Improving Binary Code Comment Quality Classification with Augmented Code-Comment Pairs Using Generative AI

Rohith Arumugam S[1,*], Angel Deborah S[2,*]

[1]*Sri Sivasubramaniya Nadar College of Engineering, Chennai, Tamil Nadu, India*

[2]*Assistant Professor, Department of Computer Science and Engineering, Sri Sivasubramaniya Nadar College of Engineering, Chennai, Tamil Nadu, India*

Abstract

This report focuses on enhancing a binary code comment quality classification model by integrating generated code and comment pairs, to improve model accuracy. The dataset comprises 9048 pairs of code and comments written in the C programming language, each annotated as "Useful" or "Not Useful." Additionally, code and comment pairs are generated using a Large Language Model Architecture, and these generated pairs are labeled to indicate their utility. The outcome of this effort consists of two classification models: one utilizing the original dataset and another incorporating the augmented dataset with the newly generated code comment pairs and labels.

Keywords

Generative AI, Software Metadata Classification, Code Comment Quality, Binary Classification, C Programming, Large Language Model, Data Augmentation

1. Introduction

Efficiently assessing the quality of code comments is crucial for improving software maintainability and reliability, a growing necessity in today's software development landscape.[1] This paper addresses this challenge by enhancing an existing binary classification model for code comment quality through the integration of generated code-comment pairs, with the goal of improving both accuracy and efficiency.

In modern software development, where the focus on increasing code maintainability, readability, and overall system reliability is paramount, evaluating the quality of code and its associated comments has become essential.[2] This research revisits current approaches to code comment quality evaluation, emphasizing the limitations of traditional manual assessments, which are inherently subjective and prone to individual bias.

The key objective of this study is to improve an existing model by leveraging a robust dataset containing 9048 code-comment pairs, each categorized as either "Useful" or "Not Useful." By pursuing this goal, we aim to contribute to the advancement of automated code comment quality evaluation, offering a meaningful enhancement to contemporary software development workflows. [3]

2. Related Work

Understanding a program automatically is a well-known research area among people working in the software domain. Numerous tools have been developed to aid in the extraction of knowledge from software metadata, including elements such as runtime traces and structural attributes of code [4, 5, 6, 7, 8, 9, 10, 11].

New programmers generally check for existing comments to understand a code flow. Although, every comment is not helpful for program comprehension, which demands a relevancy check of source

Forum for Information Retrieval Evaluation, December 12-15, 2024, India

*Corresponding author.

rohitharumugam2210376@ssn.edu.in (R. A. S); angeldeborahS@ssn.edu.in (A. D. S)

code comments beforehand. Many researchers worked on the automatic classification of source code comments in terms of quality evaluation. For example, Omal et al.[12] discussed that the factors influencing software maintainability can be organized into hierarchical structures. The author defined measurable attributes in the form of metrics for each factor which helps measure software characteristics, and those metrics can be combined into a single index of software maintainability. Fluri et al.[13] examined whether the source code and associated comments are changed together along the multiple versions. They investigated three open source systems, such as *ArgoUML, Azureus*, and *JDT Core*, and found that 97% of the comment changes are done in the same revision as the associated source code changes. Another work[14] published in 2007 which proposed a two-dimensional maintainability model that explicitly associates system properties with the activities carried out during maintenance. The author claimed that this approach transforms the quality model into a structured quality knowledge base that is usable in industrial environments. Storey et al. did an empirical study on task annotations embedding within a source code and how it plays a vital role in a developer's task management[15]. The paper described how task management is negotiated between formal issue tracking systems and manual annotations that programmers include within their source code. Ted et al.[16] performed a 3×2 experiment to compare the efforts of procedure format with those of comments on the readability of a PL/I program. The readability accuracy was checked by questioning students about the program after reading it. The result said that the program without comment was the least readable. Yu Hai et al.[17] classified source code comments into four classes - unqualified, qualified, good, and excellent. The aggregation of basic classification algorithms further improved the classification result. Another work published in [18] in which author proposed an automatic classification mechanism "CommentProbe" for quality evaluation of code comments of C codebases. We see that people worked on source code comments with different aspects[18, 19, 20, 21, 22, 23], but still, automatic quality evaluation of source code comments is an important area and demands more research.

