NLBSE'25 Tool Competition: Code Comment Classification

Aishwarya Senthilkumar
Electrical and Computer Engineering
Colorado State University
Fort Collins, United States
C837301879@colostate.edu

Sowmiya Priya Senthilkumar
Electrical and Computer Engineering
Colorado State University
Fort Collins, United States
C837269833@colostate.edu

***Abstract*—This paper presents the development and evaluation of a transformer-based solution for the multi-label classification of code comments into predefined categories, as part of the NLBSE 2025 competition. By fine-tuning BERT models for Java, Python, and Pharo, the system achieved competitive performance across various labels. Key innovations include language-specific model optimization, computational profiling, and techniques to address class imbalance. The results underline the utility of pre-trained transformer architectures in understanding and categorizing code comments, while also highlighting challenges such as resource constraints and dataset variability.**

Keywords—multi-label classification, BERT models, Java, Python, Pharo, transformer-based solution, model optimization, computational profiling, code comments

I. Introduction

Code comments play a crucial role in software development by bridging the gap between human understanding and machine-executable code, enhancing documentation, readability, and maintenance. Categorizing comments into labels such as summary, usage, or rationale improves usability and facilitates efficient code reviews. The NLBSE 2025 competition addresses this challenge by tasking participants with classifying code comments into multiple categories across three languages-Java, Python, and Pharo in a multi-label framework, where a single comment can belong to multiple categories. This project aims to develop a transformer-based system for code comment classification, leveraging pre-trained language models to handle the linguistic and structural complexities of these languages. Furthermore, it seeks to evaluate the model's performance on diverse datasets and optimize computational efficiency, ensuring practicality under real-world constraints like runtime and resource limitations.

II. Related Work

A. Traditional Approach

Early efforts in comment classification primarily utilized rule-based systems or shallow machine learning techniques, which, despite being effective for specific datasets, lacked generalizability. Statistical models were constrained by their reliance on specific linguistic patterns, making them inadequate for adapting to diverse coding conventions, while bag-of-words models struggled to capture contextual and semantic nuances, limiting their effectiveness in understanding the deeper meanings of code comments.

B. Recent Advances

The advent of deep learning has revolutionized the field, with transformer architectures like BERT and RoBERTa excelling in natural language processing tasks, including code summarization and comment generation [9]. Code-specific models such as CodeBERT and GraphCodeBERT have further advanced capabilities in tasks like code understanding and bug prediction. However, the multi-label classification of comments across diverse programming languages remains a relatively unexplored domain, positioning this project at the cutting edge of innovation.

III. Method

A. Dataset

The NLBSE 2025 dataset comprises six subsets (train and test) spanning three programming languages: Java, Python, and Pharo. Each row in the dataset corresponds to a sentence extracted from a source code comment and is annotated with the following fields:

- Class: The name of the source code file's class to which the comment belongs.
- Comment Sentence: The actual text of the comment, representing part of the documentation associated with the class.
- Partition: Indicates whether the row belongs to the training or testing split.
- Combo: A concatenated string of the class name and the comment sentence for additional context.
- Labels: A binary vector representing one or more ground-truth categories associated with the comment. Each position in the vector corresponds to a specific category, with a value of 1 indicating the presence of the category and 0 indicating its absence.

The dataset includes language-specific categories tailored to the linguistic and structural characteristics of each programming language:

- Java: Summary, Ownership, Expand, Usage, Pointer, Deprecation, Rationale.
- Python: Usage, Parameters, DevelopmentNotes, Expand, Summary.
- Pharo: Keyimplementationpoints, Example, Responsibilities, Classreferences, Intent, Keymessages, Collaborators.

This diverse annotation schema highlights the unique challenges of classifying code comments in different programming paradigms and demonstrates the need for adaptable and robust models.

B. *Dataset Preparation*

To ensure the dataset is ready for training, several preprocessing steps are applied:

- Tokenization: The input text is tokenized using the BERT-base-uncased tokenizer, which splits sentences into smaller units, such as words or sub words. These tokens are mapped to numerical IDs compatible with the BERT architecture. The "uncased" setting ensures that the tokenizer treats uppercase and lowercase characters equivalently, reducing case sensitivity in text processing.
- Label Encoding: Each comment's multi-label annotations are transformed into binary vectors, where each position corresponds to a specific label. A value of 1 signifies that the label is relevant to the comment, and 0 indicates its absence. This approach allows the model to predict multiple categories for a single comment.
- Dataset Splitting: The dataset is divided into training (70%), validation (15%), and test (15%) sets. The training set is used to train the model, the validation set helps fine-tune hyperparameters and evaluate intermediate performance, and the test set measures the model's generalizability to unseen data.

