

Transforming Code Review with DistilBERT: Multi-Label Classification of Code Comments in Java, Python, and Pharo

Austin Youngren, Josiah Hegarty, Clay Roberson

January 6, 2025

Abstract

Code comments play a crucial role in enhancing the readability and maintainability of software by bridging human intent with machine logic. However, variations in commenting styles among developers and across programming languages present substantial challenges. This study explores the use of the lightweight, pre-trained transformer model, DistilBERT, for the multi-label classification of code comments in Java, Python, and Pharo. By comparing DistilBERT’s performance with the setFit baseline model, we assess its ability to classify comments based on their natural language content and prioritize relevant information in software engineering tasks. Our results show that DistilBERT outperforms the baseline model in both classification accuracy and inference speed, particularly when using optimal hyperparameters, including a learning rate of $5e-5$, a batch size of 4, and 20 epochs. These findings suggest that DistilBERT offers an improved solution for automating the classification of code comments, ultimately reducing cognitive load and improving the software review process. Future work will focus on refining this model, expanding its language support, and exploring multi-task learning techniques for broader applications across software engineering tasks.

1 Introduction

While writing, editing, and reading source code, developers often leave comment code to explain, document, and justify design choices. Comments are an invaluable tool for increasing readability of code, serving as the bridge between human intent and the machine logic of source code. However, inconsistencies in commenting styles, both between individual developers and across programming languages, can pose significant challenges.

For example, while reviewing code, developers often need to invest a significant amount of time

and effort interpreting comments instead of focusing on the code itself [Rani et al., 2021]. Additionally, not all code comments are helpful for a given task, and modern quality assurance practices, such as Modern Code Review (MCR), can quickly lead to an overwhelming number of comments that are difficult for developers or automated systems to process [Al-Kaswan et al., 2023]. In response, there is a growing demand for effective multi-label text classification tools capable of labeling code comments based on their natural language content. These tools aim to help developers and automated systems prioritize the most relevant comments and source code for their software engineering tasks [Ochodek et al., 2022].

In this paper, we explore the effectiveness of using a lightweight, pretrained transformer model, known as DistilBERT, for building this kind of multi-label code comment classification system. To do this, we compare this model’s performance in classifying comments from three languages: Java, Python, and Pharo, to that of the setFit model; a well-known baseline for multi-label text classification. Our hope is that DistilBERT’s general reliability and relatively fast inference times will provide a novel method for performing multi-label code comment classification that is competitive with, if not better than, our current baseline model.

2 Related Work

The article *“How to Identify Class Comment Types? A Multi-language Approach for Class Comment Classification”* [Rani et al., 2021] emphasizes the classification of class comments across Java, Python, and SmallTalk. It explores the variations in commenting styles and the information provided by the content across these languages to build a system that categorizes comments to support developers. The findings highlight distinct language-specific practices, such as Java’s reliance on Javadoc, Python’s informal and sparse comment style, and SmallTalk’s standalone detailed

class comments.

This project is built based on the specifications from *"2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE 2025)"* [Al-Kaswan et al., 2023]. This competition challenges programmers to build a model for the classification of comments in Java, Python, and Pharo that surpasses their baseline model. The NLBSE 2025 competition regulates the system to Google Collab, which utilizes a T4 graphics card to ensure there is no system bias between competitors. The NLBSE competition measures a submission score based on runtime, FLOPs, and accuracy for a total of 19 labels. The languages and their labels can be seen in Figure 1.

```
labels = {
    'java': ['summary', 'Ownership', 'Expand', 'usage', 'Pointer', 'deprecation', 'rational'],
    'python': ['usage', 'Parameters', 'DevelopmentNotes', 'Expand', 'Summary'],
    'pharo': ['keyImplementationpoints', 'Example', 'Responsibilities', 'ClassReferences', 'Intent', 'Keymessages', 'Collaborators']
}
```

Figure 1: Each language and their respective labels within the data set.

This project further references an article aiming to classify the primary purpose of comment code written by software developers in Java *"Classifying code comments in Java open-source software systems"* [Pascarella and Bacchelli, 2017]. The authors analyzed six projects to understand the intention behind code comments and categorize them based on their content. They manually classified over 2,000 comments to reveal patterns and commonalities. The insights gained from this paper were the types and frequencies of comments in these projects that were utilized in a machine learning approach for automatic classification.

3 Methodology

In this section, we describe the methodology used in our study, which includes data preprocessing, model architecture, and the testing procedures. Specifically, we first explain the data preprocessing steps required to prepare the dataset for the model. Then, we describe the model architecture, including the pre-trained DistilBERT transformer model used for classification. Finally, we discuss the experimental setup and hyperparameter tuning approach.

