normalizers are made carefully to retain logical structure. This phase enhances robustness, decreases vocabulary size, and improves learning efficacy without changing program action.

3) Programming Language Detection

As the framework works with both C++ and Python inputs, an automatic language classification model is used before encoding tokens. Language classification is obtained by utilizing file extensions, reserved keyword analysis, and syntactic patterns. This allows the selection of proper tokenization rules and encoding approaches, assuring that language-specific features are captured accurately.

4) Sequence Encoding

After tokenization, the extracted tokens are mapped into numerical representations appropriate for model ingestion. All tokens are encoded by employing a vocabulary created from the training corpus, and sequences are truncated or padded to a fixed length to uphold uniform input dimensions. Language-aware embedding is used to capture syntactic dependency and semantic relations inside the code. Its encoded representation acts as a normalized and structured input for subsequent feature extractors and classifier mechanisms.

B. CodeBERT-Based Embeddings

Code embedding allows the comprehension of logical relations and semantic codes. To efficiently capture the complexities of code, code embedding is used to solve two crucial considerations: (i) Vulnerable code components consist of essential elements, such as conditional judgments, as well as variable declarations, which are taken together from other essential positions; (ii) Code statements inside a function, logically and consecutively relevant, need embedding to maintain longer-distance dependency and uphold semantical precision.

CodeBERT [12] was used for code embedding, employing the multi-layer Transformer to produce semantical depictions of source code. The multi-head self-attention model at the heart of CodeBERT calculates token-level relations for capturing variable dependencies and control flow data. CodeBERT focuses on crucial code elements efficiently, enabling feature extraction through different significant areas.

Capturing the longer-distance relations inside the code, the encoded position is integrated into the token embedding of the input layer. To add an encoded position for the token vectors, the method preserves the best semantic and structural features, taking logical connections through distant code statements.

$$PE_{(pos,2i)} = \sin\left(\frac{pos}{10000^{\frac{2i}{d_{mode1}}}}\right) \quad (1)$$

$$PE_{(pos,2i+1)} = \cos\left(\frac{pos}{10000^{\frac{2i}{d_{mode1}}}}\right) \quad (2)$$

C. Weighted Voting-Based Ensemble Classification

The Weighted Voting-based Ensemble Model (WVEM) integrates GCN, DBN, and Bi-TCN to classify code readability levels. In this method, classifications that have shown high outcomes in prior evaluations are awarded maximal weights that enhance performance. The decision-making method employs the weighted strategy to ensure that a more consistent classification has more impact on the final decision.

$$WC = \frac{AC}{\Sigma AN} \quad (3)$$

1) GCN

In GCN [13], the embedded vectorization of text and image nodes is upgraded to integrate the connection data from adjacent nodes. In the GCN model, the text and image node embeddings are continually updated via adjacent node details. The images denote visual representations attained from source code snippets, comprising syntax-aware code visualizations and cropped code-image segments obtained during preprocessing. The visual features complement the textual embeddings derived from the source code. The edge weights among the text and image nodes are determined by the use of cosine similarity for deriving semantic correlations between textual descriptions and visual code features. The GCN propagates data across neighboring nodes up to three hops, allowing the final node embeddings to incorporate contextual information from both textual and visual neighborhoods. Moreover, the weight of the edges is decided by the cosine

similarity between the sentences and the cropped images. Therefore, this method upgrades every node's data more visibly while the relations among texts and images are nearby. The GCN model consists of three layers, and l implies the values of $1, 2,$ or 3. For $l = 1$, x_i^{l-1} denotes the early embedding obtained via BLIP. In the final layer ($l = 3$), x_i^L denotes the ending node embedded for each node i.

After attaining the final embedded vectors for each node, these are aggregated into a graph-level embedded vector for the graph-level categorizations. Using mean pooling, the graph-level embedded vector has the mean of each node's embedded vectors. The global mean pooling vector for each node and for subject s implies h_s as.

$$h_s = \frac{1}{2}\left(\langle h_{s,i}^v \rangle + \langle h_{s,j}^t \rangle\right) \quad (4)$$

where $\langle \cdot \rangle$ implies the operation of mean, $h_{s,i}^v$ signifies the final image embedding, and $h_{s,j}^t$ denotes the text node embedding.

