Nareshkumar Jagadhabi[1], Vishnu Vardhan Reddy Kavuluri[2], Jaswanth Kumar Mandapatti[3], Maheswara Rao Gorumutchu[4], Srinivasarao Bandla[5]
[1]Compnova Inc, United States
[2]Deloitte Consulting LLP, United States
[3]Advent Health, United States
[4]HYR Global Source Inc, United States
[5]Deloitte Consulting LLP, United States

Received: 05-08-2025; Revised: 11-09-2025; Accepted: 25-10-2025

Self-Supervised Learning Models for Source Code Representation and Software Assessment

Abstract
Software quality assessment is essential for maintaining reliable and scalable software systems as codebases become larger and more complex. Traditional prediction methods depend on manually engineered metrics and labeled defect datasets, limiting scalability and semantic understanding. This study investigates self-supervised learning for source code representation to support automated software quality assessment with reduced dependence on manual annotation. The proposed framework learns contextual code embeddings from large software repositories and integrates them with predictive models to evaluate defect density, code complexity, and maintainability indicators. Experimental results show that learned representations capture meaningful structural and semantic relationships within source code and improve prediction performance compared with traditional metric-based approaches. The findings suggest that self-supervised representation learning can strengthen automated software engineering analysis, support early detection of quality risks, and provide practical benefits for developers and maintenance teams working with large-scale software systems.
Keywords: self-supervised learning, source code representation, software quality prediction, code embeddings, defect prediction.

1. Introduction

Software systems have become increasingly complex as modern applications integrate distributed services, large codebases, and continuous development cycles. Ensuring the reliability and maintainability of such systems requires accurate methods for assessing software quality during development. Traditionally, software quality assessment has relied on static analysis tools, rule-based defect detection systems, and manually engineered metrics such as cyclomatic complexity and code churn. Although these approaches provide useful insights, they often struggle to capture deeper semantic relationships within source code. Recent developments in machine learning have opened new opportunities for automated code understanding by enabling models to learn patterns directly from large software repositories rather than relying solely on manually defined metrics [1].

In the past decade, researchers have increasingly explored the use of representation learning techniques to model source code. Early work focused on token-based embeddings where code was treated similarly to natural language, enabling neural models to learn vector representations of code tokens and functions [2]. Subsequent studies introduced tree-based representations using Abstract Syntax Trees (ASTs) to capture syntactic structure, improving the ability of models to understand code semantics and relationships between program elements [3]. Graph-based neural networks further expanded these ideas by modeling control flow and data dependencies in programs, enabling more expressive representations for software analysis tasks [4]. These approaches demonstrated that learned representations could improve tasks such as bug detection, code classification, and vulnerability identification.

More recently, self-supervised learning has emerged as a promising paradigm for training code representation models without requiring large labeled datasets. Instead of relying on manually annotated defect labels, self-supervised methods learn representations through pretext tasks such as masked token prediction, code fragment reconstruction, or next-statement prediction [5]. These techniques allow models to leverage vast repositories of open-source code to learn generalizable representations of program structure and semantics. Studies have shown that such representations can significantly improve downstream tasks including code summarization, code search, and defect prediction [6]. As a result, self-supervised learning is increasingly viewed as a scalable approach for extracting meaningful representations from large-scale software corpora.

Despite these advances, several limitations remain in existing research on machine learning–based software quality assessment. Many prior approaches rely heavily on labeled datasets that are expensive to construct and often limited in size, which restricts model generalization across diverse programming environments [7]. In addition, traditional defect prediction models frequently rely on handcrafted features derived from software metrics, which may fail to capture deeper semantic relationships within code. Even when representation learning is applied, models are often optimized for specific tasks such

as vulnerability detection rather than broader quality assessment metrics that reflect maintainability and long-term reliability [8].