3.1 Data Preparation for Model Training

To get DistilBERT to work with the given baseline model we needed to tokenize the data prior to the model running. Specifically, we tokenized the 'combo' column of the dataset, which contains the natural language of the comment and the class. We further preprocessed the data by changing the labels format to torch tensors and converting them

from integers into float values. These steps were required for the Trainer class to function.

3.2 Model Architecture

For this project we utilized a pre-trained transformer language model, known as DistilBERT, which is a model designed for natural language processing [Sanh et al., 2020]. This model was created using knowledge distillation, a process that trains a smaller model to mimic the behavior of a larger, more robust model. Additionally, like BERT, DistilBERT primarily uses Masked Language Modeling (MLM), where random tokens in a sequence of code or text are masked, and the model is trained to predict these masked tokens based on the surrounding context. Because of this, DistilBERT is 40% smaller than BERT while being able to achieve close to 97% of BERT's language understanding, while having 60% faster inference times.

We believe that due to DistilBERT's faster inference speeds, and due to the model's ability to efficiently capture contextual and syntactical relationships in text, it will be capable of effectively classifying code comments for Java, Pharo, and Python at a higher and faster level than the paraphrase-MiniLM-L3-v2 used in the NLBSE 2025 [Colavito et al., 2025] baseline code.

To test this, we chose to test various hyperparameter values through the utilization of a grid search. The code was written using Jupyter Notebooks [Jupyter, 2024] in Python version 3.12 [Python Software Foundation, 2024], with our preferred framework being PyTorch [PyTorch, 2024]. Additionally, wandb.ai [Biewald, 2020] was also used to track and analyze the metrics of our trained models during and after training.

3.2.1 Hyperparameter Tuning

The dataset encompasses three programming languages, each with a distinct number of labels: Java with 7 labels, Python with 5 labels, and Pharo with 7 labels. Using similar logic as the NLBSE 2025 baseline code [Colavito et al., 2025], we decided on building three models, one for each language, then used grid search to tune models across all combinations of architectures. This would allow us to easily separate the results for each language in exchange for increasing computation time. We utilized a grid search to fine tune our hyperparameter selection, using for-loops, to increase model accuracy and performance.

Through performing a grid search with different values for the hyperparameters of number of epochs, batch sizing, and learning rate, we found that certain hyperparameter values used for in the original code, such as learning rate (5e-5), were

the best choice, while the hyperparameter values still had room for improvement. The hyperparameter values for the number of epochs was changed from 5, which was used in the competition baseline model, to 20. Additionally, the batch size was greatly reduced from the competition baseline of 32 to 4.

3.2.2 Performance Metric Functions

To obtain the necessary metrics to evaluate model performance, we chose to build a custom function that allowed our metrics to match the specific formatting needs, set by the NLBSE 2025 [Colavito et al., 2025], for the competition’s preferred scoring method. The formatting for the competition required us to build a dictionary of the scoring, which tracked the language, label category, precision, recall, and average harmonic mean of precision and recall (F1) associated with the model. We chose to include an additional descriptor to the dictionary which contained the ‘model name,’ which was a descriptor of the model’s architecture, for our grid search.

Additionally, since we needed language and label data for the dictionary, we built a wrapper function that handled the evaluation results, then fed in the language and label data. The calculations themselves were rather straightforward as F1, average precision, and average recall could be pulled from the evaluation results.

We were also required to measure the runtime and FLOPs needed for the trained model to predict the labels of each sample in the test set. To accomplish this, we defined a separate function that records the amount of time needed by the model to predict the labels of a preprocessed dataset. Within this function we utilized a PyTorch profiler to accurately measure the FLOPs. For the final run, these measurements were made a total of 10 times for each language in the test set, and the averaged the results of the runtime and FLOPs were recorded for each.

3.3 Model Performance Scoring

The submission score is represented with the score function, which is a weighted composite score. This formula is given by the NLBSE 2025 [Colavito et al., 2025] and evaluates the quality of the model by utilizing performance metrics, including the average F1 metric, average runtime, and average FLOPs across all models. The F1, precision, and recall metrics are defined as follows by the NLBSE 2025 [Colavito et al., 2025] competition:

$$P_c = \frac{TP_c}{TP_c + FP_c} \quad (1)$$

$$R_c = \frac{TP_c}{TP_c + FN_c} \quad (2)$$

$$F_{1,c} = 2 \cdot \frac{P_c \cdot R_c}{P_c + R_c} \quad (3)$$

where TP_c , FP_c and FN_c represent the true positives, false positives, and false negatives for a category c , respectively.