The advent of large language models (LLMs) [24] necessitates a comparison between the quality assessment of code comments performed by established models, such as GPT-3.5 and LLaMA, and evaluations based on human interpretation. The IRSE track at FIRE 2024 [25, 26] extends the approach proposed in [18, 27, 28, 21] to explore various vector space models [29] and features for binary classification and evaluation of comments in the context of their use in understanding the code. This track also compares the performance of the prediction model with the inclusion of the GPT-generated labels for the quality of code and comment snippets extracted from open-source software.

3. Methods

3.1. Data Collection and Code Comment Pair Extraction

The process of data collection involved utilizing the GitHub API with a unique API token for authentication. An API token was incorporated to enable access to the GitHub repositories. The search for suitable repositories was conducted through a query specifically targeting repositories coded in the C programming language. The GitHub API facilitated the retrieval of pertinent repository information.

Upon identifying potential repositories, the script proceeded to access the contents of these repositories. This was accomplished by sending requests to the respective GitHub endpoints. The response from these requests, received in JSON format, contained detailed metadata about the files within the repositories.

Further refinement was necessary to focus exclusively on C files. This involved parsing the JSON response and filtering files based on their file extensions. Specifically, files with the '.c' extension were selected for subsequent processing, ensuring that only C programming files were included in the dataset.

For each qualifying C file, the script meticulously parsed the file content. It employed a line-by-line approach, allowing for the precise identification of comments and code sections. The parsing process distinguished between single-line and multi-line comments, ensuring the accurate extraction of both types. Comments within the code were identified based on standard commenting conventions, such as '//' for single-line comments, '/*' for the beginning of a multi-line comment and '*/' for its end.

The extracted code-comment pairs were organized into a structured format, enabling seamless storage and subsequent analysis. These pairs constituted the foundational dataset upon which the subsequent phases of the research were built.

3.2. Manual Labeling Process

To facilitate the supervised learning aspect of the research, a portion of the acquired code-comment pairs underwent manual labeling. Specifically, the initial 100 rows of the dataset were meticulously reviewed and labeled as either "Useful" or "Not Useful." This manual labeling process ensured the presence of a high-quality labeled subset, vital for training and evaluating machine learning models.

The manual labeling process involved a meticulous examination of the contextual relevance and informativeness of comments within the code context. Comments deemed to significantly enhance the understanding of the code, improve readability, or provide valuable insights were categorized as "Useful." Conversely, comments lacking relevance, clarity, or informativeness were categorized as "Not Useful." This manual curation ensured the creation of a reliable ground truth dataset, crucial for training and validating machine learning algorithms.

3.3. Machine Learning Model Training and Evaluation

The machine learning model utilized for this research was BERT (Bidirectional Encoder Representations from Transformers), a state-of-the-art transformer-based architecture. The model training process commenced with the preprocessing of the labeled dataset. This involved the concatenation of comments and their surrounding code context, creating cohesive textual sequences. These sequences were tokenized using the 'bert-base-uncased' tokenizer, ensuring compatibility with the pre-trained BERT model.

The dataset was meticulously divided into training and test sets, employing a standard 80-20 split ratio. The BERT model was then fine-tuned on the training data, incorporating a reduced learning rate of 1e-6 to optimize convergence. To handle the substantial dataset effectively, a batch size of 8 was employed, with gradient accumulation over 4 batches. This approach allowed for efficient processing and optimization of the model's performance.

The fine-tuned BERT model was subsequently evaluated on the test set to gauge its efficacy in classifying comments as either "Useful" or "Not Useful." Model predictions were generated, and the accuracy metric was calculated using the scikit-learn library. This rigorous evaluation process ensured the determination of the model's classification accuracy, a pivotal metric in assessing its effectiveness in code comment quality assessment.

