Development and Evaluation of a Deep Learning-Based Model for Source Code Quality Classification Using Industrial Data

Shuichi Tokumoto[1], Ryotaro Imai[1], and Shinji Kusumoto[2]

[1] Information Technology R&D Center, Mitsubishi Electric Corporation, Japan

[2] Graduate School of Information Science and Technology, The University of Osaka, Japan

Abstract

In recent years, there has been growing interest in automatic source code classification technologies to improve software productivity. However, many organizations face difficulties adopting machine learning solutions due to security constraints that restrict the use of online tools. This study aims to develop and validate a deep learning-based model capable of operating entirely within a secure corporate environment to classify the quality of source code. The model, referred to as the Source Code Quality Classification Model (SCQC model), was trained and evaluated using both open-source software (OSS) and internal source code. First, a training dataset was constructed from OSS repositories, and the resulting model achieved an accuracy of up to 82.1%. To examine its generalizability, the model was applied to internal source code. The accuracy declined significantly due to differences in code structure and development practices, highlighting the critical importance of domain alignment. Further experiments with internal data demonstrated that restricting the target scope by programming language and product category could improve prediction accuracy. These findings suggest that it is feasible to build practical classification models when training data is tailored to the specific characteristics of the development environment. The results indicate a promising direction for implementing such models in real-world settings. However, challenges remain, including the preparation of high-quality labeled training data and adapting models to specific domains. Future work will focus on addressing these issues and exploring integration of the SCQC model into actual code review and quality assurance workflows.

Keywords: Source Code Quality, Deep Learning, Code Quality Classification, Domain Adaptation, Industrial Software Development

1. Introduction

In recent years, the reuse-based development, where features are continuously extended based on existing software, has become mainstream in software development. This approach is commonly adopted in large and complex software systems to improve

development efficiency and reduce effort [1]. However, in reuse-based development, a lack of understanding of existing features and architecture, misjudgment of the scope of impact, and inadequate testing can lead to unintended defects or performance degradation, resulting in a decline in software quality. Prior research has reported correlations between structural complexity, change history metrics, and defects [2]. To prevent such quality degradation, it is essential to verify the reused software in advance and conduct impact analysis to assess how modifications affect existing functionality.

As software functionality continues to grow and become more complex, source code tends to expand. With repeated reuse, its readability often declines, leading to the breakdown of the modular structure. This structural degradation leads to decreased maintainability and complicates modification and maintenance tasks [3]. In many organizations, coding is outsourced to external contractors, and the design staff who place orders may not fully understand the structure or processing details of the source code [1]. Against this backdrop, it is crucial to understand the quality of the source code being reused early in the development process to mitigate development risks. Traditional methods for evaluating source code quality have relied on static analysis tools and software metrics (e.g., McCabe's cyclomatic complexity [4], Halstead metrics). However, these methods mainly visualize the structural characteristics and complexity of the code and do not directly predict the presence of defects or risks. As a result, the interpretation and judgment of evaluation results heavily depend on the experience and skills of developers and quality assurance personnel.

Recently, machine learning techniques—such as deep learning and generative AI—have made rapid progress, expanding beyond fields like natural language processing and image recognition to applications in software engineering as well [5][6]. This has increased expectations for applying these technologies to automate software quality assessment. However, many deep learning models rely on online knowledge bases, which poses security risks in corporate environments where source code and other confidential data cannot be sent externally. As a result, there is a growing need for training and operating models entirely within local environments, especially in companies handling sensitive software.

This study focuses on the applicability of machine learning models in actual software development environments within companies. The goal is to construct a lightweight defect classification model that can operate in limited environments and with minimal information. Using general open-source software (OSS) tools, we constructed a learning model that operates in local environments and designed and evaluated a practical source code quality assessment method. Specifically, we used past source code modification histories as training data and created a model to predict defects at the function level. The model learning and prediction processes were performed on a PC using methods and machine learning frameworks provided by OSS. To validate the

effectiveness of this approach, we conducted evaluation experiments using the company's own source code and discussed its accuracy, reproducibility, and applicability in business scenarios. In section 2, we review related research on software quality prediction, while section 3 discusses the challenges to be addressed in this study. Section 4 explains the proposed quality assessment model, and section 5 presents the validation results using OSS and internal source code. In section 6, we discuss these results, and in section 7, we conclude this study.

2. Related Work

Software quality prediction has been a key technology in quality assurance for software development. One of the most common methods for this is the use of software metrics. A representative study in this area is McCabe's cyclomatic complexity [4]. Software metrics quantify the structural features of source code, and developers use these values to predict the quality of the software. Similarly, many studies have been conducted using statistical methods. For instance, Nagappan et al. statistically analyzed the relationship between software metrics and defects and developed a defect prediction model [7].

With the development of data utilization in statistical methods, machine learning has gained attention for software quality prediction. In these approaches, predictive models have been developed to identify the presence of defects using features derived from source code and development process data. Khoshgoftaar et al. proposed a defect prediction model using neural networks, demonstrating higher prediction accuracy compared to traditional statistical methods [8]. Xing et al. performed quality prediction based on software metrics using Support Vector Machines (SVM) [9].