The average F1 is weighted with the highest priority at 0.6, reflecting the model’s ability to accurately classify the comments. Average runtime is weighed at 0.2 and penalizes models with higher runtime than the maximum allowable time of 5 seconds. Average FLOPs are weighted at 0.2 and penalizes models with higher FLOPs compared to the maximum value of 5000 FLOPs. The final score is the weighted sum of these three components, which ensures a balanced evaluation that considers both model accuracy and efficiency.

The formula for the submission score can be found in Appendix C.

4 Results

In this section, we present the results of our experiments with the DistilBERT-based model for multi-label classification of code comments. Our primary focus was to evaluate the impact of various hyperparameters, such as learning rate, batch size, and the number of epochs, on the model’s performance. We conducted a grid search to optimize these hyperparameters and assessed the model’s effectiveness using metrics like the average F1 metric, precision, and recall. Additionally, we discuss the performance and results of the final model configuration, including an analysis of how the chosen hyperparameters influenced the overall classification accuracy across the Java, Python, and Pharo datasets.

4.1 Grid Search

To ensure this model was the best case, we took to additional statistics concerning average F1 metric. When analyzing the learning rates of the separate model types (Figure 6, Appendix A.5), or when analyzing the average F1 metrics of all models categorized by learning rate (Figure 3, Appendix A.2), we can see that a learning rate value of 5e-5 consistently out-performs its 5e-6 counterparts. This indicates that learning rates of 5e-6 or smaller may prevent the model from effectively learning certain patterns within the data.

When looking at batch size values using data provided in Figure 6, Appendix A.5, the value of batch size on model performance is not entirely clear. When analyzing the models using a learning rate of 5e-5, and paring them via maximum

epochs (i.e., the model with learning rate = $5e-5$, epoch = X, batch size = 4, compared to the model with learning rate = $5e-5$, epoch = X, batch size = 8), there does not seem to be specific batch size value that consistently outperforms another. We do see that the batch size of 4 outperforms their counterpart for models whose epochs equal 10 and 20, but a batch size of 8 outperforms its counterpart when epochs equal 15.

Though batch sizes of 4 performed better than batch sizes of 8 in two of the three architecture types (epochs=[10, 15, 20], learning rate= $5e-5$), we found the data insufficient to justify the value of 4 as the preferred batch size for the final model. Using the data in Figure 4, which averages the F1 metrics of the models based on architecture batch sizes, we were able to determine that batch sizes of 4 improve F1 metrics by 0.03543 more than batch sizes of 8 on average.

The performances of models based on epochs, when looking at Figure 6, it is hard to discern whether the number of epochs has a large impact on model performance. Again, we turn to the average F1 metrics of the models based on its architecture use case for the specified hyperparameter, epochs. It is in this table (Figure 5, Appendix A.4), that we can see a clear improvement as the number of epochs increase. We see a steady rise in model performance as the maximum number of epochs increase, making 20 maximum epochs the best choice. Though, it is unclear if a value larger than 20 will be beneficial and will need to be further tested in future studies.

4.2 Final Model

As shown in Figure 7, we can see that the final DistilBERT model was able to provide predictions for the test dataset with an average F1 score of 0.7241 for Java, 0.5747 for Python, 0.6555 for Pharo, and an overall average F1 score of 0.6595 across all three languages. This is compared to the baseline model results, shown in Figure 10, which achieves an average F1 score of 0.6978 for Java, 0.6030 for Python, 0.6068 for Pharo, and an overall average F1 score of 0.6394 across all three languages tested.

In addition, Figure 7 shows that the DistilBERT model, on average, took approximately 0.03505 seconds and used approximately 32.6 flops for its forward pass of the test set. While the results of the baseline model, shown again in Figure 10, indicate that it took approximately 0.8853 seconds and approximately 999.0 flops on average for its forward pass predictions on of the test set data.

Finally, as shown in Figures 9 and 12, the final submission score of DistilBERT model was 0.79 while the submission score for the baseline model

was 0.71.

5 Discussion

5.1 Evaluating the Optimal Hyperparameters for Grid Search

The architecture that performed the best, in accordance with F1 metrics, utilized a learning rate= $5e-5$, a batch size=4, and 20 maximum epochs. Though, this architecture only outperformed the next best by an F1 metric of roughly 0.0007. The second-place model’s architecture only differed in its batch size, 8. Though the batch size seemed to be the least influential hyperparameter of the three tested, utilizing a larger batch size of 8 could prove beneficial for systems unable to utilize a batch size of 4 due to limited GPU memory.

When analyzing the architectures tested for the grid search, the learning rate looks to be the hyperparameter which affected model performance the most, as we see the largest gaps of F1 metrics between architectures when only the learning rate changes between architectures. Learning Rates of $5e-6$ and smaller seem to weaken performance, while $5e-5$ consistently provide F1 metrics greater than 0.75. Though, it is unclear if learning rates larger than $5e-5$ will improve or weaken model performance.