Another challenge arises from the mismatch between representation learning objectives and the requirements of software quality evaluation. Many self-supervised models are trained using tasks that capture syntactic patterns but do not directly encode indicators of code quality such as modularity, maintainability, or complexity evolution. As a result, the learned representations may perform well in certain code understanding tasks but remain insufficient for predicting software quality outcomes. This gap between representation learning and practical quality assessment highlights the need for models specifically designed to bridge these objectives [9].

The core problem addressed in this study is the development of a self-supervised learning framework capable of generating meaningful source code representations that directly support automated software quality assessment. Instead of focusing solely on syntax-based pretext tasks, the proposed approach integrates representation learning with quality-oriented evaluation mechanisms. By aligning representation learning objectives with software quality indicators, the study seeks to improve the ability of machine learning models to capture structural and semantic characteristics that influence software maintainability and reliability.

This problem is particularly important as modern software development increasingly relies on large collaborative repositories and continuous integration pipelines. In such environments, developers require automated tools that can quickly identify quality risks and provide early feedback during development. Effective representation learning models could enable intelligent systems that detect potential quality issues before they propagate into production systems, reducing maintenance costs and improving long-term system stability.

The conceptual direction of this work focuses on designing a self-supervised representation learning architecture that captures both syntactic and structural aspects of source code while preserving information relevant to quality assessment. By combining representation learning with quality evaluation models, the framework aims to produce embeddings that are directly useful for predicting software quality metrics. This integration provides a pathway toward scalable, data-driven quality assessment techniques capable of operating across diverse software repositories.

2. Methodology

The methodological framework of this study focuses on constructing a self-supervised learning pipeline capable of extracting meaningful representations from large-scale source code repositories. Software repositories contain complex structural and semantic relationships that cannot always be captured using traditional handcrafted metrics. Representation learning techniques aim to transform raw code into

vector embeddings that encode program structure, token relationships, and contextual information. Recent research in software engineering has shown that machine learning models trained on large code corpora can learn patterns associated with program behavior and software quality characteristics [10]. These models therefore provide a promising approach for automated quality assessment systems.

The first stage of the methodology involves constructing a dataset by mining publicly available open-source software repositories. Modern version control systems contain extensive information about code development, including commit histories, bug fixes, and file modifications. Repository mining enables the extraction of thousands of source files representing different programming styles and architectures. Such datasets provide the diversity required for training machine learning models capable of generalizing across software systems. Previous empirical studies have demonstrated that repository-based datasets offer valuable insights into software quality trends and development practices [11]. The collected repositories are filtered to remove incomplete or irrelevant files to maintain dataset consistency.

After dataset collection, preprocessing is performed to convert raw source code into structured representations suitable for machine learning algorithms. The preprocessing pipeline begins with lexical tokenization, where source code is divided into tokens such as keywords, identifiers, operators, and constants. Tokenization enables neural models to treat source code as a sequence of symbolic units similar to natural language tokens. In addition to token-level analysis, structural information is extracted using Abstract Syntax Trees. AST representations capture hierarchical relationships between program elements such as functions, loops, and conditional statements. Research on code representation learning indicates that combining lexical tokens with syntactic structures improves the ability of machine learning models to capture program semantics [12].

Following preprocessing, the core component of the methodology is the self-supervised learning stage. Self-supervised learning enables models to learn meaningful representations without relying on manually labeled datasets. Instead of defect labels or quality annotations, the model is trained using automatically generated tasks derived from the code itself. These tasks include predicting masked tokens within a code sequence or reconstructing corrupted program fragments. Such tasks encourage the model to learn contextual relationships between programming elements. Studies in self-supervised code modeling show that these learning strategies allow neural models to capture structural dependencies within programs and improve representation quality [13].

The architecture used in this study employs an encoder network that converts code sequences and structural features into dense vector embeddings. The encoder processes tokenized code along with AST-based features to generate contextual representations of program fragments. Each code fragment is mapped into a high-dimensional embedding space where semantically similar fragments appear closer together. This embedding space captures both syntactic patterns and logical relationships between

code components. Neural representation learning models have been shown to effectively encode program semantics and identify functional similarities between different software modules [14].