With the advancement of deep learning, its application to software defect prediction has also progressed. Deep learning has the advantage of being able to extract and learn more complex features compared to traditional machine learning. One key advantage is that it can automatically extract and learn structural and semantic features from source code. Pham et al. proposed a deep learning model based on LSTM by converting source code into Abstract Syntax Trees (ASTs) [10]. This approach enables defect prediction while preserving the syntactic structure of the source code. More recently, defect prediction methods using Transformer-based models have been studied, improving prediction accuracy by applying natural language processing techniques to source code analysis [11].

However, several challenges have been identified with the use of software metrics and statistical methods. In software metrics-based approaches, the evaluation is limited to the perspective of pre-defined software metrics, making it difficult to capture other features. Additionally, since the evaluation is performed from the perspective of the

evaluator, prediction accuracy tends to vary depending on the target project or domain. Statistical methods are also heavily dependent on how data is collected, its quantity, and its distribution, making general application difficult [7][9]. Furthermore, in machine learning-based quality prediction research using OSS, high accuracy is not always achieved. For example, while Pham et al.'s deep learning model showed some improvement in accuracy, it was not demonstrated whether that accuracy is useful in actual development environments.

Most existing methods have been validated mainly for research or OSS, with few applications in real corporate settings. Their effectiveness is often limited by the characteristics of the target software. In practice, technical and organizational barriers remain, such as data preparation, model accuracy, operational constraints, and cost.

This study addresses these issues by building and evaluating a deep learning model aimed at practical use within companies. Specific constraints and challenges are discussed in the next section.

3. Challenges in This Study

This study aims to construct a system for source code quality assessment (predicting the presence or absence of defects) using deep learning in a corporate software development environment. However, in order to implement and operate a practical deep learning model within a company, several technical and operational challenges must be addressed beforehand. This section outlines the constraints underlying this study and the major challenges that need to be resolved.

Operation in a Local Environment: Many software systems developed within companies contain sensitive information, such as proprietary data or customer details. As a result, using external cloud services or APIs for training and inference is difficult due to the risk of information leakage. Therefore, it is essential to construct and operate a model that can complete the entire process of training and inference within the company’s closed network environment. This requires the ability to build models using OSS frameworks for training, assuming that the process is self-contained.

Acquisition of Training Data: To construct a quality assessment model using deep learning, a large training dataset is necessary. For learning to determine the presence or absence of defects, labeled source code indicating good or bad quality is required. However, it is rare for source code within a company to have pre-existing defect labels. As a result, it is essential to have defect-related information that can be labeled, along with the corresponding source code management. This implies that effective defect management and version control systems are essential for generating training datasets.

Accuracy of Quality Assessment: Based on internal interviews, it was found that software development requires a prediction accuracy of 80% or higher. This requirement is based on the cost and risks of corrective actions that would be taken based on the prediction results. If the prediction accuracy is low, incorrect modifications or rework may occur, potentially leading to reduced work efficiency.

4. Evaluation Method

4.1. Basic Method

This study proposes a deep learning-based method for assessing source code quality, aiming to reduce subjectivity and reliance on developer experience. The method builds a *classification model*—a model that predicts whether a given unit of source code contains a defect (1) or not (0). The model is trained using small code units labeled with defect information, based on actual source code and related defect records. In this study, a function or method is used as the unit of classification.

For vectorizing the source code, the open-source Word2Vec [12][13] is used, and for constructing the classification model, TensorFlow [14][15] is employed. Word2Vec is a distributed representation learning technique in natural language processing that uses neural networks to learn the semantic similarity of words (in this case, source code tokens) as numerical vectors. TensorFlow is an open-source machine learning library developed by Google. It is easy to operate in a local environment and has the advantage of being suitable for adoption in corporate environments, thanks to extensive documentation and community support. Since uploading company source code to the cloud for training poses security risks, such as information leakage, TensorFlow, which can be fully utilized in a local environment, is adopted in this study to suit internal use.

Figure 1 shows the learning flow of the proposed source code quality prediction method in this study. The method consists of the following three processes:

Source Code Acquisition Process: The source code used for training is obtained from version control systems such as Git. By analyzing the modification history, the differences before and after each commit are examined to identify functions where defects were fixed. The version of the code before the fix is labeled as "defect present (1)," and the version after the fix as "defect absent (0)." This labeling process is performed automatically using a tool.

Training Data Creation Process: For the labeled functions, preprocessing steps such as removing blank lines and comments and tokenizing the code are applied. Then, distributed representations are generated using Word2Vec, resulting in each function being represented as a fixed-length vector suitable for input to the classification model.