The second most influential hyperparameter was the number of epochs used to train the models. As the maximum value for epoch increases, the F1 metric increases for each architecture which utilizes the same learning rate and batch size. Showing that the model benefits the most from 20 epochs. Though, it is unclear if increasing the number of epochs past 20 will be beneficial to the model.

5.2 Final and Baseline Model Comparison

The final average F1 scores of the DistilBERT model (Figures 7&8) and the baseline model, (Figures 10&11), indicate that the DistilBERT model was slightly more successful in classifying the test set code comments for Java, and Pharo, while the baseline model was marginally more successful in classifying the comments for Python. Based on the average F1 scores of 0.6595 for DistilBERT and 0.6394 for the baseline, the results also indicate that the DistilBERT model was, on average, slightly more successful than the baseline model in accurately classifying the comments across all languages tested.

While the final F1 results were reasonably similar for the DistilBERT and baseline model, the timing and profiling information shown in Fig-

ure 7 and 10 indicate that the DistilBERT model was able to perform predictions significantly faster than the baseline model, and with considerably less computational overhead. These findings are also reflected in the submission scores for each model, shown in Figure 9 and 12. These results indicate that our teams submission score was approximately 0.8 percent higher than the baseline model, with 0.71 for the baseline model and 0.79 for our teams final run with DistilBERT. Based on the submission score and F1 metrics for both the baseline and final models, the results indicate that our final model moderately outperformed the baseline across all key metrics. This suggests that DistilBERT is a competitive solution for multi-label text classification of code comments, when compared to contemporary baseline models.

6 Conclusion

Code comments are crucial for enhancing the interpretability and readability of source code. However, inconsistencies in commenting styles—both within individual projects and across different programming languages—can present significant challenges, as noted in previous studies on comment categorization[Rani et al., 2021][Al-Kaswan et al., 2023]. In this study, we explored the use of a lightweight, pre-trained transformer model, DistilBERT, to perform multi-label classification of code comments in Java, Python, and Pharo. Our results indicate that DistilBERT outperforms the baseline set-Fit model in terms of both classification accuracy and inference speed, particularly when using a learning rate of $5e-5$, a batch size of 4, and 20 epochs, corroborating findings from earlier works that explored machine learning-based comment classification[Pascarella and Bacchelli, 2017][Ochodek et al., 2022].

These findings demonstrate that DistilBERT offers a promising alternative for automating the classification of code comments, contributing to improved software maintainability by reducing the cognitive load on developers and streamlining the code review process, a challenge identified in prior research [Al-Kaswan et al., 2023]. While this approach shows considerable promise, further research is needed to refine the model, particularly in expanding its support for additional programming languages and exploring further hyperparameter optimization. Future work could also focus on leveraging multi-task learning techniques to create a more generalized model that can handle a broader array of code comment classifications across different software engineering tasks, as discussed in the context of related stud-

ies on multi-language code comment classification [Rani et al., 2021]; [Al-Kaswan et al., 2023].

7 Future Work

7.1 Study Limitations

Due to this project being developed for a competition we were limited to the use of a T4 graphics card for producing the final results. To save time, we utilized a NVIDIA RTX 3080 GPU to perform the grid search, which contributed to the lower value of average F1 metric of the final model, compared to its grid search counterpart, within the final notebook. Moving forward, this project can be moved to superior technology, allowing for an increased speed of categorization. This project also required the calculation of weighted composite metric, using the model’s average F1 metric, its average runtime, and the average number of FLOPs over ten iterations, during model evaluation, which significantly increased the time taken to perform hyperparameter grid searches to fine tune the model. This model could be effectively applied for categorization without the need to measure the FLOPs, which would almost half the total run time of the model during training.

Additionally, our grid search capabilities were limited due to storage space. Since we saved each model after training, we often saw the grid search ending training runs early due to the inability to save some models during later parts of the training. Due to this, our grid search data does not contain balanced data to support a ‘best-value’ for the hyperparameter, weight of decay.

If we were given more time, we would remove the code which saves each trained model, which would allow us to test additional hyperparameters, such as the threshold for precision calculations, model optimizers, and activation functions, while testing larger ranges of values for already tested hyperparameters.

7.2 Direction for Future Development

It is possible to reduce the number of models from three down to one by designing a unified model architecture that can handle all tasks the three models currently address. This could be achieved through a multi-task learning approach, where a single model is trained on different objectives simultaneously. While this approach is possible, our experience in building machine learning models for natural language processing is limited, therefore a multi-model approach better fits our skill set.

