NLBSE 2025 Code Comment Classification Using A Lightweight BERT Transformer

Brendan Scheidt, Coen Petto, Jeremiah Geisterfer
Computer Science
Colorado State University
bscheidt@colostate.edu, coenp@colostate.edu, jcgeist@colostate.edu

***Abstract*—Code comment classification interacts with software engineering and related disciplines by enhancing code comprehension and improving maintenance protocols. Despite the effectiveness of conventional transformer-based models, they often require substantial computational overhead and introduce excessive latency, rendering them less attractive for real-time applications. In this paper, we present a lightweight sentence transformer model designed to efficiently classify code comments into meaningful categories across multiple programming languages – Java, Python, and Pharo – without introducing a significant loss in performance. In our approach, we leverage augmenting the training data through synonym replacement to address the class imbalance, integrate a custom Focal Loss with label smoothing loss function to enhance model robustness, and employ a method of discovering optimal classification thresholds based on the maximization of the F1 score of each label. Through fine-tuning a compact pre-trained model (prajjwal1/bert-tiny) and using mixed precision inference, we deploy a model that can operate approximately 11 times faster than the given baseline while retaining the ability to produce comparable F1 scores. After experimentation, our results demonstrate that our model can achieve a higher submission score of 0.75 compared to the baseline's 0.70. This validates the efficacy of prioritizing computational efficiency while maintaining competitive predictive performance. Our findings suggest that a lightweight model will offer a viable solution for practical applications that require both speed and accuracy in code comment classification.**

I. Introduction

Code comments are essential for improving the readability and maintainability of software projects. They are one of a few options for programmers to explain their logic or provide context in a collaborative manner. The sheer volume of comments inside of large code bases presents a challenge to maintaining quality documentation. Efficiently classifying these comments into meaningful categories can greatly aid in automated documentation management, code review, and other software engineering practices [1] [2].

The goal of this paper is to provide a lightweight efficient means to this classification problem for the languages Java, Python, and Pharo. The structure of the paper is as follows: first, we discuss related work and key advancements in NLP and multi-label classification that make this paper possible. Next, we present our methodology, including pre-processing, oversampling, model architecture, custom loss function, and optimized thresholding. We then display our results and visualizations of our data augmentations, which are followed by detailed discussions of interpretations of our findings, implications for the future, and directions for future research.

II. Related Work

Natural Language Processing, text classification, and the use of transformers have been thoroughly researched, and many techniques have been developed to improve accuracy, use less computational space, and decrease training time.

BERT [3] (Bidirectional Encoder Representations from Transformers) has demonstrated high levels of effectiveness for pre-training downstream NLP tasks and can be used for natural language inference and token-level tasks. BERT allows for a single layer of fine-tuning instead of an entire architecture retraining for individual tasks.

Multilabel classification using thresholding [4] presents methods of adjusting the tipping point at which a label would be included in a prediction. This has shown improvements over fixed thresholding in the F1 score and precision while showing no negative impacts on computational costs. Focal loss has been shown to mitigate some imbalances in training data between classes [5] by downplaying the importance of well-represented classes during training. This allows the model to focus on samples that are harder to classify. By introducing a scaling factor into the loss function that decays as confidence in the correct class increases, the model adjusts the loss contribution of each sample during training. This eliminates some of the need for class balancing and improves the results of class detection. Nitesh V. Chawla et al. propose another method to address class imbalance using the Synthetic Minority Oversampling Technique or SMOTE [6]. SMOTE creates synthetic samples by interpolating between data points in underrepresented classes. The more generalized decision boundaries improve classifier performance and reduce overfitting compared to oversampling with sample duplication.

Jason Wei et al. focus on data augmentation for text classification tasks when working with small datasets; their EDA (Easy Data Augmentation) process uses several techniques that have been shown to improve accuracy on text classification problems [7]. They leverage synonym replacement to randomly replace nonstop words in a sentence with a synonym from a pool of candidates (WordNet) that retain the same part-of-speech as the original word. This process preserves context and improves generalizations while being straightforward to implement and incurring low resources to run.

Pre-Trained Distillation or PD uses a masked language model before training with knowledge distillation [8]. This method substantially compresses the model, allowing for faster training times and the use of less computational resources while maintaining the same level of accuracy as a larger model. This provides insights into how NLP models perform when compressed.

