Identification of the Relevance of Comments in Codes Using Graph Neural Networks

Durairaj Thenmozhi[1], Aadit P[1], Adithya S[1], Harshil Malisetty[1] and Rohan R[1]

[1]Department of Computer Science and Engineering, SSN College of Engineering, Chennai, India

Abstract

This study presents our submission to the FIRE 2025 IRSE shared task, which focuses on automatically determining whether a comment within source code is genuinely helpful or redundant. Unlike prior approaches that primarily relied on bag-of-words features or large transformer based models, we propose a graph oriented approach in which each code-comment pair is transformed into a graph structure. Here, each node corresponds to individual tokens from the code-comment pairs, edges capture different sorts of relationships between code-comment pair tokens such as normal sequential order of tokens, skip connections for longer range dependencies, and dedicated bridge links that explicitly connect code elements to comment tokens. For said node representation, we use 53-dimensional vectors obtained from pretrained embeddings and we train a compact Graph Neural Network comprising of only 9762 learnable parameters. To further assess the robustness of this approach, we further evaluate it on synthetically constructed datasets, where the trained model attains 62.5% accuracy on a smaller set of 16 samples and 83% on a larger set of 100 samples.

1. Introduction

Comprehending source code is a fundamental prerequisite in modern software development. When engineers are tasked with updating or extending an application, they frequently rely on accompanying comments to understand the purpose of implementations and the rationale behind design decisions. However, the quality and usefulness of such comments in real-world projects can vary considerably depending on multiple factors.

The Forum for Information Retrieval Evaluation (FIRE) 2022 edition introduced a shared task aimed at automatically classifying whether code comments are useful or not useful for given code snippets written in the C programming language. As part of this initiative, a dataset of manually labeled code–comment pairs was built, where domain experts assessed the usefulness of each comment.

Previous research addressing this problem has explored a wide range of computational approaches. Majumdar [2] developed *CommentProbe*, which employs custom embeddings called SWVec, trained on Stack Overflow data, combined with LSTM neural networks, achieving an F1 score of 0.8634 on this task. The FIRE 2022 shared task [1, 6] included submissions comparing traditional bag of words models with transformer based approaches such as BERT, RoBERTa, and ALBERT. Work by Sruthi and Basu [6] demonstrated that, in certain cases, simpler bag of words models with TF-IDF weighting outperformed more complex transformer models.

Most existing studies treat source code purely as natural language text, applying standard natural language processing techniques without accounting for the inherent structural properties of code. A graph based formulation can potentially enable a model to better determine whether a comment is genuinely useful. Graph Neural Networks (GNNs) have demonstrated strong performance across various code analysis tasks but, to the best of our knowledge, have not yet been applied to this specific problem.

This work makes three primary contributions. First, we propose a graph based representation for each code - comment pair that explicitly connects tokens from both sides through multiple types of links. Second, we develop and train a lightweight GNN classifier that utilizes 53-dimensional node

Forum for Information Retrieval Evaluation, December 17-20, 2025, India

adithya2410422@ssn.edu.in (A. S)

features with only 9,762 trainable parameters. Finally, we evaluate the model's generalization capability using synthetically generated datasets of varying sizes.

2. Related Work

2.1. FIRE 2022 Shared Task Results

The FIRE 2022 IRSE shared task attracted several participants who explored diverse computational approaches for assessing the usefulness of code comments. [10, 11, 12] share insights about this task with a different perspective of using metadata for better contexts. Sruthi and Basu [6] submitted five experimental runs that evaluated both classical machine learning and deep learning methods. Their classical models relied on TF-IDF and entropy based feature weighting schemes, combined with classifiers such as Support Vector Machines (SVMs), Random Forests, and Logistic Regression. They also finetuned transformer based architectures, including BERT, RoBERTa, and ALBERT. Their findings revealed that, in some cases, simpler models outperformed more complex ones. The best bag of words approach achieved an F1 score of 0.72 on the training set, whereas ALBERT attained only 0.67. Both approaches exhibited noticeable performance degradation on the test set, with their best submission reaching 53% accuracy.

2.2. Graph Representations and Comment Quality

Graph based representations have gained increasing attention in recent years for a variety of code-analysis tasks. Chen and Monperrus [8] provided a comprehensive survey of embedding techniques for source code, highlighting the use of Abstract Syntax Trees (ASTs), Control Flow Graphs (CFGs), Data Flow Graphs (DFGs), and Program Dependence Graphs (PDGs). Our approach differs in that it constructs custom graphs that directly connect code structures with corresponding natural language comments, thereby enabling the model to learn associations between structural code patterns and comment usefulness.