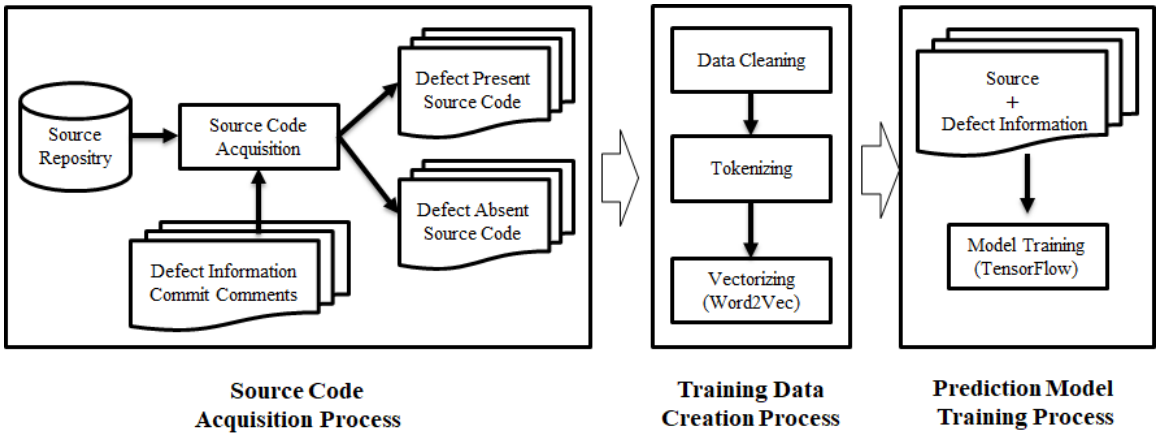


Figure 1: Learning Flow of the Source Code Quality Prediction Method

Classification Model Training Process: Using the vectorized functions and their corresponding labels obtained in the previous step, a binary classification model is built with TensorFlow. This model learns to predict the presence or absence of defects based on the structural features of the functions, serving as a classifier that evaluates the quality of new functions.

The trained classification model outputs the likelihood of defect presence (1/0) when given new source code (functions) as input. By providing a clear binary output, this method eliminates subjective evaluation by developers and offers intuitive, easily understandable metrics. This method is intended to serve as an auxiliary tool for code review and specification verification by developers, with the final quality judgment being made through human review.

4.2. Implementation and Evaluation Environment

To verify the method proposed in the previous section, the Defect Classification System (DCS) was developed. The operating system, programming languages, and existing tools used in the development of DCS are listed in Table 1. In DCS, Word2Vec was first used to learn distributed representations in order to vectorize source code at the function level. For this training, the Noise-Contrastive Estimation (NCE) loss function was used, and optimization was performed using the GradientDescentOptimizer. The key parameter settings for training Word2Vec are shown in Table 2. The resulting vectors were then used to train a binary classification model (referred to as the classification model) that predicts the presence or absence of defects. In this training process, the cross-entropy loss function was used, and RMSPropOptimizer was adopted as the optimization method. These configuration settings were determined at the outset of the study and were consistently applied throughout all evaluations presented in this paper.

Table 1: DCS Software Environment

OS		Ubuntu	16.04 LTS
Programming Language		Python	3.6.1
OSS	Machine Learning Framework	TensorFlow	1.1.0
	Language Processing Tool	Word2Vec	0.5.1

Table 2: Configuration and Operational Parameters of the Word2Vec

Batch Size	500
Embedding Size	300
Vocabulary Size	2,000
Window Size	3
Learning Rate	0.05
Number of Epochs	100,000

4.3. Data Collection and Composition for Training

In machine learning, improving model performance requires large volumes of high-quality training data. In this study, we constructed labeled source code datasets that indicate the presence or absence of defects, using not only OSS but also internally developed code.

To facilitate labeling, a dedicated tool was developed. This tool analyzes commit comments stored in version control systems and automatically extracts commits containing defect-related keywords such as "bug," "Bug," "Fix," "fix," "Fixed," "fixed," "Patch," "patch," "defect," and "Defect." It then obtains the code differences between the relevant revisions to identify the modified functions. The functions before the fix are labeled as “defect present (1),” and the functions after the fix are labeled as “defect absent (0),” which are then registered as training data on a function-by-function basis.

Through this approach, we were able to systematically collect and construct training datasets consisting of approximately 1,000 to 10,000 function-level code samples. The target source code used for evaluation was primarily embedded software, and the programming languages analyzed included C, C++, and C#.

5. Experiments and Evaluation

This section verifies the effectiveness of the proposed source code quality classification method, specifically focusing on the classification model that functions as part of the DCS. The evaluation was conducted from the following perspectives:

- Evaluation of the performance and practicality of the classification model constructed using OSS data
- Evaluation of the applicability of the model to different development environments and types of software
- Performance evaluation of the classification model when applied to software with stable software components

5.1. Evaluation of the Classification Model Built with OSS Data

The evaluation was conducted using source code from OSS projects written in the C language. Multiple OSS projects were selected as targets, based on the ease of obtaining defect-related information and their active and continuously maintained development communities (for details of the projects, refer to Appendix). From each project, a labeled dataset was created by associating the source code with its corresponding defect-related information. The dataset was randomly split into 90% training data and 10% validation data, ensuring that different data were used for training and validation to evaluate generalization performance.

To assess the effect of training data volume on model accuracy, the number of training instances was varied incrementally (from 1,000 to 10,000). As shown in Figure 2, the accuracy improved as the data volume increased, achieving an accuracy of 80.4% when trained with 8,000 samples. This result exceeded the operational target defined in this study. However, after 8,000 samples, the accuracy plateaued, indicating diminishing returns from adding more data.