III. Method

A. Problem Definition

Our task focused on the multi-label classification of comment sentences attached to code snippets in three primary programming languages: Java, Python, and Pharo. We note that each sentence in the dataset may belong to multiple categories specific to its respective programming language, as defined by prior work [9]. Our goal was to not only develop a model that accurately predicts these categories, but also optimizes the computational efficiency across the task. Computational efficiency is measured as a metric involving inference runtime and giga floating-point operations per second (GFLOPS). We aimed to outperform the baseline submission score in the NLBSE'25 code comment classification competition [10]. In Fig. 1, we show the pipeline of data flowing through our experiment. The formula used to determine the success of our model accuracy and efficiency defined by the competition holders [10] is as follows:

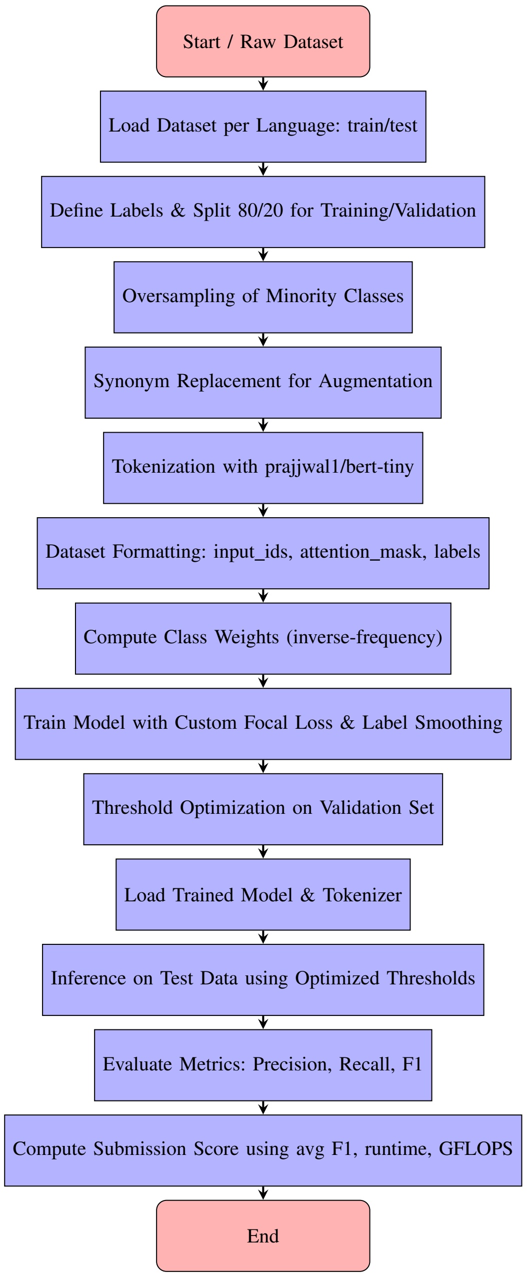
$$\begin{aligned}\text{submission\_score}(\text{model}) &= 0.60 \times \text{avg. } F_1 \\ &+ 0.2 \times \frac{\text{max\_avg\_runtime} - \text{measured\_avg\_runtime}}{\text{max\_avg\_runtime}} \\ &+ 0.2 \times \frac{\text{max\_avg\_GFLOPS} - \text{measured\_avg\_GFLOPS}}{\text{max\_avg\_GFLOPS}}.\end{aligned}$$

"Where:

- *avg. F1* is the average of the F1 scores achieved by the proposed classifiers (implementing a single model, e.g., SVM, BERT, etc.) across the 19 categories.
- *max_avg_runtime* indicates the maximum average inference runtime (using a Colab T4 instance) and the *measured_avg_runtime* is the actual average inference runtime of the proposed classifiers. The runtime should be measured 10 times and should be averaged across all categories and samples (i.e., code comments). Solutions with a runtime higher than *max_avg_runtime* are allowed.
- *max_avg_GFLOPS* indicates the maximum average compute in Giga floating Point operations per Second. Solutions with computational effort higher than *max_avg_GFLOPS* are allowed."

B. Dataset

For training and evaluating our model, we used a dataset provided by the competition organizers in the Hugging Face Dataset format which contains 14,875 code comment sentences that were extracted from 20 open-source projects [10].