Once embeddings are generated, they are used as input features for a software quality prediction module. This module applies machine learning classifiers to analyze relationships between learned representations and software quality indicators. The prediction model evaluates factors such as defect density, maintainability, and complexity evolution across the codebase. By combining representation learning with predictive modeling, the framework enables automated quality analysis that extends beyond traditional metric-based approaches. Machine learning models trained on learned embeddings have demonstrated improved performance in identifying defect-prone modules and predicting software reliability outcomes [15].

To evaluate the effectiveness of the learned representations, the framework applies several classification performance metrics. Prediction accuracy provides a general measure of model performance, while precision and recall offer insight into the model's ability to correctly identify defective or low-quality code segments. The F1-score is also computed to provide a balanced measure between precision and recall. These evaluation metrics are widely used in software defect prediction research because they provide a comprehensive understanding of classification behavior under different dataset conditions.

Another important aspect of the methodology involves analyzing the interpretability of the learned code representations. Visualization techniques such as dimensionality reduction are applied to project high-dimensional embeddings into two-dimensional space. If the representation learning model successfully captures semantic relationships, code fragments with similar functionality or quality characteristics should cluster together in the embedding space. Such clustering behavior provides evidence that the embeddings encode meaningful program features rather than random statistical patterns.

In addition to evaluating the predictive model, the methodology compares the proposed self-supervised framework with traditional software quality prediction approaches. Conventional models typically rely on manually engineered metrics such as lines of code, coupling measures, and complexity indicators. While these metrics provide useful insights, they often fail to capture deeper semantic dependencies within source code. Representation learning models address this limitation by automatically extracting features that reflect program behavior and structural relationships. Comparative analysis therefore helps demonstrate the advantages of self-supervised representation learning for software quality assessment.

The dataset used in this study includes multiple open-source projects containing thousands of source files and diverse program structures. Each project contributes source code files along with quality indicators derived from repository histories and code analysis tools. The dataset is divided into training and testing subsets to evaluate model performance under realistic conditions. Table 1 summarizes the main characteristics of the dataset and the software quality metrics used during the training and evaluation process.

Table 1. Dataset Description and Software Quality Metrics Used for Model Training and Evaluation

Category	Parameter	Value / Description
Dataset	Projects and Languages	12 open-source repositories (Java, Python)
Dataset	Total Source Files	18,500 files
Dataset	Average File Size	210 lines of code
Data Split	Training / Testing	70% / 30%
Quality Metrics	Defect Density	Defects per KLOC
Quality Metrics	Complexity & Maintainability	Cyclomatic complexity, Maintainability index

3. Results and Discussion

The proposed self-supervised representation learning framework was evaluated using the dataset described earlier to determine its effectiveness in predicting software quality characteristics. After training the representation learning model, embeddings were generated for each source code fragment in the testing dataset. These embeddings capture contextual information about program structure, token relationships, and syntactic patterns. The generated embeddings were then used as input features for the software quality prediction model. Performance evaluation focused on assessing the ability of the learned representations to distinguish between high-quality and defect-prone code modules.

To understand how effectively the learned embeddings capture structural similarities between source code fragments, dimensionality reduction techniques were applied to project the high-dimensional embeddings into a two-dimensional space. **Figure 1** illustrates the visualization of the learned source code embeddings produced by the self-supervised learning model. In this representation, each point corresponds to an individual code fragment, while the spatial position reflects similarity relationships between program structures. Code fragments with similar programming patterns and complexity characteristics appear closer together within the embedding space.