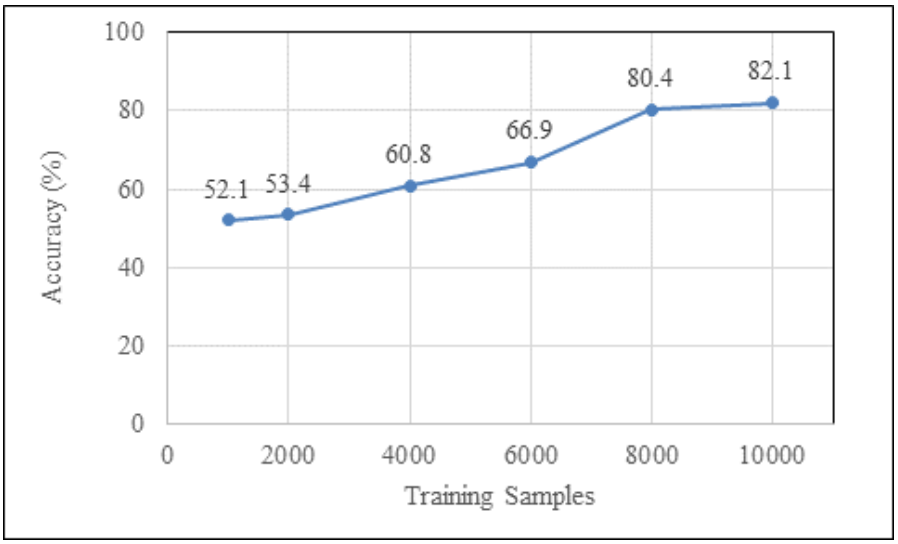


Figure 2: Trends in Accuracy with Varying Training Data Volume

The classification model built from OSS code (trained with 10,000 samples) was applied to software developed at Factory A within the company. All of the source code was written in the C language. When applied to 1,000 validation samples created similarly to the OSS data, the model achieved an accuracy of 61.2%. This relatively low accuracy suggests that the model had limited generalization capability, likely due to structural and functional dissimilarity between the OSS and Factory A's software.

As a comparative experiment, six developers from other departments manually classified a subset (120 samples) of Factory A's validation data. The conditions for the manual classification were as follows:

- Only the source code of the validation data was provided (no specification information)
- All code samples were unfamiliar to the participants
- No time limit was imposed
- External references were prohibited

Table 3 presents the classification results by each developer, and Figure 3 compares their average accuracy with that of the classification model. The average human accuracy was 53.5%, which was lower than the model's accuracy of 61.2%. This result suggests that the classification model demonstrates a certain degree of effectiveness when no specification information is available.

Table 3: Results of Manual Classification by Developers

Participant	Accuracy	Total Response Time	Development Experience	Total Lines of Code Written (Past)
A	55.8%	2h 51min	8 years	5KL
B	50.0%	2h 36min	8 years	2KL
C	48.3%	4h 14min	9 years	3KL
D	55.0%	4h 28min	10 years	5KL
E	57.5%	2h 06min	14 years	100KL
F	54.2%	1h 26min	26 years	200KL

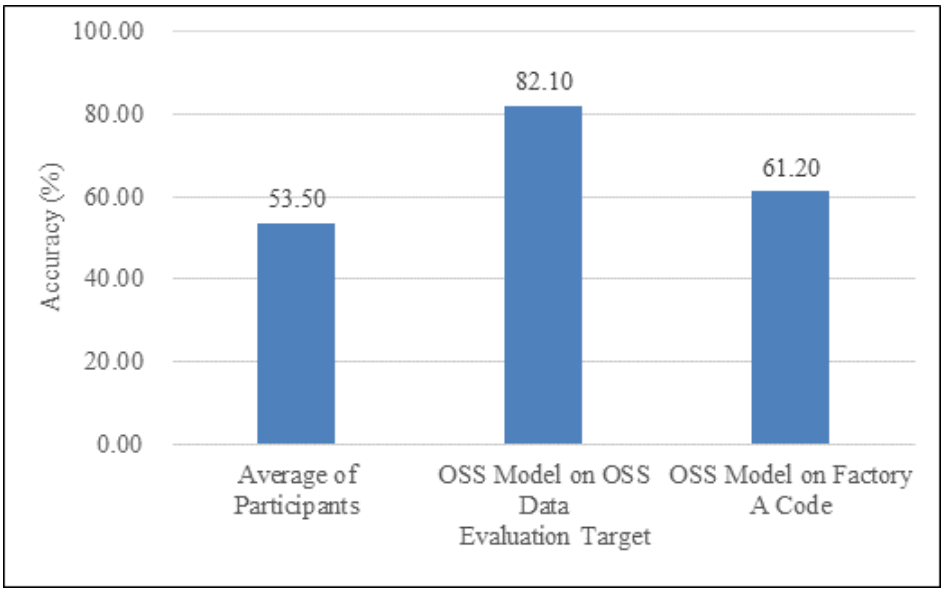


Figure 3: Accuracy Comparison Between Human Participants and the Classification Model

Nevertheless, both results fell significantly short of the target operational accuracy (≥80%) set for internal use. This highlights the limitations of applying OSS-trained models to internal software. Based on this challenge, the following sections present the construction and accuracy evaluation of models trained on internally developed software, aiming for practical deployment.