Start / Raw Dataset
Load Dataset per Language: train/test
Define Labels & Split 80/20 for Training/Validation
Oversampling of Minority Classes
Synonym Replacement for Augmentation
Tokenization with prajjwal1/bert-tiny
Dataset Formatting: input_ids, attention_mask, labels
Compute Class Weights (inverse-frequency)
Train Model with Custom Focal Loss & Label Smoothing
Threshold Optimization on Validation Set
Load Trained Model & Tokenizer
Inference on Test Data using Optimized Thresholds
Evaluate Metrics: Precision, Recall, F1
Compute Submission Score using avg F1, runtime, GFLOPS
End

Fig. 1: High-level Data Flow and Processing Pipeline

The dataset was preprocessed by splitting it into training and testing sets for each programming language, leaving 10,555 combined samples for Java, 2,555 for Python, and 1,765 for Pharo. The sentences were additionally preprocessed by changing every character to its lowercase version, removing multiple line endings, and removing special characters respective to each language [10]. Each sample of the data contained but was not limited to: a comment sentence feature and a label feature represented as a binary vector that indicates the presence or absence of a language-specific category. In this dataset, Java and Pharo each had 7 categories that the binary label vector may represent, while Python had 5.

C. Data Preprocessing and Augmentation

Addressing the class imbalance inherent within the dataset's multi-label setting required careful implementation of a robust data augmentation strategy. Inspired by the Synthetic Minority Oversampling Technique (SMOTE) [6], we began by applying an oversampling technique to the training data while leaving the validation and test data unchanged. In our experiment, minority class samples of each language's training set were duplicated to help balance the distribution of examples for each classification category. This oversampling technique relied on calculating each training set's per-sample class frequency, inverting and normalizing it for balanced weighting, and duplicating samples based on the target sample count and previously computed class weight.

1) Frequency-Based Weighting Scheme: Consider a dataset of N samples that may take on one or multiple labels corresponding to C classes. Each sample comment, x_n, has a label vector, $\mathbf{y_n}$, of length C defined as:

$$\mathbf{y_n} = (y_{n,1}, y_{n,2}, \ldots, y_{n,C}) \in \{0,1\}^C$$

Then, for each class $i \in \{1, \ldots, C\}$, the number of samples in the dataset that have that class represented in their label vector is defined as:

$$c_i = \sum_{n=1}^{N} y_{n,i}$$

The frequency of each class i in the dataset is:

$$f_i = \frac{c_i}{N}$$

For each class i, the inverse frequencies would be:

$$\alpha_i = \frac{1}{f_i + \epsilon}$$

Where ϵ is a small constant to prevent division by 0. Normalize the inverse frequencies so their mean is 1:

$$\bar{\alpha}_i = \frac{\alpha_i}{\frac{1}{C}\sum_{j=1}^{C} \alpha_j}$$

Each sample, n, has multiple labels. The class weight of that sample was then defined as the sum of the normalized inverse frequencies of all classes it possesses:

$$w_n = \sum_{i=1}^{C} y_{n,i}\, \bar{\alpha}_i$$

Each sample's weights were converted into a probability distribution over all N samples:

$$p_n = \frac{w_n}{\sum_{m=1}^{N} w_m}$$

Where $\{p_n\}$ formed a valid probability distribution:

$$\sum_{n=1}^{N} p_n = 1$$

Furthermore, we introduced more diversity into our training data by applying synonym replacement to the duplicated samples; hyperparameters determine the target dataset size and the the number of words within the sentence to replace with their synonyms using WordNet [7].

2) Synonym Augmented Oversampling: Let $T > 1$ be the factor by which we increased the dataset size, then: $N' = T \times N$

Draw N' indices $s_1, s_2, \ldots, s_{N'}$ from $1, \ldots, N$ with replacement where each index n is chosen according to $\{p_n\}$.