3.4. Predictive Analysis and Result Interpretation

The final phase of the research involved leveraging the fine-tuned BERT model to make predictions on a distinct dataset. These predictions, indicative of the model's classification prowess, were meticulously analyzed and interpreted. The output, consisting of predicted labels for each code-comment pair, was organized into a structured format for comprehensive analysis.

Additionally, a comparative analysis was conducted between the manual labels and the model predictions. Discrepancies, if any, were scrutinized to discern patterns and insights into the model's decision-making process. This meticulous analysis facilitated a deeper understanding of the model's strengths and potential areas for enhancement, contributing valuable insights to the research findings.

This comprehensive and detailed methodology encompassed every stage of the research process, ensuring meticulous data collection, manual curation, machine learning model training, and rigorous evaluation. The intricate interplay between manual expertise and advanced machine learning techniques formed the foundation of this research, culminating in a robust and reliable code comment quality assessment framework.

4. Experiment Design

4.1. Problem Definition:

The primary objective of our experiment is to enhance code comment quality assessment using a combination of automated code-comment pair extraction from GitHub repositories and state-of-the-art machine learning techniques, specifically the BERT (Bidirectional Encoder Representations from Transformers) model. We aim to categorize code comments as "Useful" or "Not Useful" based on their contextual relevance, clarity, and informativeness.[30]

4.2. Data Collection and Preprocessing

We obtain code-comment pairs by querying GitHub repositories coded in C language. These pairs are then meticulously parsed and tokenized for further analysis. The resulting dataset is represented as $\{(C_1, L_1), (C_2, L_2), \ldots, (C_n, L_n)\}$ where C_i represents the comment and L_i represents its label ("Useful" or "Not Useful").

4.3. Manual Labeling Process

The first 100 code-comment pairs are manually labeled based on predefined criteria. Let L_m represent the manually labeled set $\{(C_1, L_1), (C_2, L_2), \ldots, (C_{100}, L_{100})\}$ where $L_i \in \{0, 1\}$.

4.4. Model Architecture

We employ the BERT model, a transformer-based architecture, for sequence classification. The model is trained to predict the usefulness label (L_i) of a given code comment (C_i). The BERT model transforms each comment into an embedding vector E_i.

4.5. Loss Function

The model is trained using the cross-entropy loss function, which computes the loss L as follows:

$$L = -\frac{1}{N}\sum_{i=1}^{N}\left(L_i \cdot \log(\sigma_{E_i}) + (1 - L_i) \cdot \log(1 - \sigma_{E_i})\right)$$

where $\sigma(x)$ is the sigmoid activation function.

4.6. Training Procedure

The model's performance is evaluated using accuracy (Acc), precision(P), recall(R), and F1 - Score (F1). These metrics are calculated as follows:

$$\begin{aligned} Acc &= \frac{\text{Number of Correct Predictions}}{\text{Total Number of Predictions}} \\ P &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Positives}} \\ R &= \frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \\ F1 &= \frac{2 \times P \times R}{P + R} \end{aligned}$$