$$y = Linear(h_s) \quad (5)$$

2) DBN

This model offers an unsupervised hierarchical feature extractor. The method contains various RBM layers, which are probabilistic NNs that learn the model of input information through unsupervised training. Every layer gradually abstracts characteristics that, in turn, allow for discovering intricate patterns. In this stage, the DBN structure comprises L stacked RBM layers. The results of an RBM layer are input to the succeeding one. This configuration allows the gradual abstraction of features, allowing the DBN to learn gradually high-level and intricate forms of input data. The joint distribution among the hidden and visible units categorizes the probabilistic behavior of all RBM layers.

$$P(v,h) = \frac{1}{Z} ex\left(-E(v,h)\right) \quad (6)$$

where Z denotes a partition function and $E(v,h)$ signifies the energy function.

$$E(v,h) = -v^T W h - b^T v - c^T h \quad (7)$$

3) BiTCN

This is a DL model that is intended for improving longer time series forecasts, structured into three significant elements: feature fusion layer, residual connections, and bidirectional temporal blocks. This model uses dilation convolution as a primary function. Backward convolution, which is a significant element of the BiTCN, presents a further pathway that allows the network to integrate both past dependencies and upcoming covariates. The arithmetic model of backward and forward dilated convolutions is:

$$H_f(t) = \sum_{i=0}^{k-1} h_f(i) \cdot x_{t-d_m \cdot i} \quad (8)$$

$$H_b(t) = \sum_{i=0}^{k-1} h_b(i) \cdot x_{t+d_m \cdot i} \quad (9)$$

where $H_f(t)$ denotes the forward outcome of BiTCN, and $H_b(t)$ implies the backward outcome of BiTCN, $h_f(i)$ and $h_b(i)$ signify the forward and backward components in the convolution kernels, k is the dimension of the kernel, x_t represents the input, and $d_m \cdot i$ implies the expansion factor. A feature fusion layer combines backward and forward outcomes into a unified representation. This method efficiently utilizes data from both temporal perceptions, improving the feature extractor and enhancing prediction outcomes.

Table I shows the parameter settings for the three DL models.

TABLE I. PARAMETER CONFIGURATIONS

Model	Layers	Neurons/ Filters	Activation	Additional details
GCN	3 GCN	128, 64, 32	ReLU	Cosine similarity edges and global mean pooling
DBN	3 Hidden	256, 128, 64	Sigmoid	RBM-based feature learning
BiTCN	2 Conv	128, 64 filters	ReLU	Dilated convolutions, dropout, and Softmax output

D. Nadam-Driven Performance Enhancement

Nesterov-accelerated Adaptive Moment Estimation (Nadam) [14] is an expansion of Adam optimization, which combines NAG into the momentum element of Adam. In contrast to standard momentum upgrades, Nadam uses Nesterov's lookahead approach, where the gradient is calculated after velocity. This enables the optimization for anticipating the additional parameters' position, leading to constant upgrades and a more responsive system.

E. LIME-Based Interpretability Analysis

LIME [15] is a local model-independent explanation method for ML models, which explains the prediction for a particular example. By perturbing and sampling the example in the feature space, LIME processes a model interpretation about the sample of interest, like a linear regression method. Interpretable models estimate the activities of ML models on a local scale and are used for explaining the performance. Depending on the involvement of all features, LIME builds a boundary about the vital portion of the input that will have a higher weight in the decision-making procedure.

III. PERFORMANCE VALIDATION

The ECRVR-MVEL method was evaluated by employing the Code Snippets: Insights and Readability dataset [16]. This is a collection of C++ and Python code snippets annotated with various code properties. Table II shows details on the dataset.

TABLE II. DATASET DETAILS

Category	Python	C++	Combined readability score
Low Readability	561	502	—
Medium Readability	560	500	—
High Readability	560	502	—
Total Samples	1681	1504	—
Mean	—	—	3.79
Standard Deviation	—	—	1.72
Minimum	—	—	-7.19
Maximum	—	—	7.15
25% (Low)	—	—	Score ≤ 2.81
50% (Medium)	—	—	2.81 < Score ≤ 5.02
75% (High)	—	—	Score > 5.02

To evaluate the performance of the proposed model, a detailed comparison was made with baseline classifiers such as Decision Tree (DT) [10], Logistic Regression (LR) [10], Random Forest (RF) [10], Naïve Bayes (NB) [11], Bayesian Network (BN) [11], Support Vector Machine (SVM) [11], and Neural Network (NN) [11]. These models were selected because they are commonly used benchmark approaches in software analytics, vulnerability prediction, and classification-based intelligent systems. In addition, the chosen models indicate various learning paradigms such as statistical, probabilistic, ensemble, kernel-based, and neural learning, allowing a detailed validation of the proposed model. The DT model carries out hierarchical recursive partitioning of features for classification, whereas LR determines class probabilities via a sigmoid-based linear decision function. RF is an ensemble learning model that incorporates many multiple DTs using bootstrap aggregation and majority voting. NB and BN are probabilistic learning approaches that depend on Bayes theorem, and BN defines dependencies among variables. The SVM designs an optimum separating hyperplane by the use of kernel-based learning, whereas the NN defines a multilayer feedforward neural architecture trained using backpropagation learning. To ensure effective comparison, the existing methods were reimplemented and validated under an identical experimental setup, preprocessing, and evaluation measures. The dataset was split into training/testing data using a uniform split ratio of 80:20. The experimental settings involved in the baseline models are shown in Table III.