Then, for each selected sample sentence x at index, s_k in dataset D, we applied the synonym replacement function $\mathrm{syn}()$ defined by prior work [11] to the comment x_{s_k}:

$$x_{s_k}^{aug} = \mathrm{syn}(x_{s_k}, n_{syn})$$

Where $syn(x_{s_k}, n_{syn})$ replaces n_{syn} words in x with synonyms. The final oversampled and augmented dataset is then defined as:

$$D' = \{(x_{s_1}^{aug}, y_{s_1}), (x_{s_2}^{aug}, y_{s_2}), \ldots, (x_{s_{N'}}^{aug}, y_{s_{N'}})\}$$

D. Model Architecture

When choosing a model for this task, we selected an architecture suitable for maintaining performance while balancing computational efficiency. We adopted a pre-trained tiny BERT model (prajjwal1/bert-tiny) [12] that contained two transformer layers and approximately 4 million parameters. Note that this model contains substantially fewer parameters than most standard BERT models with approximately 110 million parameters [3]. This smaller architecture was the main component that affected computational calculation during inference. Furthermore, to adapt the model to work for multi-label classification, we chose to introduce a custom classification head that outputs sigmoid-activated scores for each possible category, which allows independent probability prediction per label per language.

E. Custom Loss Function

To protect the model's ability to learn from imbalanced data and hard-to-classify samples, we utilized a custom loss function that combined Focal Loss and label smoothing, implemented based on ideas proposed in prior work [5], [13].

1) Label Smoothing: Label smoothing modifies a ground truth label $y_i \in \{0,1\}$ for each class i with a given smoothing factor $\epsilon \in [0,1]$. We defined a single smoothed label within the unmodified label vector $\mathbf{y}$ as:

$$y'_i = y_i(1-\epsilon) + \frac{\epsilon}{2}$$

Then, given a logit z_i and and a smoothed label y'_i , the binary cross-entropy loss was computed using the logistic function $\sigma(z) = \frac{1}{1+\epsilon^{-z}}$:

$$\mathrm{BCE}(z_i, y'_i) = -(y'_i \log(\sigma(z_i)) + (1 - y'_i)\log(1 - \sigma(z_i)))$$

Focal loss then added a focusing parameter $\gamma \geq 0$ and class weights α_i, and we computed:

$$pt_i = \exp(-\mathrm{BCE}(z_i, y'_i))$$

Noting that $\exp(-\text{BCE}(z_i, {y'}_i))$ is equivalent to:

$$p_{t_i} = \begin{cases} \sigma(z_i), & \text{if } y'_i = 1 \\ 1 - \sigma(z_i), & \text{if } y'_i = 0 \end{cases}$$

And the focal loss for class i is then:

$$\text{FL}(z_i, {y'}_i) = (\alpha_i) \cdot (1 - pt_i)^\gamma \cdot \text{BCE}(z_i, {y'}_i)$$

If α_i is not provided, it is assumed that $\alpha_i = 1\ \forall i$. Given that we have N samples and C classes, the total loss of the training epoch then becomes the average of the total loss aggregated over all samples and classes:

$$L_{mean} = \frac{1}{N \cdot C} \sum_{n=1}^{N} \sum_{i=1}^{C} \text{FL}(z_{n,i}, {y'}_{n,i})$$

Focal Loss reduces the relative loss for well-classified easy samples, enabling our model to focus on hard, misclassified ones [5]. Label smoothing, in conjunction, regularizes the model by mitigating overconfidence throughout training and improves generalization [13]. To aid in addressing the class imbalance problem, class weights that are inversely proportional to label frequencies were incorporated.

F. Training Procedure

To capture nuances and differences on a language-to-language basis, 3 models with the same architecture were trained, one on each programming language. Prior to training, validation sets were separated from the respective language's training data following an $80/20$ split of the original data utilizing Hugging Face's Dataset library and their test_train_split() function to sample equally probabilistic samples from the training data [14]. Data augmentation and synonym replacement were then performed on the training set while keeping the validation set unaltered to maintain the original class distribution during the evaluation of the validation set. Hyperparameters used in training, such as the learning rate and batch size, were then fine-tuned using the validation set after epochs. To help prevent overfitting, an early stopping callback was integrated, and mixed-precision training was enabled to optimize resource usage while speeding up training.

G. Threshold Optimization

To convert probabilistic outputs into binary label predictions, our experiment implemented a form of optimized threshold discovery that operated on the validation set's performance [4]. Setting these thresholds based on validation set performance instead of the test set was important to prevent the leakage of test data into the model's training.