5.2. Evaluation Using Product Software

5.2.1. Evaluation Based on Development Environments and Software Types

In this section, the classification method was evaluated using software developed at Factory B within the organization. The target software was embedded software developed under consistent processes and quality standards within the factory. Similar to the evaluation using OSS, a classification model was built from source code in the version control system by leveraging bug tracking IDs and commit messages. Based on the findings from the OSS-based evaluation, it was determined that at least 8,000 training samples are necessary to achieve an accuracy above 80%. Therefore, only results for datasets with 8,000 samples or more are reported here. Evaluation results categorized by development environments and software characteristics are presented in Tables 4 to 6.

Factory-Wide Classification Model: The model trained on the entire dataset (328,070 samples) achieved an accuracy of 64.8%, which is comparable to the result observed with Factory A's data. This indicates that building a generalized classification model across all software in the factory may be difficult.

Model by Programming Language: Models built separately for C, C++, and C# all achieved accuracy in the 60% range, suggesting that, as with the factory-wide model, classification based solely on programming language is not highly effective.

Model by Product Type[1]**:** Accuracy varied significantly across products (from 77.9% to 31.2%). For Products 1 through 3, accuracy was already near the target, and further improvement beyond 80% may be achievable through parameter tuning. However, for products such as Product 7, the accuracy was as low as 31.2%, indicating that product-specific models are not universally applicable across all products.

Model by Product Lineage[2]**:** Accuracy hovered around 60%, similar to the results observed for language-based models. This implies that building models at the product lineage level is also challenging.

Table 4: Accuracy by Classification Category

No.	Data Type	Data Size	Accuracy (%)
(1) Overall Factory Result			
1	All Data	328,070	64.8
(2) By Programming Language			
2	C	100,426	64.2
3	C++	196,208	61.6
4	C#	31,436	60.9
(3) By Product Type			
5	Product 1	32,856	77.9
6	Product 2	77,526	77.4
7	Product 3	16,968	77.2
8	Product 4	52,256	61.2
9	Product 5	72,834	59.0
10	Product 6	36,618	58.3
11	Product 7	8,334	31.2
(4) By Product Lineage			
12	Lineage 1	6,326	60.6
13	Lineage 2	99,018	58.0
14	Lineage 3	3,278	56.9

[1] Variants with similar functions (e.g., performance, cost, or regional specs) are treated as the same product type.

[2] Classification based on software characteristics (e.g., control programs, GUI software, etc.).

Table 5: Accuracy by Product Type and Programming Language Combination

No.	Product Type	Language	Data Size	Accuracy (%)
15	Product 5	C++	47,172	84.9
16	Product 1	C	24,744	82.1
17	Product 6	C++	36,504	79.7
18	Product 4	C	40,380	78.5
19	Product 2	C++	70,854	75.0
20	Product 3	C++	16,674	62.6
21	Product 5	C#	23,038	52.2

Table 6: Accuracy by Product Lineage and Programming Language Combination

No.	Lineage	Language	Data Size	Accuracy (%)
22	Lineage 2	C	93,620	78.3
23	Lineage 3	C++	188,446	72.1
24	Lineage 2	C#	8,044	56.6
25	Lineage 1	C	3,496	51.6
26	Lineage 3	C#	23,362	50.7

Model by Combined Attributes: When models were built using combinations of development language and either product type or product lineage, a maximum accuracy of 84.9% was achieved. In particular, the combination of product type and development language appears promising for practical use.

These results indicate that setting an appropriate classification scope is critical to constructing practical classification models. It was observed that rather than using simple criteria such as programming language or product lineage, classification designs that take product-specific characteristics into account have a significant impact on accuracy.

Nonetheless, ensuring sufficient quantity and quality of training data remains a challenge. Compared to OSS-based data, internally developed software often exhibits structural, and quality biases accumulated over years of product evolution, making it more difficult to extract consistent features. Based on the insights obtained in this section, the next section focuses on software from Factory C, where software structures are more stable, to further evaluate the applicability of the proposed method.

5.2.2. Evaluation in the Case of Stable Software Structures

In contrast to the previous section, the applicability of the classification method was evaluated using software developed at Factory C within the organization. The software developed at Factory C is embedded software responsible for equipment control. A key characteristic of this software is its product variety expansion, which is achieved

by standardizing the basic structure and recombining software components. All source code is written in the C programming language, and component composition is varied based on performance, scale, and operating region. This software matches the classification conditions of "programming language" and "product type" suggested as effective in the previous section, making it a suitable target for building the classification model.