TABLE III. EXPERIMENTAL SETTINGS

Method	Experimental settings
DT	Gini criterion
LR	L2 regularization
RF	100 trees, bootstrap aggregation
NB	Gaussian probability model
BN	Bayesian dependency inference
SVM	RBF kernel, C = 1.0
NN	2 hidden layers (128, 64), ReLU, Nadam, LR = 0.001
Common settings	Train-test split = 80:20, Batch size = 32, Epochs = 100

Table IV shows a brief comparison of the models on the Python dataset. The results illustrate that the DT, LR, RF, and NB models exhibited lower classification outcomes. The ECRVR-MVEL method illustrates promising performance with $accur_y$ of 98.15%, $preci_n$ of 97.23%, $recal_l$ of 97.24%, and $F1_{score}$ of 97.21%.

TABLE IV. COMPARATIVE STUDY OF ECRVR-MVEL WITH EXISTING APPROACHES ON PYTHON CODE DATA

Python Data					
Methods	$Accur_y$	**Error Rate**	$Preci_n$	$Recal_l$	$F1_{score}$
DT [10]	60.40	39.60	77.98	78.88	78.43
LR [10]	65.10	34.90	83.59	71.08	76.83
RF [10]	69.20	30.80	77.25	74.93	76.07
NB [11]	88.58	11.42	89.35	90.70	90.02
BN [11]	87.02	12.98	85.17	83.13	84.14
SVM [11]	89.62	10.38	88.43	76.42	81.99
NN [11]	90.11	9.89	83.01	80.60	81.79
Proposed	98.15	1.85	97.23	97.24	97.21

Table V shows the comparative analysis of ECRVR-MVEL with the other models on CPP data. The simulation results show that the ECRVR-MVEL achieved higher performance, with $accur_y$ at 98.38%, where the LR, RF, NB, BN, SVM, DT, and NN methods achieved $accur_y$ at 92.84%, 69.23%, 78.04%, 94.58%, 73.20%, 76.00%, and 88.58%, respectively.

TABLE V. COMPARATIVE STUDY OF ECRVR-MVEL WITH EXISTING APPROACHES USING CPP CODE DATA

CPP Data					
Methods	$Accur_y$	**Error Rate**	$Preci_n$	$Recal_l$	$F1_{score}$
DT [10]	92.84	7.16	90.61	80.56	85.29
LR [10]	69.23	30.77	69.26	83.17	75.58
RF [10]	78.04	21.96	73.55	72.22	72.88
NB [11]	94.58	5.42	73.54	69.54	71.48
BN [11]	73.20	26.80	86.09	92.89	89.36
SVM [11]	76.00	24.00	80.98	79.90	80.44
NN [11]	88.58	11.42	85.35	87.77	86.54
Proposed	98.38	1.62	97.61	97.60	97.59

IV. CONCLUSION

This study presented the ECRVR-MVEL approach as an efficient model for code readability analysis to enhance software quality, maintainability, and developer productivity. The novelty of the proposed work involves the incorporation of advanced preprocessing techniques, CodeBERT-based feature extraction, and a WMVE classification framework for improved readability detection. The proposed WMVE classifier can differentiate code readability into high, medium, and low levels, while the Nadam optimizer boosts the convergence speed and overall model efficiency. Moreover, the inclusion of the LIME approach helps to increase the transparency and interpretability of the framework by clarifying the involvement of important features in prediction outcomes. Experimental results and comparative analysis demonstrate that the proposed ECRVR-MVEL model outperforms existing approaches with accuracies of 98.15% and 98.38% on Python and CPP datasets, respectively, highlighting its significance and contribution to the field of automated software quality assessment.

In the future, the proposed ECRVR-MVEL framework can be extended by the inclusion of multiple advanced transformer-based architectures and attention strategies to further improve readability classification results. The framework can also be extended to support multilingual programming environments and large-scale industrial software repositories. Furthermore, real-time readability feedback and automated code improvement recommendations can be integrated to make it useful for real-time software development applications.