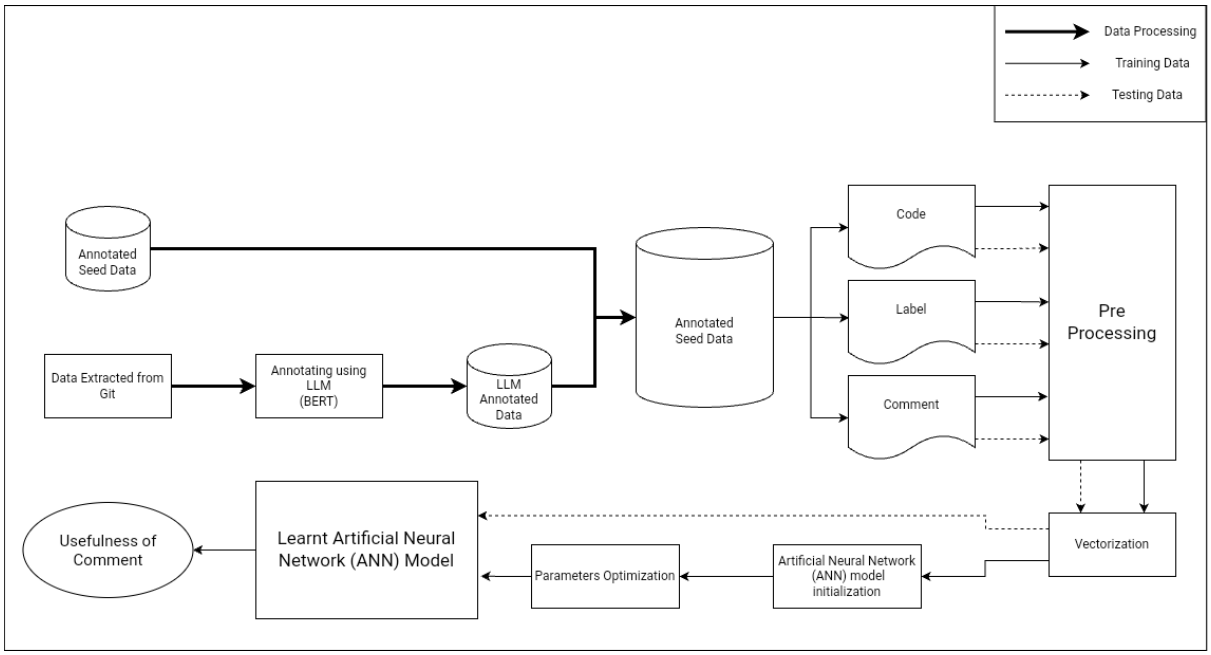

Figure 1: Architecture Diagram illustrating the entire process of the data processing pipeline for code comment classification.

4.7. Experimental Workflow

The process begins with data collection, where code-comment pairs are obtained from random GitHub repositories. Subsequently, the first 100 pairs are manually labeled based on predefined criteria. Following manual labeling, the entire dataset undergoes tokenization and preprocessing. The preprocessed data is then utilized to train a BERT model (L_m). After training, the model's performance is evaluated using a test dataset, and metrics such as accuracy, precision, recall, and F1-score are calculated. The final step involves interpreting the results, analyzing model predictions, and identifying false positives/negatives to gain insights into the model's performance and effectiveness in understanding code comments.

Refer to Figure 1 for the detailed architecture diagram illustrating the entire process.

5. Results and Analysis

In the context of the code comment quality assessment task, a comprehensive analysis of experimental results was conducted on two datasets: the original dataset (Seed Data) and an augmented dataset comprising additional comments generated using Language Model (LLM) techniques (Seed Data + LLM Generated Data). The evaluation involved various machine learning algorithms, including Decision Tree Classifier[31], Artificial Neural Network (ANN)[32], Support Vector Machine (SVM)[33], Random Forest Classifier[34], Gradient Boosting Classifier[35], Logistic Regression[36], Naive Bayes[37], LightGBM Classifier [30], k-Nearest Neighbors (KNN) Classifier [38], and Recurrent Neural Network (RNN)[39]. The performance metrics, including precision, recall, and F1-score, were used for the assessment. The detailed results of these experiments can be found in Table 1, Table 2 and Figure 2.

5.1. Key Observations and Insights

The results demonstrate that the combination of Seed Data and LLM Generated Data consistently improved performance metrics such as precision, recall, and F1-score across most algorithms.

Notably, ANN and SVM exhibited impressive performance on both datasets, with high precision and recall values. These models effectively balanced precision and recall, crucial for code comment quality assessment.