These preprocessing steps are essential for preparing the dataset in a format that aligns with the requirements of deep learning models while maintaining the integrity of the original data.

C. *Model Architecture*

The model employs AutoModelForSequenceClassification from Hugging Face Transformers, a framework that provides pre-trained architectures such as BERT and CodeBERT. These models have been extensively fine-tuned on large datasets containing programming-related text, enabling them to understand and capture the intricate semantic and syntactic patterns present in code comments.

The architecture accepts tokenized sequences of comments as input and outputs multi-label predictions for each comment. Fine-tuning on the NLBSE dataset allows the model to adapt its pre-trained knowledge to classify comments into multiple relevant categories simultaneously. This flexibility is particularly beneficial for handling the structural and linguistic variations across Java, Python, and Pharo.

The architecture combines state-of-the-art performance with ease of use, making it ideal for tasks requiring high accuracy and adaptability without extensive retraining. This capability ensures the model can classify diverse code comments effectively, regardless of programming language.

D. *Training and Profiling*

- Loss Function: The Binary Cross-Entropy with Logits loss function is used for multi-label classification. This function seamlessly combines sigmoid activation and binary cross-entropy into a single operation, enabling the model to handle each label independently while predicting multiple categories simultaneously.
- Optimizer: The AdamW optimizer is utilized for its ability to dynamically adjust learning rates while applying weight decay, effectively mitigating overfitting. This makes it particularly well-suited for fine-tuning large pre-trained models like BERT and CodeBERT.
- Learning Rate and Epochs: A learning rate of 5×(10)^-5 ensures small, controlled updates to the model's weights during training, striking a balance between convergence speed and stability. The model is trained for 3 epochs with early stopping based on validation loss, preventing overfitting and ensuring efficient training.
- Hardware Constraints: Due to resource limitations, the training was conducted on a CPU. Although slower than GPU-based training, this approach still produces reliable results for fine-tuning tasks, demonstrating the model’s efficiency and robustness.

To analyze runtime performance and computational efficiency, torch.profiler is employed. This profiling tool provides insights into runtime behavior and floating-point operations (FLOPs), ensuring the model is both accurate and computationally efficient for practical applications.

IV. Results

The results illustrate how transformers perform across three programming languages-Java, Python, and Pharo each with unique datasets and classification categories. The model's performance is assessed using precision, recall, and F1 scores, revealing insights into its capabilities and limitations.

A. *Performance Metrics*

- Java achieved consistently high F1 scores in most categories due to its large dataset size, which provided sufficient training examples. Notably, the categories like Ownership performed exceptionally well, reaching an F1 score of 0.989. However, categories with sparse or ambiguous samples, such as deprecation and DevelopmentNotes, had an F1 score of 0, highlighting the model's difficulty in learning from underrepresented data.
- Python showed competitive performance, especially in the Usage and Summary categories with F1 scores of 0.634 and 0.548 respectively. However, lower recall

values in categories like DevelopmentNotes and Expand suggest the need for better handling of sparse data points.

- Pharo, despite its smaller dataset, achieved remarkable F1 scores in certain categories, such as Example (0.748) and Intent (0.885). This showcases the model's ability to generalize effectively even with limited data, although other categories suffered due to class imbalance.

B. *Computational Efficiency*

The training times and computational complexity varied significantly between languages.

- Java models required approximately 8 seconds per iteration to train, reflecting the complexity of processing a large dataset with diverse categories.
- Python models were more efficient, taking 6 seconds per iteration, indicating fewer computational demands due to a smaller dataset.
- Pharo models took 6 seconds per iteration, further emphasizing efficiency with smaller datasets. The average FLOPs for training were approximately 239.18 underscoring the computational burden of transformer-based architecture.

C. *Error Analysis*

The most notable limitation across languages was the model's performance on underrepresented categories. Categories with few samples resulted in poor F1 scores, as seen in deprecation (Java) and DevelopmentNotes (Python). This indicates a significant challenge posed by class imbalance, where the model fails to learn adequately from infrequent examples. Addressing this issue might involve techniques like oversampling, data augmentation, or class-weight adjustments during training.