The area of comment code classification can be further improved through the introduction of more examples and languages into this model. By

building a model that is robust and efficient across most programming languages it would wholistically boost the entirety of the field into a better generalization across diverse coding tasks. This would in turn create a more unified and inclusive ecosystem by reducing the need for language-specific solutions, fostering better understanding, optimization, and generation.

References

- [Al-Kaswan et al., 2023] Al-Kaswan, A., Izadi, M., and Van Deursen, A. (2023). Stacc: Code comment classification using sentencetransformers. In *2023 IEEE/ACM 2nd International Workshop on Natural Language-Based Software Engineering (NLBSE)*, pages 28–31.
- [Biewald, 2020] Biewald, L. (2020). Experiment tracking with weights and biases. Software available from wandb.com.
- [Colavito et al., 2025] Colavito, G., Al-Kaswan, A., Stulova, N., and Rani, P. (2025). The nlse’25 tool competition. In *Proceedings of The 4th International Workshop on Natural Language-based Software Engineering (NLSE’25)*.
- [Jupyter, 2024] Jupyter (2024). Jupyter: Open-source software for interactive computing. Accessed: 2024-12-02.
- [Ochodek et al., 2022] Ochodek, M., Staron, M., Meding, W., and Söder, O. (2022). Automated code review comment classification to improve modern code reviews. In Mendez, D., Wimmer, M., Winkler, D., Biffi, S., and Bergsmann, J., editors, *Software Quality: The Next Big Thing in Software Engineering and Quality*, volume 439 of *Lecture Notes in Business Information Processing*. Springer, Cham.
- [Pascarella and Bacchelli, 2017] Pascarella, L. and Bacchelli, A. (2017). Classifying code comments in java open-source software systems. In *2017 IEEE/ACM 14th International Conference on Mining Software Repositories (MSR)*. IEEE.
- [Python Software Foundation, 2024] Python Software Foundation (2024). Python 3.12.0 release. Accessed: 2024-12-02.
- [PyTorch, 2024] PyTorch (2024). Pytorch: An open source deep learning platform. Accessed: 2024-12-02.
- [Rani et al., 2021] Rani, P., Panichella, S., Leuenberger, M., Di Sorbo, A., and Nierstrasz, O. (2021). How to identify class comment types? a multi-language approach for class comment classification. *Journal of systems and software*, 181:111047.
- [Sanh et al., 2020] Sanh, V., Debut, L., Chaumond, J., and Wolf, T. (2020). Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter.

Appendices

The appendix section provides additional statistical information for our final results and grid search.