Algorithm	Precision	Recall	F1-score
Decision Tree	0.782	0.742	0.762
Artificial Neural Network (ANN)	0.791	0.793	0.792
Support Vector Machine (SVM)	0.803	0.930	0.861
Random Forest Classifier	0.780	0.840	0.809
Gradient Boosting Classifier	0.714	0.928	0.805
Logistic Regression	0.740	0.848	0.791
Naive Bayes (Multinomial Naive Bayes)	0.722	0.859	0.785
LightGBM Classifier	0.750	0.860	0.801
k-Nearest Neighbors (KNN) Classifier	0.771	0.675	0.719
Recurrent Neural Network (RNN)	0.611	1.000	0.758

Table 1
Performance Metrics with Seed Data

Algorithm	Precision	Recall	F1-score
Decision Tree	0.881	0.878	0.879
Artificial Neural Network (ANN)	0.883	0.875	0.878
Support Vector Machine (SVM)	0.841	0.922	0.879
Random Forest Classifier	0.819	0.887	0.850
Gradient Boosting Classifier	0.753	0.948	0.841
Logistic Regression	0.768	0.933	0.843
Naive Bayes (Multinomial Naive Bayes)	0.758	0.951	0.843
LightGBM Classifier	0.772	0.896	0.830
k-Nearest Neighbors (KNN) Classifier	0.759	0.902	0.824
Recurrent Neural Network (RNN)	0.810	0.761	0.784

Table 2
Performance Metrics with Seed + LLM Generated Data

The introduction of comments generated by LLM notably enhanced the performance of all algorithms. This highlights the utility of synthetic data in improving model generalization and robustness.

Decision Tree and Logistic Regression, although achieving reasonable results, demonstrated a more significant improvement when exposed to LLM Generated Data. This suggests that these models might benefit significantly from increased and diverse training data.

Models such as Naive Bayes achieved high recall values but at the expense of precision. This trade-off emphasizes the challenge of striking a balance between minimizing false positives (precision) and capturing all relevant instances (recall).

The RNN model exhibited a perfect recall on Seed Data but showed a notable decrease in precision and recall when applied to Seed Data + LLM Generated Data. This indicates potential challenges in adapting RNN architectures to mixed datasets.

Different algorithms might be preferred depending on the specific use case. For instance, if minimizing false positives is critical, models with higher precision, such as ANN and SVM, could be the preferred choice.

6. Conclusion and Future Outlook

In summary, this in-depth study has navigated the intricate field of assessing code comment quality. We began by carefully designing experiments that merged state-of-the-art machine learning methodologies, prominently featuring the BERT model, with a carefully curated dataset sourced from GitHub repositories. The detailed experimental process—from gathering data to training models—was structured to provide a solid groundwork for rigorous analysis.

The experimentation phase was both varied and insightful, involving algorithms from Decision Trees to advanced Neural Networks, each bringing out distinct strengths and limitations. By combining