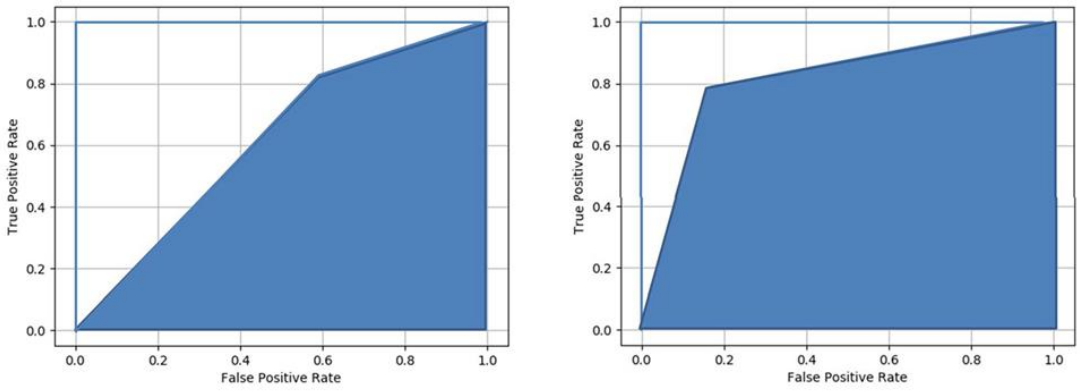
The construction method of the classification model and the dataset preparation process followed the same approach used for Factory B. However, in this evaluation, more detailed analysis was conducted by introducing commonly used performance metrics—such as Precision, Recall, F1 score, and Area Under the Curve (AUC)—in addition to Accuracy. This was deemed necessary to thoroughly analyze the risk of overfitting and the balance of accuracy in the classification model for Factory C 's software.

Table 7 shows the transition of each performance metric depending on the amount of training data. When the number of training samples exceeded 10,000, improvements were observed across all metrics in a well-balanced manner. In particular, the accuracy reached 0.81 and the F1 score reached 0.80, both indicating results at a level feasible for actual operation. However, when the number of training samples increased to 12,000, some metrics (such as Precision) showed a decline.

Figure 4 presents the ROC curves for models trained with 4,000 and 10,000 samples. The ROC curve plots the False Positive Rate (FPR) on the x-axis and the True Positive Rate (TPR) on the y-axis, and the area under the curve (AUC score) is used to evaluate classification performance. For the model trained with 4,000 samples, the AUC score remained at 0.61, with the TPR increasing proportionally with the FPR—reaching a TPR of 0.8 only when the FPR reached 0.6. In contrast, the model trained with 10,000 samples achieved an AUC score of 0.81 and reached a TPR of 0.8 already at an FPR of 0.2, indicating a high true positive rate with fewer false positives. These results suggest that the model trained with 10,000 samples provides more balanced and higher classification performance.

Table 7: Summary of Performance Metrics

Metric / Training Data Size	4,000	6,000	8,000	10,000	12,000
Accuracy score	0.63	0.73	0.75	0.81	0.90
Precision score	0.62	0.75	0.79	0.85	0.75
Recall score	0.61	0.70	0.80	0.77	0.92
F-measure score	0.61	0.72	0.80	0.80	0.83
AUC score	0.61	0.73	0.76	0.81	0.81



(a) 4,000 training samples (b) 10,000 training samples

Figure 4: ROC curves

Figure 5 presents a graph showing the transition of accuracy and error with respect to the number of training epochs. The x-axis represents the number of training epochs, while the y-axis shows either the accuracy or error. The model's accuracy on the training data converged to around 0.8 with increasing training epochs, while the accuracy on validation data stagnated around 0.5. At this point, the generalization error increased, indicating overfitting. On the other hand, when using 10,000 training samples, both accuracy and error demonstrated favorable trends. This suggests that securing a sufficient amount of training data is essential for improving the practical utility of the classification model.