Appendix A: DistilBERT Grid Search

Appendix A.1: Metric Results of All Models

<input type="checkbox"/> Name (36 visualized)	eval/f1	eval/loss	eval/precision	eval/recall
pharo_lr-5e-06_epoch-20_batchSize-8_weightsOfDecay-0.01	0.5011	0.22067	0.63754	0.42283
python_lr-5e-06_epoch-20_batchSize-8_weightsOfDecay-0.01	0.5563	0.29667	0.65433	0.49948
java_lr-5e-06_epoch-20_batchSize-8_weightsOfDecay-0.01	0.87321	0.093831	0.89824	0.85414
pharo_lr-5e-05_epoch-20_batchSize-8_weightsOfDecay-0.01	0.7077	0.28732	0.83414	0.65159
python_lr-5e-05_epoch-20_batchSize-8_weightsOfDecay-0.01	0.7515	0.53325	0.7783	0.73044
java_lr-5e-05_epoch-20_batchSize-8_weightsOfDecay-0.01	0.86257	0.096832	0.89581	0.83737
pharo_lr-5e-06_epoch-20_batchSize-4_weightsOfDecay-0.01	0.59312	0.21757	0.89917	0.51216
python_lr-5e-06_epoch-20_batchSize-4_weightsOfDecay-0.01	0.65092	0.33929	0.77102	0.598
java_lr-5e-06_epoch-20_batchSize-4_weightsOfDecay-0.01	0.88382	0.11888	0.89727	0.87315
pharo_lr-5e-05_epoch-20_batchSize-4_weightsOfDecay-0.01	0.71032	0.32038	0.80908	0.65878
python_lr-5e-05_epoch-20_batchSize-4_weightsOfDecay-0.01	0.74356	0.61761	0.7598	0.73332
java_lr-5e-05_epoch-20_batchSize-4_weightsOfDecay-0.01	0.86748	0.1266	0.89387	0.84671
pharo_lr-5e-06_epoch-15_batchSize-8_weightsOfDecay-0.01	0.44419	0.22946	0.64596	0.35009
python_lr-5e-06_epoch-15_batchSize-8_weightsOfDecay-0.01	0.52292	0.31167	0.643	0.47052
java_lr-5e-06_epoch-15_batchSize-8_weightsOfDecay-0.01	0.85785	0.086676	0.91101	0.81899
pharo_lr-5e-05_epoch-15_batchSize-8_weightsOfDecay-0.01	0.68901	0.25304	0.82501	0.62772
python_lr-5e-05_epoch-15_batchSize-8_weightsOfDecay-0.01	0.72962	0.52394	0.74983	0.71415
java_lr-5e-05_epoch-15_batchSize-8_weightsOfDecay-0.01	0.87489	0.15034	0.89315	0.85935
pharo_lr-5e-06_epoch-15_batchSize-4_weightsOfDecay-0.01	0.52783	0.19421	0.6517	0.44877
python_lr-5e-06_epoch-15_batchSize-4_weightsOfDecay-0.01	0.59679	0.31253	0.81511	0.54031
java_lr-5e-06_epoch-15_batchSize-4_weightsOfDecay-0.01	0.85328	0.10867	0.87843	0.83406
pharo_lr-5e-05_epoch-15_batchSize-4_weightsOfDecay-0.01	0.67531	0.30258	0.85305	0.61509
python_lr-5e-05_epoch-15_batchSize-4_weightsOfDecay-0.01	0.7146	0.61431	0.71573	0.71831
java_lr-5e-05_epoch-15_batchSize-4_weightsOfDecay-0.01	0.85646	0.16609	0.89385	0.82759
pharo_lr-5e-06_epoch-10_batchSize-8_weightsOfDecay-0.01	0.1767	0.27836	0.41782	0.15081
python_lr-5e-06_epoch-10_batchSize-8_weightsOfDecay-0.01	0.42356	0.33741	0.63049	0.35313
java_lr-5e-06_epoch-10_batchSize-8_weightsOfDecay-0.01	0.81811	0.089402	0.90663	0.76561
pharo_lr-5e-05_epoch-10_batchSize-8_weightsOfDecay-0.01	0.66468	0.23713	0.85389	0.58661
python_lr-5e-05_epoch-10_batchSize-8_weightsOfDecay-0.01	0.72025	0.3804	0.75856	0.68804
java_lr-5e-05_epoch-10_batchSize-8_weightsOfDecay-0.01	0.86127	0.1082	0.87163	0.85549
pharo_lr-5e-06_epoch-10_batchSize-4_weightsOfDecay-0.01	0.34644	0.24076	0.65786	0.27564
python_lr-5e-06_epoch-10_batchSize-4_weightsOfDecay-0.01	0.51859	0.31344	0.66243	0.46079
java_lr-5e-06_epoch-10_batchSize-4_weightsOfDecay-0.01	0.84718	0.09818	0.88142	0.81973
pharo_lr-5e-05_epoch-10_batchSize-4_weightsOfDecay-0.01	0.6761	0.28469	0.80595	0.61887
python_lr-5e-05_epoch-10_batchSize-4_weightsOfDecay-0.01	0.7499	0.45009	0.77422	0.73597
java_lr-5e-05_epoch-10_batchSize-4_weightsOfDecay-0.01	0.86145	0.12996	0.87998	0.84563

Figure 2: Overall grid search results showing the F1, Loss, Precision, and Recall metrics for each model trained.

Appendix A.2: Average Metric Results for Learning Rates

<input type="checkbox"/> <input checked="" type="radio"/> Name (36 visualized)		eval/f1	eval/loss	eval/precision	eval/recall
• <input checked="" type="radio"/> <input checked="" type="radio"/> learning_rate: 0.000005	18	0.61066	0.21598	0.74775	0.55823
• <input checked="" type="radio"/> <input checked="" type="radio"/> learning_rate: 0.00005	18	0.76204	0.31015	0.82477	0.73061

Figure 3: Grid search results for learning rates, which shows the F1, Loss, Precision, and Recall metrics for each model trained.

Appendix A.3: Average Metric Results for Batch Sizes

<input type="checkbox"/> <input checked="" type="radio"/> Name (36 visualized)		eval/f1	eval/loss	eval/precision	eval/recall
• <input checked="" type="radio"/> <input checked="" type="radio"/> batch_size: 8	18	0.66863	0.25081	0.76696	0.62424
• <input checked="" type="radio"/> <input checked="" type="radio"/> batch_size: 4	18	0.70406	0.27532	0.80555	0.6646

Figure 4: Grid search results for batch sizes, which shows the F1, Loss, Precision, and Recall metrics for each model trained.