Research on the quality and consistency of code comments has been active for over a decade. Tan [7] developed *iComment*, a tool that integrates natural language processing, machine learning, and program analysis to detect inconsistencies between code and comments, achieving accuracy rates ranging from 90.8% to 100%. Majumdar [2] introduced *CommentProbe*, a framework designed specifically for comment usefulness classification. Their approach utilized specialized embeddings, termed SWVec, and achieved an F1 score of 0.8634 on C language codebases. More recently, Majumdar [3] explored the use of large language models (LLMs) for code comprehension, investigating how retrieval and reasoning mechanisms can be leveraged to generate meaningful and contextually appropriate comments.

3. Methodology

3.1. Problem Formulation and Dataset

The dataset employed in this study was introduced as part of the FIRE 2022 IRSE shared task [1]. It comprises 11,452 code - comment pairs written in the C programming language, each manually annotated by domain experts as either *useful* or *not useful*. The dataset includes code snippets extracted from open-source repositories such as glibc, ffmpeg and linux. Prior to model development, the data were preprocessed by removing comment markers, standardizing indentation, and converting all tokens to lowercase to ensure uniformity.

3.2. Graph Construction from Code - Comment Pairs

Each code - comment pair is represented as an undirected, weighted graph using the NetworkX library. The graph construction process consists of three primary stages: tokenization, node creation, and edge formation.

- **Tokenization:** For the source code, text is segmented into tokens using the regular expression `[+[]\w]+`, which retains words, operators, and punctuation as separate entities. All tokens are converted to lowercase. For comments, comment markers are removed, and alphabetic words are extracted using `[a-zA-Z][a-zA-Z0-9]*`. Tokens consisting of a single character are discarded to reduce noise.
- **Node Creation:** Each token is represented as a node that stores its textual content, source (code or comment), and positional index within the sequence. Code tokens are positioned before comment tokens to preserve a logical structure.
- **Edge Formation:** Multiple types of edges are added to capture both local sequential dependencies and long-range relationships. Sequential edges with a weight of 1.0 connect adjacent tokens within the same sequence. Additional edges include skip links for non-adjacent dependencies, bridge edges connecting related code and comment tokens, and self-loops for all nodes to retain self-referential information.

3.3. Node Feature Extraction

Each node is encoded as a 53-dimensional feature vector. These embeddings are pretrained on the training corpus and capture token identity, contextual information from neighboring tokens, origin type (code or comment), and normalized positional attributes. The embeddings are saved and reused across all subsequent experiments to maintain consistency and reduce computational overhead.

3.4. Graph Neural Network Architecture

The proposed model builds upon the Graph Convolutional Network (GCN) framework introduced by Kipf and Welling [9]. The first GCN layer transforms the 53-dimensional token embeddings into 64-dimensional feature representations using ReLU activation, while the second layer preserves the same dimensionality. Following node level updates, a global mean pooling operation aggregates the node embeddings into a single graph level representation. Dropout with a rate of 0.3 is applied to mitigate overfitting.

The pooled representation is then passed through a fully connected layer that reduces its dimensionality from 64 to 32, followed by a final linear layer that outputs two logits corresponding to the binary classes. Overall, the model comprises 9,762 trainable parameters, emphasizing its lightweight design. The GCN layer updates each node representation according to the following formulation:

$$h_i^{(l+1)} = \sigma\left(\sum_{j \in \mathcal{N}(i) \cup \{i\}} \frac{1}{\sqrt{d_i d_j}} W^{(l)} h_j^{(l)}\right),$$

where $h_i^{(l)}$ denotes the feature vector of node i at layer l, $\mathcal{N}(i)$ represents the set of neighboring nodes, d_i is the degree of node i, $W^{(l)}$ is the learnable weight matrix, and σ is the ReLU activation function.

3.5. Training Configuration and Class Balancing

The dataset was divided into 80% for training (9,161 samples) and 20% for testing (2,291 samples). The training subset exhibited a class imbalance, with 3,510 *not useful* samples and 5,651 *useful* samples. To address this issue, an upsampling strategy was adopted by generating synthetic variants of the *not useful* samples. Gaussian noise with a standard deviation of $\sigma = 0.05$ was added to the 53-dimensional embeddings, producing a balanced training set with 5,651 examples per class.