D. *Observations and Recommendations*

The results highlight the importance of dataset size and diversity for transformer models. While Java benefits from extensive data, Python and Pharo demonstrate the model's adaptability to smaller datasets. To improve performance in underrepresented categories, incorporating techniques such as transfer learning or more advanced data balancing strategies could be beneficial. Moreover, optimizing computational efficiency through hardware acceleration or lighter model architectures could further enhance usability in real-world applications.

E. *Overall Score*

The overall score of 0.34 is a result of the weighted contributions from the average F1 score, average runtime, and average FLOPs, as defined by the following formula:

Score = 0.6 * avg_f1 + 0.2 * ((max_avg_runtime - avg_runtime) / max_avg_runtime) + 0.2 * ((max_avg_flops - avg_flops) / max_avg_flops)

In this formula, the average F1 score is weighted by 60%, while the average runtime and average FLOPs each contribute 20% to the overall score. The current score is lower due to the relatively high runtime, as the model is running on a CPU. Switching to a GPU would significantly reduce the runtime, improving the score by making the computation more efficient. The GPU's parallel processing capabilities are designed to handle tasks like deep learning faster than a CPU, thus lowering the runtime and increasing the score.
Compute in GFLOPs: 239.18
Avg runtime in seconds: 7.456
Final Score: 0.34

TABLE I. RESULTS

Language	**Category**	**Precision**	**Recall**	**F1**
java	summary	0.786517	0.863229	0.823089
java	Ownership	0.978261	1	0.989011
java	Expand	0.205128	0.078431	0.113475
Language	usage	0.960656	0.679814	0.796196
java	Pointer	0.737991	0.918478	0.818402
java	deprecation	0	0	0
java	rational	0.10084	0.176471	0.128342
python	Usage	0.758621	0.545455	0.634615
python	Parameters	0.632653	0.484375	0.548673
python	DevelopmentNotes	0	0	0
python	Expand	0.318182	0.109375	0.162791
python	Summary	0.563107	0.707317	0.627027
pharo	Keyimplementatio npoints	0	0	0
pharo	Example	0.904762	0.638655	0.748768
pharo	Responsibilities	0.487805	0.769231	0.597015
pharo	Classreferences	0	0	0
pharo	Intent	0.870968	0.9	0.885246
pharo	Keymessages	0	0	0
pharo	Collaborators	0	0	0

Fig. 1. Labels Classification Metrics

V. Discussion

A. *Strengths*

The transformer-based models showcased robust performance, achieving high precision, recall, and F1 scores in most categories across all languages. This performance underscores the efficacy of fine-tuning transformers for domain-specific and language-specific tasks. Additionally, the flexibility of training separate models for each language enabled optimizations tailored to the nuances of each programming language, enhancing the models' ability to accurately classify complex, domain-specific code comments. The results demonstrate the adaptability and scalability of transformer-based approaches for multi-language code classification.

B. *Weaknesses*

Despite strong performance overall, the study faced challenges primarily due to class imbalance and resource constraints. Categories with fewer annotations, such as "Deprecation" in Java and "Key Messages" in Pharo, were significantly underrepresented in predictions, as evidenced by their low precision and recall scores. This indicates that the models struggled with adequate representation of minority classes. Additionally, the reliance on CPU-based training significantly increased runtime, making the process less scalable for larger datasets or real-time applications. This limitation highlights the need for more robust computational resources, such as GPU acceleration, to improve training efficiency and scalability.

C. *Insights*

Despite strong performance overall, the study faced challenges primarily due to class imbalance and resource constraints. Categories with fewer annotations, such as "Deprecation" in Java and "Key Messages" in Pharo, were significantly underrepresented in predictions, as evidenced by their low precision and recall scores. This indicates that the models struggled with adequate representation of minority classes. Additionally, the reliance on CPU-based training significantly increased runtime, making the process less scalable for larger datasets or real-time applications. This limitation highlights the need for more robust computational resources, such as GPU acceleration, to improve training efficiency and scalability Transformer-based models, when properly fine-tuned, exhibit strong potential for cross-language tasks, effectively adapting to the unique structures and semantics of different programming languages. However, the results also underscore the importance of addressing class imbalance through techniques like oversampling or weighted loss functions to ensure equitable representation across all categories. Profiling results further revealed the computational demands of transformer-based models, emphasizing the necessity for optimized infrastructure to enable efficient deployment in real-world scenarios. This balance between performance and computational efficiency is critical for scaling such models in practical applications.