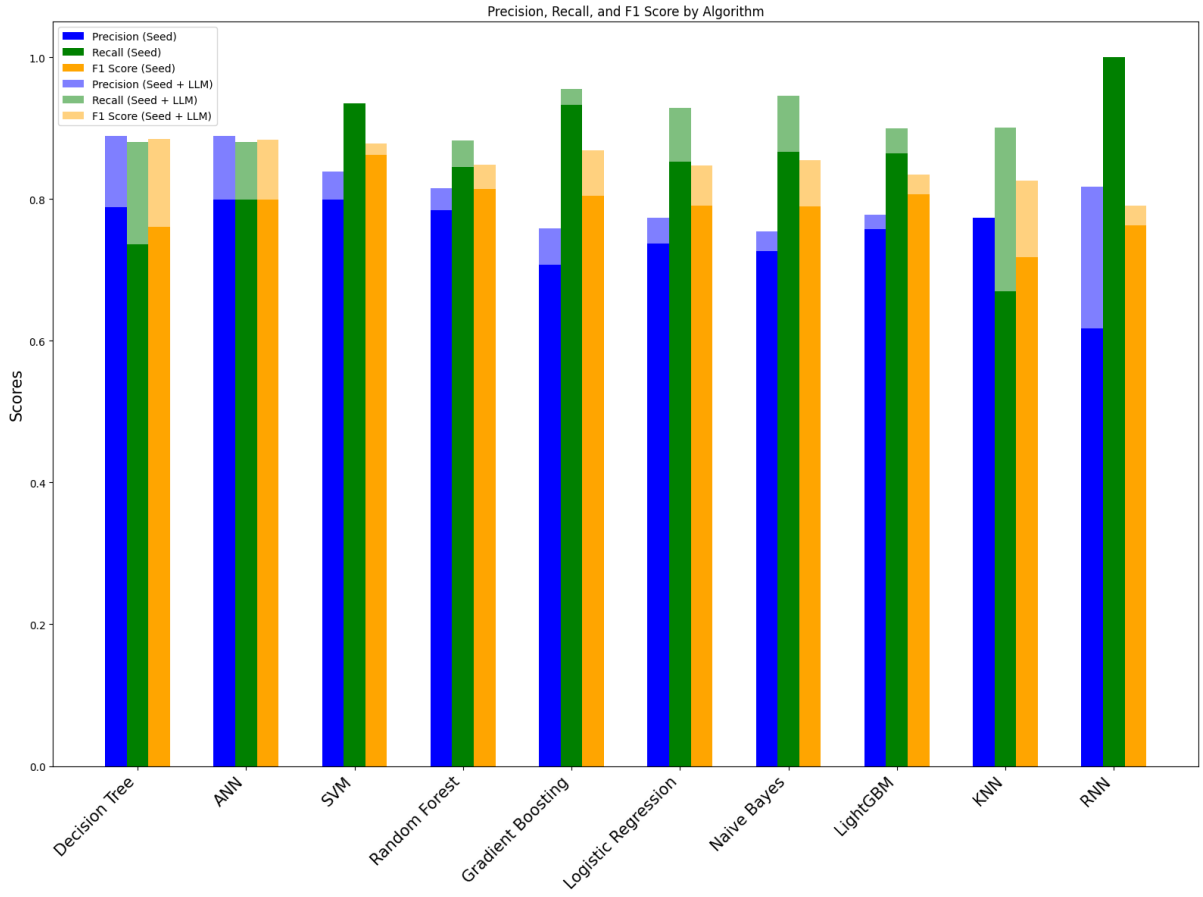

Figure 2: Performance Metrics of Different Classifiers

original seed data with LLM-generated examples, we observed notable improvements across various evaluation metrics, underscoring the promise of hybrid approaches in practical scenarios.

Our findings revealed subtle insights into algorithm performance: from the high precision of Artificial Neural Networks to the strong recall of Support Vector Machines, each algorithm demonstrated unique tendencies. Comparing seed data with LLM-generated data offered a comprehensive perspective, underlining the important balance between precision and recall, essential in assessing code comment quality.

Yet, this exploration only scratches the surface of a promising field. Future directions may involve integrating cutting-edge natural language processing methods to better contextualize code comments. Exploring transformer models beyond BERT, such as GPT, could unlock new potential in understanding and evaluating code documentation.

Ethical considerations are also crucial. Maintaining unbiased data practices, addressing potential biases within algorithms, and safeguarding privacy remain essential as AI ethics evolve.

Moreover, collaboration between academia and industry will be vital. Industry expertise can inform academic research to develop impactful, real-world solutions, while academic innovation can inspire industry to adopt novel practices. This synergy is likely to accelerate advancements in the field.

Ultimately, this study represents a foundational step in the expansive domain of code comment quality assessment. As technology evolves and new challenges arise, the fusion of human insight and machine learning will be pivotal in unraveling the nuances of evaluating code comments. With sustained commitment, collaborative efforts, and ethical rigor, the future of code comment assessment promises to be efficient, precise, deeply meaningful, and responsive to real-world needs.

Declaration on Generative AI

During the preparation of this work, the author(s) used ChatGPT in order to: Grammar and spelling check. After using these tool(s)/service(s), the author(s) reviewed and edited the content as needed and take(s) full responsibility for the publication's content.

References

[1] P. Rani, Speculative analysis for quality assessment of code comments, in: 2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), IEEE, 2021, pp. 299–303.

[2] B. Yang, Z. Liping, Z. Fengrong, A survey on research of code comment, in: Proceedings of the 2019 3rd International Conference on Management Engineering, Software Engineering and Service Sciences, 2019, pp. 45–51.