Appendix A.4: Average Metric Results for Epochs

<input type="checkbox"/> <input checked="" type="radio"/> Name (36 visualized)		eval/f1	eval/loss	eval/precision	eval/recall
• <input checked="" type="radio"/> <input checked="" type="radio"/> epochs: 20	12	0.72513	0.27241	0.81072	0.68483
• <input checked="" type="radio"/> <input checked="" type="radio"/> epochs: 15	12	0.69523	0.27113	0.78965	0.65208
• <input checked="" type="radio"/> <input checked="" type="radio"/> epochs: 10	12	0.63869	0.24567	0.75841	0.59636

Figure 5: Grid search results for epochs, which shows the average F1, Loss, Precision, and Recall metrics for each epoch group.

Appendix A.5: Average F1 metrics by Architecture

```
'lr-5e-05_epoch-10_batchSize-4_weightsOfDecay-0.01': 0.763809402132499,
'lr-5e-06_epoch-10_batchSize-4_weightsOfDecay-0.01': 0.5762243510409606,
'lr-5e-05_epoch-10_batchSize-8_weightsOfDecay-0.01': 0.7517311590795167,
'lr-5e-06_epoch-10_batchSize-8_weightsOfDecay-0.01': 0.47797133051203045,
'lr-5e-05_epoch-15_batchSize-4_weightsOfDecay-0.01': 0.7523858683081371,
'lr-5e-06_epoch-15_batchSize-4_weightsOfDecay-0.01': 0.6658800533871522,
'lr-5e-05_epoch-15_batchSize-8_weightsOfDecay-0.01': 0.7681797815506027,
'lr-5e-06_epoch-15_batchSize-8_weightsOfDecay-0.01': 0.6173060758906344,
'lr-5e-05_epoch-20_batchSize-4_weightsOfDecay-0.01': 0.7769692605356904,
'lr-5e-06_epoch-20_batchSize-4_weightsOfDecay-0.01': 0.7154331131665084,
'lr-5e-05_epoch-20_batchSize-8_weightsOfDecay-0.01': 0.7762857137735598,
'lr-5e-06_epoch-20_batchSize-8_weightsOfDecay-0.01': 0.6527187327721913,
```

Figure 6: The best performing model of the grid search, and its F1 metric.

Appendix B: DistilBERT Final Model

Appendix B.1: Final Model Metrics by Language

```
Total Flops: 326.22309888000007
Total Time: 0.35062170028686523
Average Flops: 32.622309888000004
Average Time: 0.03506217002868653

Average F1 for java: 0.7241158002383561
Average Precision for java: 0.7614856631881299
Average Recall for java: 0.7034891854723387

Average F1 for python: 0.5746734587251828
Average Precision for python: 0.6035773920556529
Average Recall for python: 0.5558594537391655

Average F1 for pharo: 0.6555011011496797
Average Precision for pharo: 0.6958893902388541
Average Recall for pharo: 0.6319236102312124

Average F1 for across all languages: 0.6595097685969562
Average Precision for across all languages: 0.6957638070666924
Average Recall for across all languages: 0.6382729915063518
```

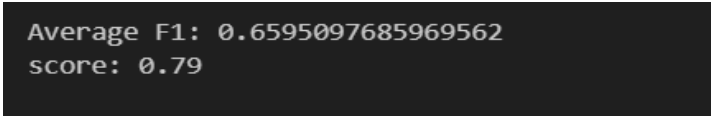
Figure 7: The final model’s F1, Precision, and Recall metrics for all languages.

Appendix B.2: Final Model Metrics by Language Labels

	lan	cat	precision	recall	f1
0	java	summary	0.867925	0.928251	0.897075
1	java	Ownership	1.000000	1.000000	1.000000
2	java	Expand	0.339130	0.382353	0.359447
3	java	usage	0.935401	0.839907	0.885086
4	java	Pointer	0.862944	0.923913	0.892388
5	java	deprecation	0.900000	0.600000	0.720000
6	java	rational	0.425000	0.250000	0.314815
7	python	Usage	0.720721	0.661157	0.689655
8	python	Parameters	0.803571	0.703125	0.750000
9	python	DevelopmentNotes	0.309091	0.414634	0.354167
10	python	Expand	0.543478	0.390625	0.454545
11	python	Summary	0.641026	0.609756	0.625000
12	pharo	Keyimplementationpoints	0.758621	0.511628	0.611111
13	pharo	Example	0.942857	0.831933	0.883929
14	pharo	Responsibilities	0.600000	0.692308	0.642857
15	pharo	Classreferences	0.666667	0.500000	0.571429
16	pharo	Intent	0.866667	0.866667	0.866667
17	pharo	Keymessages	0.607843	0.720930	0.659574
18	pharo	Collaborators	0.428571	0.300000	0.352941

Figure 8: The final model’s F1, Precision, and Recall metrics for each of the languages’ labels.