DECLARATION OF COMPETING INTERESTS

The authors declare that they have no competing interests.

ACKNOWLEDGMENT

Not Applicable.

DATA AVAILABILITY

The dataset used in this study is openly available at [16].

REFERENCES

[1] J. Johnson, S. Lubo, N. Yedla, J. Aponte, and B. Sharif, "An Empirical Study Assessing Source Code Readability in Comprehension," in *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, Sept. 2019, pp. 513–523, https://doi.org/10.1109/ICSME.2019.00085.

[2] R. P. L. Buse and W. R. Weimer, "Learning a Metric for Code Readability," *IEEE Transactions on Software Engineering*, vol. 36, no. 4, pp. 546–558, July 2010, https://doi.org/10.1109/TSE.2009.70.

[3] T. Kanoutas, T. Karanikiotis, and A. L. Symeonidis, "Enhancing Code Readability through Automated Consistent Formatting," *Electronics*, vol. 13, no. 11, May 2024, Art. no. 2073, https://doi.org/10.3390/electronics13112073.

[4] D. Tosi, "Studying the Quality of Source Code Generated by Different AI Generative Engines: An Empirical Evaluation," *Future Internet*, vol. 16, no. 6, May 2024, Art. no. 188, https://doi.org/10.3390/fi16060188.

[5] A. Vitale, E. Guglielmi, R. Oliveto, and S. Scalabrino, "Personalized Code Readability Assessment: Are We There Yet?" arXiv, 2025, https://doi.org/10.48550/ARXIV.2503.07870.

[6] D. Álvarez-Fidalgo and F. Ortin, "CLAVE: A deep learning model for source code authorship verification with contrastive learning and transformer encoders," *Information Processing & Management*, vol. 62, no. 3, May 2025, Art. no. 104005, https://doi.org/10.1016/j.ipm.2024.104005.

[7] B. Tadesse, V. Nitin, M. Salah, B. Ray, M. d'Amorim, and W. Assunção, "Code Quality Analysis of Translations from C to Rust." arXiv, 2026, https://doi.org/10.48550/ARXIV.2602.00840.

[8] H. Fawareh, H. M. Al-Shdaifat, A.-R. M, F. A. Fawareh, and M. Khouj, "Investigates the Impact of AI-generated Code Tools on Software Readability Code Quality Factor," in *2024 25th International Arab Conference on Information Technology (ACIT)*, Dec. 2024, pp. 1–5, https://doi.org/10.1109/ACIT62805.2024.10876897.

[9] W. Tang, M. Tang, M. Ban, Z. Zhao, and M. Feng, "CSGVD: A deep learning approach combining sequence and graph embedding for source code vulnerability detection," *Journal of Systems and Software*, vol. 199, May 2023, Art. no. 111623, https://doi.org/10.1016/j.jss.2023.111623.

[10] M. Duijn, A. Kucera, and A. Bacchelli, "Quality Questions Need Quality Code: Classifying Code Fragments on Stack Overflow," in *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, May 2015, pp. 410–413, https://doi.org/10.1109/MSR.2015.51.

[11] Y. Tashtoush, N. Abu-El-Rub, O. Darwish, S. Al-Eidi, D. Darweesh, and O. Karajeh, "A Notional Understanding of the Relationship between Code Readability and Software Complexity," *Information*, vol. 14, no. 2, Jan. 2023, https://doi.org/10.3390/info14020081.

[12] Z. Feng *et al.*, "CodeBERT: A Pre-Trained Model for Programming and Natural Languages," in *Findings of the Association for Computational Linguistics: EMNLP 2020*, Aug. 2020, pp. 1536–1547, https://doi.org/10.18653/v1/2020.findings-emnlp.139.

[13] T. N. Kipf and M. Welling, "Semi-Supervised Classification with Graph Convolutional Networks." arXiv, 2016, https://doi.org/10.48550/ARXIV.1609.02907.

[14] T. Dozat, "Incorporating Nesterov Momentum into Adam," in Proceedings of the 4th International Conference on Learning Representations, 2016, pp. 1–4.

[15] M. T. Ribeiro, S. Singh, and C. Guestrin, "'Why Should I Trust You?': Explaining the Predictions of Any Classifier," in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, May 2016, pp. 1135–1144, https://doi.org/10.1145/2939672.2939778.

[16] "Code Snippets: Insights and Readability." [Online]. Available: https://www.kaggle.com/datasets/paakhim10/code-snippets-insights-and-readability.