1) Optimized Threshold Calculation: Let the set $\{(p_{n,i}, y_{n,i})\}_{n=1}^{N}$ represent the probabilities and true labels on the validation set. For a set of candidate thresholds $\mathcal{T} = \{t_1, t_2, \ldots, t_m\}$ within the range of $[0.1, 0.9]$, let the predicted label for a given threshold t of class i be defined as:

$$\hat{y}_{n,i}(t) = \begin{cases} 1, & p_{n,i} \geq t \\ 0, & p_{n,i} < t \end{cases}$$

The F1 score is then computed for each candidate threshold:

$$\text{F1}_i(t) = \frac{2 \cdot \text{Precision}_i(t) \cdot \text{Recall}_i(t)}{\text{Precision}_i(t) + \text{Recall}_i(t)}$$

Where:

$$\text{Precision}_i(t) = \frac{\sum_n [\hat{y}_{n,i}(t) = 1 \ \wedge \ y_{n,i} = 1]}{\sum_n [\hat{y}_{n,i}(t) = 1]}$$

$$\text{Recall}_i(t) = \frac{\sum_n [\hat{y}_{n,i}(t) = 1 \ \wedge \ y_{n,i} = 1]}{\sum_n [y_{n,i} = 1]}$$

For optimization, we selected the threshold candidate that maximizes this category's F1 score:

$$t_i^* = \arg\max_{t \in \mathcal{T}} \ \text{F1}_i(t)$$

Then, for a given language L and its number of classes C, we defined the optimal thresholds for each class $i \in \{1, \ldots, C\}$ to be a vector:

$$\mathbf{t}_L^* = [t_1^*, t_2^*, \ldots, t_C^*]$$

We improved the model's overall predictive performance on the testing data by selecting predictive thresholds that maximized the F1 score for each label on the validation set.

H. Evaluation Metrics

On the test set, model performance was assessed using precision, recall, and F1 scores for each language's classification category, along with the average F1 score across all categories. For the computational efficiency, evaluation was based on inference runtime and GFLOPS, following the competition scoring formula [10]. To measure inference, we integrated mixed precision and the previously calculated optimized thresholds to the probabilistically converted logits to determine binary predictions per label.

1) Inference Process: Given a test dataset with N samples and C classes, input tensors X representing tokenized inputs, a trained model that outputs a logits matrix $\mathbf{Z} \in \mathbb{R}^{N \times C}$, and a vector of class-specific optimized thresholds $\mathbf{t}^* = [t_1, t_2, \ldots, t_C]$:

For each test sample $n \in \{1, \ldots, N\}$ and class $i \in \{1, \ldots, C\}$ the model outputs a logit $z_{n,i}$ and leaves us with:

$$\mathbf{Z} = [z_{n,i}]_{N \times C}$$

Where $z_{n,i} = \text{model}(X)_i$.

Let $\mathbf{P} = [p_{n,i}]_{N \times C}$ denote the probability matrix with $p_{n,i}$ defined as the sigmoid function $\sigma(z) = \frac{1}{1+e^{-z}}$ applied to each logit:

$$p_{n,i} = \sigma(z_{n,i}) = \frac{1}{1 + e^{-z_{n,i}}}$$

Given the optimized threshold vector $\mathbf{t}^*$ for each class i, a prediction is then positive if $p_{n,i} \geq t_i$, defined previously as:

$$\hat{y}_{n,i} = \begin{cases} 1, & p_{n,i} \geq t_i \\ 0, & p_{n,i} < t_i \end{cases}$$

Thus, a predicted label matrix $\hat{\mathbf{Y}} = [\hat{y}_{n,i}]_{N \times C}$ is constructed by an element-wise comparison of $\mathbf{P}$ and $\mathbf{t}^*$. Our code then returns the transpose of this prediction matrix $\hat{\mathbf{Y}}^T \in \{0,1\}^{C \times N}$ and the complete inference process is defined as:

$$\hat{\mathbf{Y}}^T = [(p_{n,i} \geq t_i)]_{C \times N}$$

$$\text{where} \quad p_{n,i} = \frac{1}{1 + e^{-z_{n,i}}} \quad , \quad z_{n,i} = \text{model}(X)_i$$

I. Computational Efficiency Measures

As previously explained, to achieve significant reductions in inference time and computational cost, we employed mixed-precision inference using NVIDIA's AMP (Automatic Mixed Precision) [15]. This approach leverages lower-precision arithmetic without the compromise of the model's accuracy. In accordance with the baseline, we used PyTorch's profiler to measure GFLOPS during inference to align our calculated metrics with competition requirements.