Appendix B.3: Final Model Submission Score

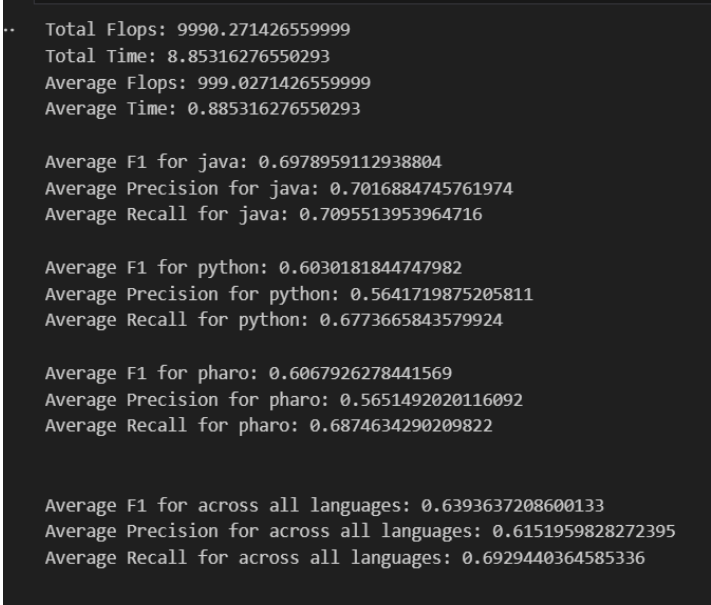


```
Average F1: 0.6595097685969562
score: 0.79
```

Figure 9: The final model’s average F1 metric and submission score.

Appendix C: SetFit Baseline Model

Appendix C.1: Baseline Metrics by Language



```
.. Total Flops: 9990.271426559999
   Total Time: 8.85316276550293
   Average Flops: 999.0271426559999
   Average Time: 0.885316276550293

   Average F1 for java: 0.6978959112938804
   Average Precision for java: 0.7016884745761974
   Average Recall for java: 0.7095513953964716

   Average F1 for python: 0.6030181844747982
   Average Precision for python: 0.5641719875205811
   Average Recall for python: 0.6773665843579924

   Average F1 for pharo: 0.6067926278441569
   Average Precision for pharo: 0.5651492020116092
   Average Recall for pharo: 0.6874634290209822

   Average F1 for across all languages: 0.6393637208600133
   Average Precision for across all languages: 0.6151959828272395
   Average Recall for across all languages: 0.6929440364585336
```

Figure 10: The SetFit baseline model’s F1, Precision, and Recall metrics for all languages.

Appendix C.2: Baseline Metrics by Language Labels

	lan	cat	precision	recall	f1
0	java	summary	0.878394	0.834081	0.855664
1	java	Ownership	1.000000	1.000000	1.000000
2	java	Expand	0.323529	0.431373	0.369748
3	java	usage	0.925065	0.830626	0.875306
4	java	Pointer	0.790179	0.961957	0.867647
5	java	deprecation	0.818182	0.600000	0.692308
6	java	rational	0.176471	0.308824	0.224599
7	python	Usage	0.700787	0.735537	0.717742
8	python	Parameters	0.793893	0.812500	0.803089
9	python	DevelopmentNotes	0.243902	0.487805	0.325203
10	python	Expand	0.433628	0.765625	0.553672
11	python	Summary	0.648649	0.585366	0.615385
12	pharo	Keyimplementationpoints	0.622222	0.651163	0.636364
13	pharo	Example	0.878049	0.907563	0.892562
14	pharo	Responsibilities	0.596154	0.596154	0.596154
15	pharo	Classreferences	0.200000	0.500000	0.285714
16	pharo	Intent	0.718750	0.766667	0.741935
17	pharo	Keymessages	0.680000	0.790698	0.731183
18	pharo	Collaborators	0.260870	0.600000	0.363636

Figure 11: The SetFit baseline model’s F1, Precision, and Recall metrics for each of the languages’ labels.

Appendix C.3: Baseline Submission Score

...	0.71
-----	------

Figure 12: The SetFit baseline model’s average F1 metric and submission score.

Appendix D: Submission Score Formula

$$\begin{aligned}
\text{submission_score}(\text{model}) = & 0.60 \times \text{avg. } F_1 \\
& + 0.2 \times \frac{(\text{max_avg_runtime} - \text{measured_avg_runtime})}{\text{max_avg_runtime}} \\
& + 0.2 \times \frac{(\text{max_avg_GFLOPS} - \text{measured_avg_GFLOPS})}{\text{max_avg_GFLOPS}}
\end{aligned} \tag{4}$$