Model training was performed using the Adam optimizer with a learning rate of 0.001 and a weight decay of 10^{-4}. A batch size of 64 was used, and training proceeded for 30 epochs. The binary cross-entropy loss function was employed as the objective. At the end of each epoch, the model checkpoint yielding the highest validation accuracy was preserved for evaluation.

3.6. Synthetic Data Generation

To evaluate the model's generalization capability beyond the original training distribution, two synthetic test sets were constructed. The smaller set contained 16 examples evenly divided between *useful* and *not useful* classes, while the larger set consisted of 100 similarly balanced examples. The *useful* comments described non trivial programming concepts such as binary search algorithms or memory management, whereas the *not useful* comments corresponded to trivial statements such as "set x to 1" or "loop through array." Each synthetic code - comment pair was processed using the same graph construction pipeline as the original dataset to ensure methodological consistency.

4. Experimental Results

4.1. Performance on Original Test Set

The proposed model achieved an accuracy of 80.8% on the original test set comprising 2,291 samples. The weighted F1 score was 0.807, indicating well-balanced performance across both classes. For the *not useful* class, precision, recall, and F1 scores were 0.77, 0.71, and 0.74, respectively. For the *useful* class, precision was 0.83, recall was 0.87, and F1 was 0.85.

The confusion matrix revealed that, among 879 *not useful* samples, 626 (71%) were correctly classified, while 253 were misclassified as *useful*. Conversely, for 1,412 *useful* samples, 1,226 (87%) were correctly identified, and 186 were misclassified as *not useful*. Table 1 presents a comparison of model performance under different experimental configurations. The baseline configuration, which utilized 6-dimensional features, achieved 79.8% accuracy. Increasing the feature dimensionality to 53 improved accuracy to 80.4%, and incorporating upsampling further enhanced it to 80.8%.

Table 1
Performance comparison across different test sets and methods.

Method / Test Set	Samples	Accuracy	F1 Score
Baseline (6D features)	2,291	79.6%	0.793
Enhanced (53D features)	2,291	80.4%	0.801
Enhanced + Upsampling	2,291	80.8%	0.807
Small Synthetic Test	16	62.5%	0.600
Large Synthetic Test	100	83.0%	0.825

These results highlight that the proposed GNN-based approach achieves competitive performance compared to traditional and transformer-based models, while maintaining a compact architecture with significantly fewer parameters.

4.2. Performance on Synthetic Test Sets

When evaluated on the small synthetic dataset containing 16 samples, the model attained an accuracy of 62.5%, corresponding to an 18.3% decrease relative to the original test set. For the larger synthetic dataset of 100 samples, accuracy increased to 83.0%. These findings suggest that model performance improves with larger and more diverse synthetic datasets, indicating sensitivity to dataset size and variability.

4.3. Training Dynamics

During training, the model demonstrated stable convergence behavior. The training loss decreased from 0.607 in the first epoch to 0.421 by the 30th epoch, while validation accuracy exhibited steady improvement during the initial 15–20 epochs before plateauing around epochs 25–26. No significant overfitting was observed, as validation accuracy remained consistent and did not deteriorate toward

the end of training. These trends indicate that the model effectively generalized to unseen data without sacrificing stability.

5. Discussion

5.1. Advantages of Graph-Based Representation

The proposed graph-based representation offers distinct advantages over treating code and comments purely as plain text. By incorporating multiple types of connections, the graph structure enables the model to better capture relationships between code tokens and corresponding comment tokens. Sequential edges preserve the inherent order of tokens, while skip connections allow the model to recognize long-range dependencies across non-adjacent tokens. Bridge connections, on the other hand, explicitly link code and comment elements, allowing the model to associate linguistic context with program structure. Collectively, these connections help the model discern whether a comment provides meaningful information or simply restates the code.

5.2. Limitations and Future Work

The proposed approach has certain limitations. First, the graph construction process relies on manually defined rules, rather than being learned adaptively from data. Second, the 53-dimensional embeddings, though effective, may not fully capture the semantic richness of complex programming constructs or natural-language comments. Incorporating larger pre-trained representations such as CodeBERT or GraphCodeBERT could provide more expressive features.

Future extensions of this work may focus on: (1) learning optimal graph structures and edge weights directly from data, (2) integrating attention-based mechanisms for more effective information propagation, and (3) employing large-scale pre-trained models trained on diverse code repositories to enhance generalization across programming languages and comment styles.