J. Implementation Details

Experiments were conducted on Google Colab's T4 GPU with high RAM in a framework structured in accordance with the competition's guidelines to generate a fair comparison to the baseline model. Our strategy aimed to achieve efficient model deployment by minimizing memory footprint and optimizing tensor operations.

IV. Results

A. Model Accuracy

All calculations and training steps for our model were ran on Google Colab using their T4 High RAM GPU, matching the baseline environment. Table I shows our model's evaluation metrics compared to the baseline's (Table II). Although our average F1 score is slightly less than the baseline's across most categories, they are meaningful considering that our trained model runs 11 times faster.

lan	cat	precision	recall	f1
java	summary	0.852679	0.856502	0.854586
java	Ownership	0.978261	1.000000	0.989011
java	Expand	0.318182	0.343137	0.330189
java	usage	0.929155	0.791183	0.854637
java	Pointer	0.799043	0.907609	0.849873
java	deprecation	0.666667	0.666667	0.666667
java	rational	0.211268	0.220588	0.215827
python	Usage	0.744444	0.553719	0.635071
python	Parameters	0.740458	0.757812	0.749035
python	DevelopmentNotes	0.302326	0.317073	0.309524
python	Expand	0.434211	0.515625	0.471429
python	Summary	0.581395	0.609756	0.595238
pharo	Keyimplementationpoints	0.625000	0.581395	0.602410
pharo	Example	0.853211	0.781513	0.815789
pharo	Responsibilities	0.689655	0.384615	0.493827
pharo	Classreferences	1.000000	0.250000	0.400000
pharo	Intent	0.657895	0.833333	0.735294
pharo	Keymessages	0.722222	0.604651	0.658228
pharo	Collaborators	0.076923	0.500000	0.133333

TABLE I: Our Model's Results

lan	cat	precision	recall	f1
java	summary	0.873385	0.829448	0.850850
java	Ownership	1.000000	1.000000	1.000000
java	Expand	0.323529	0.444444	0.374468
java	usage	0.911043	0.818182	0.862119
java	Pointer	0.738255	0.940171	0.827068
java	deprecation	0.818182	0.600000	0.692308
java	rational	0.162162	0.295082	0.209302
python	Usage	0.700787	0.735537	0.717742
python	Parameters	0.793893	0.812500	0.803089
python	DevelopmentNotes	0.243902	0.487805	0.325203
python	Expand	0.433628	0.765625	0.553672
python	Summary	0.648649	0.585366	0.615385
pharo	Keyimplementationpoints	0.636364	0.651163	0.643678
pharo	Example	0.872881	0.903509	0.887931
pharo	Responsibilities	0.596154	0.596154	0.596154
pharo	Classreferences	0.200000	0.500000	0.285714
pharo	Intent	0.718750	0.766667	0.741935
pharo	Keymessages	0.680000	0.790698	0.731183
pharo	Collaborators	0.260870	0.600000	0.363636

TABLE II: Baseline Results

B. Class Distributions

The following figures (Fig. 2, Fig. 3, Fig. 4) are the distributions of each class before (left) and after (right) our custom oversampling method which utilizes synonym replacement.

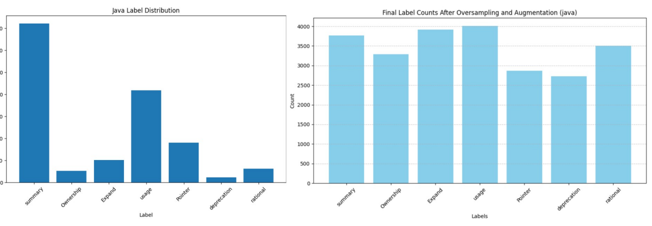


Fig. 2: Class distributions of the Java language Before (Blue) vs. After (Teal) oversampling with data augmentation.