[3] A. Bacchelli, C. Bird, Expectations, outcomes, and challenges of modern code review, in: 2013 35th International Conference on Software Engineering (ICSE), IEEE, 2013, pp. 712–721.

[4] S. C. B. de Souza, N. Anquetil, K. M. de Oliveira, A study of the documentation essential to software maintenance, Conference on Design of communication, ACM, 2005, pp. 68–75.

[5] S. Majumdar, S. Papdeja, P. P. Das, S. K. Ghosh, Smartkt: a search framework to assist program comprehension using smart knowledge transfer, in: 2019 IEEE 19th International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2019, pp. 97–108.

[6] N. Chatterjee, S. Majumdar, S. R. Sahoo, P. P. Das, Debugging multi-threaded applications using pin-augmented gdb (pgdb), in: International conference on software engineering research and practice (SERP). Springer, 2015, pp. 109–115.

[7] S. Majumdar, N. Chatterjee, S. R. Sahoo, P. P. Das, D-cube: tool for dynamic design discovery from multi-threaded applications using pin, in: 2016 IEEE International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2016, pp. 25–32.

[8] S. Majumdar, N. Chatterjee, P. P. Das, A. Chakrabarti, A mathematical framework for design discovery from multi-threaded applications using neural sequence solvers, Innovations in Systems and Software Engineering 17 (2021) 289–307.

[9] S. Majumdar, N. Chatterjee, P. Pratim Das, A. Chakrabarti, Dcube_ nn d cube nn: Tool for dynamic design discovery from multi-threaded applications using neural sequence models, Advanced Computing and Systems for Security: Volume 14 (2021) 75–92.

[10] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, A. Brechmann, Measuring neural efficiency of program comprehension, in: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, 2017, pp. 140–150.

[11] N. Chatterjee, S. Majumdar, P. P. Das, A. Chakrabarti, Parallelc-assist: Productivity accelerator suite based on dynamic instrumentation, IEEE Access (2023).

[12] P. Oman, J. Hagemeister, Metrics for assessing a software system's maintainability, in: Proceedings Conference on Software Maintenance 1992, IEEE Computer Society, 1992, pp. 337–338.

[13] B. Fluri, M. Wursch, H. C. Gall, Do code and comments co-evolve? on the relation between source code and comment changes, in: 14th Working Conference on Reverse Engineering (WCRE 2007), IEEE, 2007, pp. 70–79.

[14] F. Deissenboeck, S. Wagner, M. Pizka, S. Teuchert, J.-F. Girard, An activity-based quality model for maintainability, in: 2007 IEEE International Conference on Software Maintenance, IEEE, 2007, pp. 184–193.

[15] M.-A. Storey, J. Ryall, R. I. Bull, D. Myers, J. Singer, Todo or to bug, in: 2008 ACM/IEEE 30th International Conference on Software Engineering, IEEE, 2008, pp. 251–260.

[16] T. Tenny, Program readability: Procedures versus comments, IEEE Transactions on Software Engineering 14 (1988) 1271.