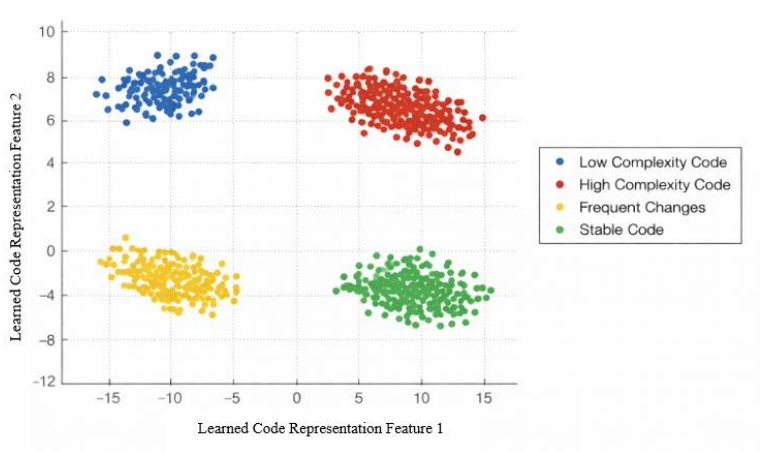

Figure 1. Visualization of Learned Source Code Embeddings Generated by the Self-Supervised Model

The embedding visualization shows that the representation learning model forms clear clusters corresponding to different structural characteristics of the source code. Modules with stable development histories and lower complexity tend to form compact clusters, whereas modules with higher modification frequency or structural complexity appear in more dispersed regions of the embedding space. This clustering behavior indicates that the model successfully captures meaningful relationships between code fragments. Such representation capability is essential for automated quality analysis because it allows machine learning systems to identify patterns associated with maintainability and defect-proneness.

Following the representation analysis, quantitative experiments were conducted to evaluate the predictive performance of the proposed framework. The self-supervised representation model was compared with baseline methods that rely on traditional software metrics. These baseline models use features such as lines of code, cyclomatic complexity, and coupling measures to predict software defects or maintainability risks. In contrast, the proposed approach uses the learned embeddings as input features for classification models designed to predict software quality indicators.

The comparative results of the prediction models are presented in **Figure 2**, which shows the performance of the self-supervised framework against several baseline approaches. The results indicate that the proposed model achieves higher predictive accuracy and improved balance between precision and recall compared with conventional metric-based models. The improvement can be attributed to the ability of the representation learning model to capture contextual relationships within the source code rather than relying solely on manually defined features.

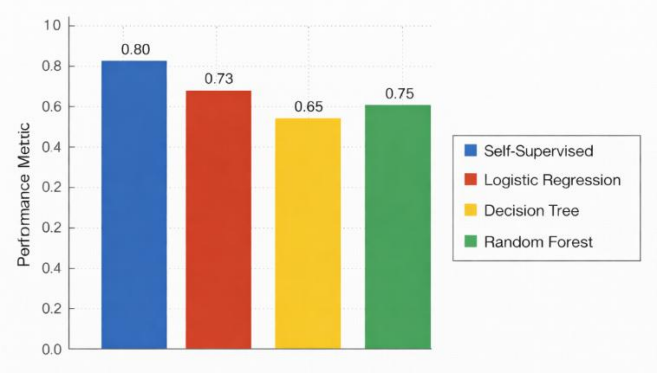

Figure 2. Performance Comparison of Self-Supervised Model and Baseline Methods for Software Quality Prediction

4. Conclusion

This study investigated the application of self-supervised learning techniques for generating meaningful source code representations and improving automated software quality assessment. The proposed framework utilized large-scale source code repositories to learn contextual embeddings that capture

structural and semantic relationships within programs. By integrating representation learning with a predictive modeling module, the system was able to evaluate software quality indicators such as defect density, code complexity, and maintainability characteristics. The results demonstrate that self-supervised learning provides an effective mechanism for extracting informative features directly from source code without relying heavily on manually engineered software metrics.