5.3. Comparison with Related Work

On the FIRE 2022 IRSE dataset, the proposed model achieved an accuracy of 80.8%, which is comparable to, though slightly below, the performance of *CommentProbe* [2], which reported an F1 score of 0.863. This difference can be attributed to the specialized embeddings used by *CommentProbe*, which were trained on extensive Stack Overflow data. When compared to the submissions by Sruthi and Basu [6], our model demonstrates superior performance, achieving an F1 score of 0.807 compared to 0.72 for the best bag-of-words model and 0.67 for transformer-based approaches. Furthermore, our model maintains a substantially smaller parameter count than transformer-based models, underscoring its efficiency and suitability for resource-constrained environments without compromising accuracy.

6. Conclusion

This paper presented a graph-based framework for classifying the usefulness of code comments using Graph Neural Networks (GNNs). The proposed model achieved an accuracy of 80.8% on the FIRE 2022 IRSE shared task, while remaining lightweight with only 9,762 trainable parameters. Evaluation on synthetic datasets demonstrated that accuracy decreased to 62.5% on a smaller set of 16 samples, but improved to 83.0% on a larger set of 100 samples. These findings suggest that the proposed graph-based approach effectively captures structural relationships between code and comments, striking a balance between the simplicity of bag-of-words models and the representational power of transformer-based architectures.

The results underscore the potential of graph-oriented models as efficient alternatives for comment usefulness classification, especially in scenarios where computational resources are limited. Future

research may explore scaling this framework with larger pre-trained embeddings and automated graph construction techniques to further enhance generalization.

All source code and experimental configurations are publicly available at: https://github.com/CrimsonCoderAadit/code_comment_project_clean

Acknowledgments

We thank the organizers of FIRE 2025 IRSE shared task for providing the dataset and evaluation framework. The authors would also like to express their gratitude to the faculty of the Department of Computer Science and Engineering, SSN College of Engineering, for their valuable guidance and support throughout this work.

Declaration on Generative AI

In the course of preparing this manuscript, the author(s) employed the generative AI tool ChatGPT. Its use was limited to performing checks for grammar and spelling. Following this, the author(s) conducted a thorough review and revision of the text and assume full responsibility for the final published content.

References

[1] S. Majumdar et al., "Overview of the IRSE track at FIRE 2022: Information Retrieval in Software Engineering," ACM, 2022.

[2] S. Majumdar et al., "Automated evaluation of comments to aid software maintenance," Journal of Software: Evolution and Process, vol. 34, 2022.

[3] S. Majumdar et al., "Comprehending C Codes with LLMs," Pattern Recognition Letters, Elsevier, 2025.

[4] S. Paul et al., "Efficiency of Large Language Models to scale up Ground Truth: Overview of the IRSE Track at Forum for Information Retrieval 2023," 2023.

[5] S. Paul et al., "Overview of the 'Information Retrieval in Software Engineering' (IRSE) track at Forum for Information Retrieval 2024," 2024.

[6] S. S, T. Basu, "Identification of the Relevance of Comments in Codes Using Bag of Words and Transformer Based Models," FIRE 2022 Working Notes, 2022.

[7] L. Tan et al., "/*icomment: Bugs or bad comments?*/," SIGOPS Oper. Syst. Rev., vol. 41, pp. 145–158, 2007.

[8] Z. Chen, M. Monperrus, "A literature study of embeddings on source code," arXiv:1904.03061, 2019.

[9] T. N. Kipf, M. Welling, "Semi-supervised classification with graph convolutional networks," ICLR, 2017.

[10] A. Deshpande, A. Maji, D. Mondol, P. P. Das, P. D. Clough, and S. Majumdar, "The Code–LLM Handshake: Smarter Maintenance Through AI," in *Proceedings of the 17th Annual Meeting of the Forum for Information Retrieval Evaluation*, pp. 9–12, 2025.

[11] A. Mitra, S. Majumdar, A. Mukhopadhyay, P. P. Das, P. D. Clough, and P. P. Chakrabarti, "Operationalizing Large Language Models with Design-Aware Contexts for Code Comment Generation," *arXiv preprint* arXiv:2510.22338, 2025.

[12] S. Majumdar and P. P. Das, "Smart Knowledge Transfer using Google-like Search," *arXiv preprint* arXiv:2308.06653, 2023.