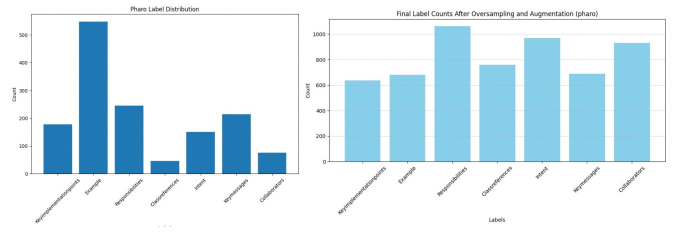


Fig. 3: Class distributions of the Pharo language Before (Blue) vs. After (Teal) oversampling with data augmentation.

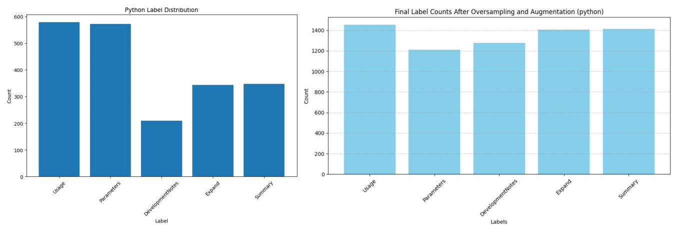


Fig. 4: Class distributions of the Python language Before (Blue) vs. After (Teal) oversampling with data augmentation.

C. *Computational Cost*

After running our training phase, the inferential prediction phase was closely monitored. The inference step of prediction was monitored over an average of 10 repetitions, keeping track of average computation time and GFLOPs. Running on the same environment as the baseline, our experiment achieved an average of 243.88685004799999 GFLOPs across inference and took an average of 0.0898674488067627 seconds to run. The baseline had an average of 999.0271426559999 GFLOPs and took an average of 0.9800318241119385 seconds to run. These results demonstrate a $\sim$ 75.59% decrease in GFLOPs and a $\sim$ 90.83% decrease in runtime for our model in its inference phase.

V. Discussion

The results of our experiment demonstrate the efficacy of achieving computational efficiency while maintaining accuracy. By leveraging a lightweight transformer model and class balancing techniques, our model achieved a submission score of 0.75, outperforming the baseline score of 0.70.

One of the key factors contributing to this score was the use of oversampling minority classes by augmenting the data with synonym replacement. This ensures that underrepresented categories were adequately learned, reducing the likelihood of misclassification in the prediction step [7]. The addition of label smoothing further improved generalization by preventing the model from becoming overly confident in its predictions.

Another important factor of our model's success was the custom focal loss function. This loss function helped the model focus on harder-to-classify samples by down-weighting the loss contribution of well-classified examples [3]

Threshold optimization further fine-tuned our model by dynamically adjusting the classification threshold for each class in the validation phase of each epoch for use on the testing data during inference [4]. This is important for our multi-label classification problem because we can then adjust decision boundaries for individual classes to enhance their overall F1 scores. For instance, certain classes may be influenced by keywords in the input, and optimizing the threshold can improve the model's ability to correctly identify the keyword's presence. This approach ensures that the model more accurately captures the significance of relevant features.

These combined enhancements significantly improved the F1 score across all categories, particularly for minority classes. In addition to preprocessing and architecture decisions, we employed multiple time and memory conscious decisions like mixed-precision inference to increase this model's computational efficiency. Our model demonstrated a significant reduction in inference time, operating approximately 11 times faster than the baseline. This can be attributed to small but effective decisions like casting down to 16 floating point computations, using the pre-calculated logits, and not computing any gradients.

In summary, our approach combined effective class balancing techniques, innovative loss functions, and computational optimizations to achieve significant performance improvements. The lightweight architecture delivers adequate accuracy but, most importantly, drastically reduces inference time, making the model viable for real-world applications. These results showcase the synergy of conscious data preparation and avoidance of unnecessary computation.

VI. Conclusion

Natural Language Processing and text classification are not new areas of study; however, advancements in techniques and processes continue to further understanding of these areas. In this experiment, we developed and evaluated a multi-label classifier created for code classification of three programming languages, Java, Python, and Pharo. The goal was to train, fine-tune, and evaluate our model on the provided data set and beat the baseline score of 0.70 [9]. Our final model outperformed the baseline with a score of 0.75, using a compact and lightweight transformer designed for low computation resources. The use of a compact model [8] allowed significantly reduced training times, increasing the submission score. These results highlight the effectiveness of using a lightweight transformer with data augmentation and oversampling when provided with a small unbalanced dataset.