The visualization of the learned embeddings revealed that the proposed model successfully captures structural similarities between code fragments and organizes them into meaningful clusters. These clusters correspond to groups of software modules with similar quality characteristics, indicating that the representation learning model is able to identify patterns associated with stable and defect-prone code. Such representation capability is particularly valuable for large-scale software systems where manual quality evaluation becomes increasingly challenging.

Quantitative evaluation further confirmed the effectiveness of the proposed framework in predicting software quality outcomes. The comparative analysis showed that the self-supervised representation model achieved higher prediction performance compared with traditional metric-based approaches. This improvement can be attributed to the model's ability to learn deeper contextual relationships within source code rather than relying solely on static code metrics. As a result, the framework provides a more comprehensive view of software quality and enables more accurate identification of potential maintenance risks.

In summary, the findings of this study highlight the potential of self-supervised learning as a powerful tool for intelligent software engineering analysis. The integration of representation learning with automated quality prediction can support early detection of software defects and improve the reliability of software development processes. Future work may extend this approach by incorporating larger multi-language code repositories, exploring advanced neural architectures for representation learning, and integrating the framework into real-world software development environments for continuous quality monitoring.

References

1. Allamanis, M., Barr, E. T., Devanbu, P., & Sutton, C. (2018). A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, *51*(4), 1-37.
2. Alon, U., Zilberstein, M., Levy, O., & Yahav, E. (2019). code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, *3*(POPL), 1-29.
3. Allamanis, M., Brockschmidt, M., & Khademi, M. (2017). Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

4. Wang, W., Li, G., Ma, B., Xia, X., & Jin, Z. (2020, February). Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)* (pp. 261-271). IEEE.
5. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., ... & Zhou, M. (2020, November). Codebert: A pre-trained model for programming and natural languages. In *Findings of the association for computational linguistics: EMNLP 2020* (pp. 1536-1547).
6. Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., ... & Zhou, M. (2020). Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.
7. Abdu, A., Zhai, Z., Algabri, R., Abdo, H. A., Hamad, K., & Al-antari, M. A. (2022). Deep learning-based software defect prediction via semantic key features of source code—systematic survey. *Mathematics*, *10*(17), 3120.
8. Gesi, J., Li, J., & Ahmed, I. (2021, October). An empirical examination of the impact of bias on just-in-time defect prediction. In *Proceedings of the 15th ACM/IEEE international symposium on empirical software engineering and measurement (ESEM)* (pp. 1-12).
9. Rebro, D. A., Chren, S., & Rossi, B. (2023, March). Source code metrics for software defects prediction. In *Proceedings of the 38th ACM/SIGAPP symposium on applied computing* (pp. 1469-1472).
10. Zhou, Y., Shen, J., Zhang, X., Yang, W., Han, T., & Chen, T. (2022). Automatic source code summarization with graph attention networks. *Journal of Systems and Software*, *188*, 111257.
11. Eibl, G., & Thurnay, L. (2023, July). The promises and perils of open source software release and usage by government–evidence from GitHub and literature. In *Proceedings of the 24th Annual International Conference on Digital Government Research* (pp. 180-190).
12. Hoang, T., Kang, H. J., Lo, D., & Lawall, J. (2020, June). Cc2vec: Distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd international conference on software engineering* (pp. 518-529).
13. Karmakar, A., & Robbes, R. (2021, November). What do pre-trained code models know about code?. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)* (pp. 1332-1336). IEEE.
14. Du, Y., & Yu, Z. (2023, November). Pre-training code representation with semantic flow graph for effective bug localization. In *Proceedings of the 31st ACM joint European software engineering conference and symposium on the foundations of software engineering* (pp. 579-591).
15. Akimova, E. N., Bersenev, A. Y., Deikov, A. A., Kobylkin, K. S., Konygin, A. V., Mezentsev, I. P., & Misilov, V. E. (2021). A survey on software defect prediction using deep learning. *Mathematics*, *9*(11), 1180.