[17] H. Yu, B. Li, P. Wang, D. Jia, Y. Wang, Source code comments quality assessment method based on aggregation of classification algorithms, Journal of Computer Applications 36 (2016) 3448.
[18] S. Majumdar, A. Bansal, P. P. Das, P. D. Clough, K. Datta, S. K. Ghosh, Automated evaluation of comments to aid software maintenance, Journal of Software: Evolution and Process 34 (2022) e2463.
[19] S. Majumdar, S. Papdeja, P. P. Das, S. K. Ghosh, Comment-mine—a semantic search approach to program comprehension from code comments, Advanced Computing and Systems for Security: Volume Twelve (2020) 29–42.
[20] S. Majumdar, A. Bandyopadhyay, S. Chattopadhyay, P. P. Das, P. D. Clough, P. Majumder, Overview of the irse track at fire 2022: Information retrieval in software engineering., in: FIRE (Working Notes), 2022, pp. 1–9.
[21] S. Majumdar, A. Bandyopadhyay, P. P. Das, P. Clough, S. Chattopadhyay, P. Majumder, Can we predict useful comments in source codes?-analysis of findings from information retrieval in software engineering track@ fire 2022, in: Proceedings of the 14th Annual Meeting of the Forum for Information Retrieval Evaluation, 2022, pp. 15–17.
[22] S. Majumdar, P. P. Das, Smart knowledge transfer using google-like search, arXiv preprint arXiv:2308.06653 (2023).
[23] P. Chakraborty, S. Dutta, D. K. Sanyal, S. Majumdar, P. P. Das, Bringing order to chaos: Conceptualizing a personal research knowledge graph for scientists., IEEE Data Eng. Bull. 46 (2023) 43–56.
[24] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al., Language models are few-shot learners, Advances in neural information processing systems 33 (2020) 1877–1901.
[25] S. Paul, S. Majumdar, R. Shah, S. Das, M. Ghosh, D. Ganguly, G. Calikli, D. Sanyal, P. P. Das, P. D Clough, A. Bandyopadhyay, S. Chattopadhyay, Generative ai for code metadata quality assessment, in: Proceedings of the 16th Annual Meeting of the Forum for Information Retrieval Evaluation, 2024.
[26] S. Paul, S. Majumdar, R. Shah, S. Das, M. Ghosh, D. Ganguly, G. Calikli, D. Sanyal, P. P. Das, P. D Clough, A. Bandyopadhyay, S. Chattopadhyay, Overview of the irse track at fire 2024: Information retrieval in software engineering, in: FIRE (Working Notes), 2024.
[27] S. Paul, S. Majumdar, A. Bandyopadhyay, B. Dave, S. Chattopadhyay, P. Das, P. D. Clough, P. Majumder, Efficiency of large language models to scale up ground truth: Overview of the irse track at forum for information retrieval 2023, in: Proceedings of the 15th Annual Meeting of the Forum for Information Retrieval Evaluation, 2023, pp. 16–18.
[28] S. Majumdar, S. Paul, D. Paul, A. Bandyopadhyay, S. Chattopadhyay, P. P. Das, P. D. Clough, P. Majumder, Generative ai for software metadata: Overview of the information retrieval in software engineering track at fire 2023, arXiv preprint arXiv:2311.03374 (2023).
[29] S. Majumdar, A. Varshney, P. P. Das, P. D. Clough, S. Chattopadhyay, An effective low-dimensional software code representation using bert and elmo, in: 2022 IEEE 22nd International Conference on Software Quality, Reliability and Security (QRS), IEEE, 2022, pp. 763–774.
[30] J. Keim, A. Kaplan, A. Koziolek, M. Mirakhorli, Does bert understand code?–an exploratory study on the detection of architectural tactics in code, in: European Conference on Software Architecture, Springer, 2020, pp. 220–228.
[31] J. R. Quinlan, Learning decision tree classifiers, ACM Computing Surveys (CSUR) 28 (1996) 71–72.
[32] A. K. Jain, J. Mao, K. M. Mohiuddin, Artificial neural networks: A tutorial, Computer 29 (1996) 31–44.
[33] P.-H. Chen, C.-J. Lin, B. Schölkopf, A tutorial on ν-support vector machines, Applied Stochastic Models in Business and Industry 21 (2005) 111–136.
[34] N. L. Afanador, A. Smolinska, T. N. Tran, L. Blanchet, Unsupervised random forest: a tutorial with case studies, journal of Chemometrics 30 (2016) 232–241.
[35] A. Natekin, A. Knoll, Gradient boosting machines, a tutorial, Frontiers in neurorobotics 7 (2013) 21.

[36] A. DeMaris, A tutorial in logistic regression, Journal of Marriage and the Family (1995) 956–968.
[37] C. Haruechaiyasak, A tutorial on naive bayes classification, Last update 16 (2008).
[38] P. Cunningham, S. J. Delany, k-nearest neighbour classifiers-a tutorial, ACM computing surveys (CSUR) 54 (2021) 1–25.
[39] G. Chen, A gentle tutorial of recurrent neural network with error backpropagation, arXiv preprint arXiv:1610.02583 (2016).