While our model did not outperform most of the baseline F1 scores or accuracy, the low use of computational resources makes our model an excellent choice for real-world applications. The improvement in scores between the baseline and our fine-tuned models shows the advantages that data augmentation, focal loss, and oversampling can have when training a model for a specific task.

VII. Future work

Several factors limited our ability to investigate the given code comment classification dataset. The largest limitation facing us was the available computational resources, which restricted our ability to experiment with larger transformer models trained on a broader, more diverse data set. Future work could address this problem by leveraging large models like BERT [3]. Additionally, further fine-tuning of the transformer model and the custom classification head could significantly improve performance, particularly when working with imbalanced datasets.

Continued exploration of sample augmentation could improve current techniques for code classification problems. Our experiment utilized data augmentation using synonym replacement inspired by Jason Wei et al., they propose additional approaches to data augmentation including context-aware augmentation that explore alternative methods for creating synthetic samples while preserving meaning other than synonym replacement [7].

Future exploration into code comment classification could go beyond the current data set and include comments from a wide range of programming languages and software platforms. Expanding the dataset could illustrate how specifically trained models generalize and provide insights into how programming syntax impacts classification performance.

References

[1] L. Pascarella and A. Bacchelli, ‘Classifying code comments in Java open-source software systems’, in 2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR), 2017.

[2] A. Al-Kaswan, M. Izadi, and A. Van Deursen, ‘Stacc: Code comment classification using sentencetransformers’, in 2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE), 2023, pp. 28–31.

[3] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding,” Proceedings of the 2019 Conference of the North, vol. 1, 2019, doi: https://doi.org/10.18653/v1/n19-1423.

[4] J. R. Quevedo, O. Luaces, and A. Bahamonde, “Multilabel classifiers with a probabilistic thresholding strategy,” Pattern Recognition, vol. 45, no. 2, pp. 876–883, Jan. 2011, Available: https://www.researchgate.net/publication/285805721_Multilabel _classifiers_with_a_probabilistic_thresholding_strategy

[5] T. -Y. Lin, P. Goyal, R. Girshick, K. He and P. Dollár, ”Focal Loss for Dense Object Detection,” in IEEE Transactions on Pattern Analysis and Machine Intelligence, vol. 42, no. 2, pp. 318-327, 1 Feb. 2020, doi: 10.1109/TPAMI.2018.2858826.

[6] N. V. Chawla, K. W. Bowyer, L. O. Hall, and W. P. Kegelmeyer, ”SMOTE: Synthetic Minority Over-sampling Technique,” Journal of Artificial Intelligence Research, vol. 16, pp. 321–357, 2002.

[7] J. Wei and K. Zou, ”EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks,” in Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing, 2019, pp. 6383–6389.

[8] Iulia Turc, M.-W. Chang, K. Lee, and K. Toutanova, “Well-Read Students Learn Better: The Impact of Student Initialization on Knowledge Distillation,” arXiv (Cornell University), Aug. 2019.

[9] 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’, Journal of systems and software, vol. 181, p. 111047, 2021.

[10] G. Colavito, A. Al-Kaswan, N. Stulova, and P. Rani, ‘The NLBSE’25 Tool Competition’, in Proceedings of The 4th International Workshop on Natural Language-based Software Engineering (NLBSE’25), 2025.

[11] EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks Github Repository [Online] Available: https://github.com/jasonwei20/eda_nlp/blob/master/code/eda.py

[12] P. Bhargava, A. Drozd, and A. Rogers, ”Generalization in NLI: Ways (Not) To Go Beyond Simple Heuristics,” arXiv preprint arXiv:2110.01518, 2021.

[13] C. Szegedy et al., ”Rethinking the Inception Architecture for Computer Vision,” in Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, 2016, pp. 2818–2826.

[14] T. Wolf et al., ”Transformers: State-of-the-Art Natural Language Processing,” in Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: System Demonstrations, 2020, pp. 38–45.

[15] NVIDIA, ”Mixed Precision Training,” [Online]. Available: https://docs.nvidia.com/deeplearning/performance/mixed-precision-training/index.html