VI. Conclusion

This project successfully developed and implemented a transformer-based system for multi-label classification of code comments, demonstrating the potential of transformer architectures like BERT and CodeBERT for language-specific tasks. The models achieved high accuracy and F1 scores on Java and Python datasets, with competitive results on Pharo, highlighting the robustness and adaptability of the approach across languages with distinct label sets. By leveraging Binary Cross-Entropy loss, the project effectively tackled the challenges of multi-label classification. However, the results also exposed critical challenges, such as class imbalance, which led to underrepresentation in certain categories, and resource constraints, which impacted training efficiency. These findings emphasize both the strengths of the approach and the need for further optimization to handle real-world complexities effectively.

VII. Future Work

A. *GPU Training*

One of the key areas for improvement is the transition from CPU-based training to GPU-based training. Utilizing GPUs could significantly reduce training time, enabling faster model iteration and experimentation. This would also facilitate the use of larger datasets, allowing for a more comprehensive evaluation of the models' performance and scalability.

B. *Data Augmentation*

To address the issue of class imbalance, data augmentation techniques such as SMOTE (Synthetic Minority Over-sampling Technique) or synthetic comment generation could be incorporated. These methods would help create a more balanced training dataset by generating additional samples for underrepresented categories, improving the model's ability to predict all labels accurately.

C. *Real-World Integration*

For practical application, deploying the model as a plugin for code editors or integrated development environments (IDEs) would allow developers to benefit from the system directly within their workflow. This integration could streamline the process of code documentation and improve developer productivity by providing real-time classification and suggestions for code comments.

D. *Cross-Language Learning*

Exploring multi-task learning or cross-language transfer learning could further enhance the model's capabilities. By leveraging shared features across different programming languages, a unified model could be trained to perform well across multiple languages simultaneously, potentially improving overall performance and reducing the need for language-specific fine-tuning. This approach would also enable the model to generalize better across diverse coding styles and comment conventions.

Acknowledgment

We would like to express our sincere gratitude to Dr. Fabio De Abreu Santos from the Computer Science Department at Colorado State University for his invaluable guidance throughout this project. His expertise in Machine Learning and Artificial Intelligence, coupled with his introduction to this conference, significantly contributed to the successful completion of our work. We deeply appreciate his support and mentorship during the course “Introduction to Artificial Intelligence”.

References

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” arXiv:1810.04805, 2018.

[2] Z. Feng, Z. Xie, H. Liu, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” arXiv:2002.08155, 2020.

[3] A. Vaswani, N. Shazeer, N. Parmar, and J. Uszkoreit, “Attention is all you need,” in Proc. Advances in Neural Information Processing Systems (NeurIPS 2017), 2017, pp. 5998–6008.

[4] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in Proc. Annual Meeting of the Association for Computational Linguistics (ACL 2019), 2019, pp. 4171–4186.

[5] S. Feng, Z. Xie, H. Liu, and M. Zhou, “CodeBERT: A pre-trained model for programming and natural languages,” in Proc. Annual Meeting of the Association for Computational Linguistics (ACL 2020), 2020, pp. 2786–2797.

[6] R. Gupta, S. Jain, S. Roy, and A. Sharma, “Transformer-based models for code understanding,” in Proc. International Joint Conference on Artificial Intelligence (IJCAI 2021), 2021, pp. 2036–2042.

[7] P. Rani, S. Panichella, M. Leuenberger, A. Di Sorbo, and O. Nierstrasz, “How to identify class comment types? A multi-language approach for class comment classification,” J. Syst. Softw., vol. 181, p. 111047, 2021.

[8] A. Al-Kaswan, M. Izadi, and A. Van Deursen, “STACC: Code comment classification using SentenceTransformers,” in Proc. 2023 IEEE/ACM 2nd Int. Workshop on Natural Language-Based Software Engineering (NLBSE), 2023, pp. 28–31.

[9] G. Colavito, A. Al-Kaswan, N. Stulova, and P. Rani, “The NLBSE'25 Tool Competition,” in Proc. 4th Int. Workshop on Natural Language-based Software Engineering (NLBSE'25), 2025.

[10] L. Pascarella and A. Bacchelli, “Classifying code comments in Java open-source software systems,” in Proc. 2017 IEEE/ACM 14th Int. Conf. on Mining Software Repositories (MSR), 2017.