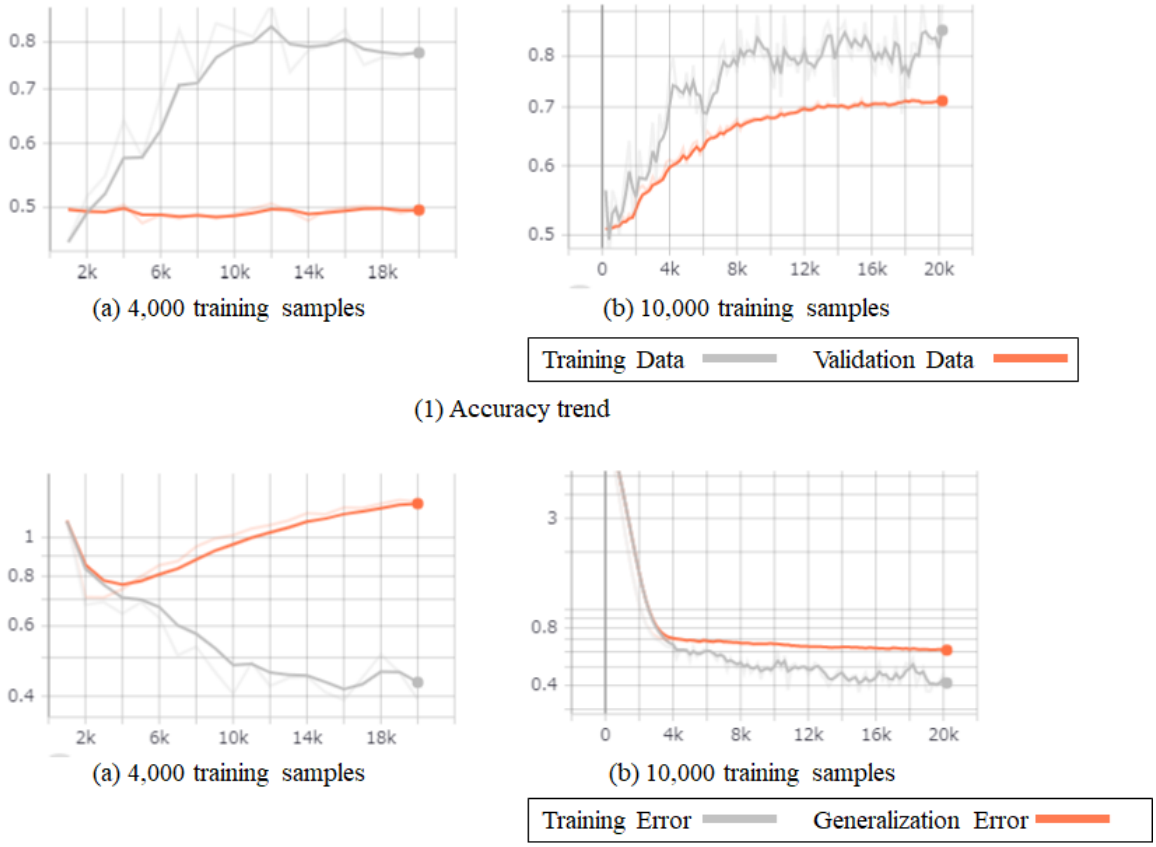


Figure 5: Trends in Accuracy and Error with Training Epochs

In the evaluation conducted at Factory B, performance metrics were not used, as the primary objective was to verify the applicability of the classification model across different contexts. In this section, however, such metrics are introduced as part of an enhanced evaluation methodology, developed based on insights from the previous section and reflecting the progression of this study.

6. Discussion

This study aimed to develop a source code classification model that can operate securely within an organization's local environment, without relying on external resources. The model was evaluated using datasets from both OSS and internally developed software in three factories. Through these evaluations, several insights were gained regarding the key conditions necessary to enhance classification performance in practical settings.

In the model trained on OSS data, increasing the training dataset size led to improved performance, with accuracy reaching 82.1% when 10,000 samples were used.

However, when this model was applied to source code from Factory A, the accuracy dropped to 61.2%. This result indicates that differences in software structure and development processes between the training and target domains can significantly impact the model's generalization capability. These findings underscore the importance of domain alignment as a critical factor in ensuring classification effectiveness.

Evaluations using source code from internal factories revealed further insights. In Factory B, training the model with only a single classification condition yielded limited accuracy. In contrast, combining product type and programming language as classification criteria resulted in a substantial improvement, achieving 84.9% accuracy. This suggests that a uniform classification approach may be insufficient, and that tailoring the model to the specific characteristics of the target software is more effective for practical deployment.

The evaluation at Factory C partially corroborated the findings from Factory B. The software at Factory C features a standardized and modularized structure and is written exclusively in C, aligning well with the effective classification conditions identified earlier. When trained on more than 10,000 samples, the model demonstrated consistently high performance across multiple metrics: Accuracy (0.81), F1 Score (0.80), and AUC (0.81). In contrast, training with only 4,000 samples led to high performance on the training set but poor generalization to the validation set, indicating a tendency toward overfitting.

Based on these results, we conclude that constructing a practical classification model for internal software requires: (1) designing classification schemes that appropriately narrow the target scope, and (2) securing a sufficient volume of labeled training data—at least 10,000 instances. Furthermore, classification model performance using multiple metrics—such as Precision, Recall, F1 Score, and AUC alongside Accuracy—provides a more comprehensive and robust assessment.

Despite these promising results, the study faces the following threats to validity:

Internal Validity: The classification models constructed in this study require more than 10,000 labeled source code samples for training. Labels were automatically assigned using commit comments and bug tracking IDs; however, concerns remain regarding the completeness of defect-related information, the accuracy of mapping these labels to the corresponding code, and the long-term feasibility of constructing large-scale, high-quality labeled datasets.

External Validity: This study focused on specific factories and products within a single company. Therefore, it is unclear whether the findings can be directly applied to other companies or environments with different development styles. Differences in

development structures, coding conventions, and version control practices may influence model performance.

Construct Validity: The dataset used was limited to OSS and a subset of internal projects, which may not fully reflect the variability in project scale or quality. Furthermore, the performance evaluation employed only limited cross-validation and statistical significance testing, requiring cautious interpretation when generalizing the results.

To enhance the reproducibility and generalizability of classification models, future work should refine labeling methods, consider domain alignment in classification schemes, and validate models across various domains.

7. Conclusion and Future Work

This study aimed to construct a source code classification model that can operate locally within organizations under security constraints. To this end, we developed and evaluated classification models using both OSS and internally developed software.

Experimental results showed that the classification model trained on OSS data achieved a maximum accuracy of 82%, demonstrating its potential for practical application. In contrast, models trained on internal datasets exhibited substantial variation in performance depending on factors such as factory, development target, software structure, and development style. Nevertheless, by narrowing the scope of the target systems and ensuring a sufficient volume of training data (approximately 10,000 samples or more), it was possible to build classification models that exceeded the target accuracy of 80% in certain cases.

Moving forward, we plan to improve development processes—such as standardizing defect-fix records—to facilitate the collection of high-quality training data. At the same time, we will explore the adoption of advanced AI techniques capable of delivering high accuracy with smaller datasets. Additionally, through continuous feedback from field applications, we aim to retrain and refine the classification models, ultimately integrating them into code review support and quality assurance activities.

References

1. Mohagheghi, P., & Conradi, R. (2007). Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Software Engineering, 12*(6), 471–516. https://link.springer.com/article/10.1007/s10664-007-9040-x

2. Zimmermann, T., Nagappan, N., Gall, H., Giger, E., & Murphy, B. (2009). Cross-project defect prediction: A large scale experiment on data vs. domain vs. process. In *Proceedings of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering* (pp. 91–100). https://doi.org/10.1145/1595696.1595713

3. Oman, P., & Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *Proceedings of the Conference on Software Maintenance* (pp. 337–344). IEEE. https://www.computer.org/csdl/proceedings-article/icsm/1992/00242525/12OmNyrqzy2

4. McCabe, T. J. (1976). A complexity measure. *IEEE Transactions on Software Engineering, 2*(4), 308–320. https://doi.org/10.1109/TSE.1976.233837

5. Yang, Y., Hu, Q., Zhang, H., Wang, Q., & Li, M. (2022). A survey on deep learning for software engineering. *ACM Computing Surveys (CSUR), 54*(10s), 1–73. https://dl.acm.org/doi/full/10.1145/3505243

6. 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), Article 81. https://doi.org/10.1145/3212695

7. Nagappan, N., Ball, T., & Zeller, A. (2006). Mining metrics to predict component failures. In *Proceedings of the 28th International Conference on Software Engineering* (pp. 452–461). https://doi.org/10.1145/1134285.1134349

8. Khoshgoftaar, T. M., & Allen, E. B. (1998). Predicting the order of fault-prone modules in legacy software. In *Proceedings of the Ninth International Symposium on Software Reliability Engineering (Cat. No. 98TB100257)* (pp. 344–353). IEEE. https://ieeexplore.ieee.org/abstract/document/730899

9. Xing, F., Guo, P., & Lyu, M. R. (2005). A novel method for early software quality prediction based on support vector machine. In *16th IEEE International Symposium on Software Reliability Engineering (ISSRE'05)* (pp. 234–241). IEEE. https://doi.org/10.1109/ISSRE.2005.6

10. Pham, T., Dam, H. K., Ng, S. W., Tran, T., Grundy, J., Ghose, A., Kim, T., & Kim, C. J. (2018). A deep tree-based model for software defect prediction. *arXiv preprint*, arXiv:1802.00921. https://arxiv.org/abs/1802.00921

11. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., & Shou, L. (2020). CodeBERT: A pre-trained model for programming and natural languages. *arXiv preprint*, arXiv:2002.08155. https://arxiv.org/abs/2002.08155

12. Mikolov, T., Chen, K., Corrado, G., & Dean, J. (2013). Efficient estimation of word representations in vector space. *arXiv preprint*, arXiv:1301.3781. https://arxiv.org/abs/1301.3781

13. Rehurek, R. (n.d.). models.word2vec – Word2Vec embeddings – Gensim. Retrieved May 6, 2025, from https://radimrehurek.com/gensim/models/word2vec.html

14. Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., ... & Zheng, X. (2016). TensorFlow: A system for large-scale machine learning. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)* (pp. 265–283). https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi

15. TensorFlow. (n.d.). Retrieved May 6, 2025, from https://www.tensorflow.org

Appendix. List of OSS Repositories Used for Classification Model Construction

No.	Repository URL
1	https://github.com/php/php-src.git
2	https://github.com/torvalds/linux.git
3	https://github.com/FFmpeg/FFmpeg.git
4	https://github.com/grpc/grpc.git
5	https://github.com/antirez/redis.git
6	https://github.com/git/git.git
7	https://github.com/RedisLabsModules/RediSearch.git
8	https://github.com/shadowsocks/shadowsocks-libev.git
9	https://github.com/firehol/netdata.git
10	https://github.com/mpc-hc/mpc-hc.git
11	https://github.com/jp9000/obs-studio.git
12	https://github.com/DrKLO/Telegram.git
13	https://github.com/ggreer/the_silver_searcher.git
14	https://github.com/pjreddie/darknet.git
15	https://github.com/pmq20/ruby-compiler.git
16	https://github.com/Bilibili/ijkplayer.git
17	https://github.com/happyfish100/libfastcommon.git
18	https://github.com/Tencent/wcdb.git
19	https://github.com/wg/wrk.git
20	https://github.com/tmux/tmux.git