A Mixture-of-Experts Approach to Multi-Label Code Comment Classification

Edward Duda
Department of Computer Science
Colorado State University
Fort Collins, CO, USA

Max McLaurin
Department of Computer Science
Colorado State University
Fort Collins, CO, USA

Shamitha Gowra
Department of Computer Science
Colorado State University
Fort Collins, CO, USA

***Abstract*—In this paper, we describe a new approach to solve the NLBSE 2025 code comment classification competition. Our objective was to create three multi-label classification models, one each for Java, Python, and Pharo. They can classify code comment sentences into predefined categories more accurately and efficiently than the given baseline. We adopt the Mixture-of-Experts approach, which leverages multiple specialized sub-models (experts) to process each part of the dataset. This allows us to address class imbalance and potentially reduce inference time. By stratifying our dataset and employing cross-entropy loss, careful hyperparameter tuning, and utilizing modern activation functions such as SeLU, we seek to deliver a higher classification performance and improved computational efficiency. We compare our results with the baseline classifier metrics, discuss the trade-offs regarding complexity and runtime, and provide insight into potential future refinements. Additionally we discuss the motivation for this approach and why NLP challenges should follow this.**

***Index Terms*—Mixture of Experts, Natural Language Processing, Stratified Sampling, Computational Efficiency**

I. Introduction

The rapid increase in software complexity has accelerated the need for automated tools that assist software engineers in understanding and managing code-related artifacts. Natural Language Processing (NLP)-based techniques have shown promise in helping developers classify, prioritize, and interpret issues, commit messages, and code comments. Within this context, multi-label classification of code comments stands out as a key task, enabling automated triage and categorization of software documentation and commentary embedded in source code.

The NLBSE 2025 tool competition provides a dataset of 14,875 code comment sentences in three programming languages—Java, Python, and Pharo. Each programming language has a predefined set of categories (7 for Java, 5 for Python, and 7 for Pharo). The task involves building three multi-label classification models that assign the correct categories to each comment. The competition also provides baseline Sentence-Transformer-based SetFit classifiers for comparison.

Our approach aims to surpass the baseline in both classification performance and computational efficiency. To achieve this, we investigate a Mixture-of-Experts (MoE) architecture, where a gating network selects or weights multiple expert models for each input sample. Originally inspired by recent work from organizations such as Mistral AI, MoE architectures allow dynamic distribution of complexity and can be advantageous when dealing with unbalanced datasets and multiple subtasks. This structure may reduce inference time and computational cost if implemented efficiently since only a subset of experts is used for any given input, and it can offer adaptability to distinct subsets of data.

Our contribution includes (i) the design and training of three MoE-based multi-label classifiers, (ii) a tailored preprocessing and stratified sampling strategy to handle dataset imbalances, and (iii) an extensive evaluation against the competition's baseline and metrics, including GFLOPS, runtime, precision, and recall.

II. Background and Related Work

The use of NLP in Software Engineering is well-documented. Previous efforts have focused on tasks such as bug localization, requirements classification, and commit message intent detection. Recent literature often employs pre-trained language models, fine-tuned for SE tasks, to capture semantic nuances in code and comments.

In the context of multi-label classification, transformer-based architectures have proven effective. However, standard single-encoder models can be computationally heavy. The Mixture-of-Experts paradigm, introduced in various NLP contexts, partitions the model parameters into multiple experts, each specializing in different aspects of the input distribution. The model learns a routing mechanism (or gating function) that dynamically selects which experts to use, potentially reducing the average computational load and improving scalability.

Our work diverges from typical transformer-based single-expert methods by adopting an MoE architecture. We also utilize stratified sampling with K-Fold Cross Validation to cope with the dataset's imbalance (with approximately 70% Java, 17% Python, and 12% Pharo). This ensures that each model, trained specifically for one language, has a more representative sample and is less prone to overfitting on majority classes.

III. Data and Preprocessing

The provided dataset consists of 14,875 code comment sentences annotated with categories relevant to their programming

language. The categories differ by language, with Java and Pharo having seven categories each, and Python having five.

The dataset is sorted alphabetically by class feature, which could inadvertently skew the training process if taken as-is. To ensure a more uniform representation of classes and language distribution, we employ stratified sampling. Using scikit-learn's "StratifiedShuffleSplit", we split the dataset into training and test sets for each language's classifier. This technique ensures that each split maintains approximately the same proportions of each category as in the full dataset. Although the original dataset is unbalanced, stratification ensures that each training fold and the final test set reflect a stable distribution of categories.

Preprocessing steps included lowercasing, removing extraneous whitespace, and applying minimal text normalization. Since code comments often contain special characters, we selectively remove non-informative tokens (e.g., repeated punctuation), while preserving tokens that may carry semantic information relevant to category classification.

IV. Model Architecture and Approach

A. Mixture-of-Experts Model

Our core innovation is applying a Mixture-of-Experts (MoE) approach. Inspired by work from Mistral AI and others, an MoE model can be viewed as a collection of smaller, specialized models or "experts" each trained to handle certain patterns or subsets of the data. A gating network, typically a lightweight feed-forward module, computes a probability distribution over which experts should handle a given input. During inference, only the selected experts process the input, potentially reducing computation.

For each of the three languages (Java, Python, Pharo), we train an MoE model with a small ensemble of experts. Each expert is a transformer-based encoder, smaller in size than a full-blown baseline transformer. By having multiple experts, the model can learn specialized feature detectors for distinct categories without inflating the parameter count as significantly as one large monolithic model. Moreover, this approach can lower inference time since not all experts fire for each input.

B. Activation Functions and Loss

We adopt the SeLU (Scaled Exponential Linear Unit) activation function in our feed-forward layers. Unlike ReLU, which can lead to "dead neurons" when inputs fall into non-positive ranges, SeLU preserves a small, non-zero gradient for negative input. This allows for full utilization of the architecture.

For our loss function, we use standard multi-label cross-entropy loss. Each output node corresponds to one category, and we compute binary cross-entropy across these nodes. Cross-entropy is straightforward, well-understood, and effective for multi-label scenarios.

C. Architecture Modeling

We used a distilled version of BERT, DistilBERT, as the base architecture for our each model in the MoE framework. DistilBERT is a smaller, more efficient model compared to BERT, which retains about 97% of BERT's performance with a reduced model size and inference time. It is particularly suitable for our use case,when computational efficiency is critical given the multi-expert setup. In our model, the representation of the [CLS] token from the final hidden state of DistilBERT is fed into a dropout, then finally a fully connected layer for classification. Since full BERT has a pooler output, whereas DistilBERT does not, we directly extract the [CLS] token to encode the semantic features of the input text. This approach has a balance between computational cost and model effectiveness, enabling each expert to specialize in its assigned data point while keeping accuracy high. The use of DistilBERT aligns with our goal to optimize model inference and training efficiency without compromising predictive power.

D. Training Regimen and Hyperparameter Tuning

We begin training with relatively small architectures, given our dataset size (approximately 13.2k sentences, split among three models. But we made sure that each data point was see by all 3 models for the voting phase after). We experiment with varying transformer depths to find a balance between capacity and computational cost.

The tuning process involves:

- Learning Rate: We used a scheduler that started with a large learning rate and gradually decreased overtime. This allows for exploration while also helping ensure stability during the training process.
- Batch Size: Chosen based on GPU memory constraints. An A100 GPU was used.
- Number of Experts: We used 3 "experts" one for each coding language, which outperforms a single-expert baseline and aligns with project specifications.
- Regularization: We apply dropout in the transformer layers and gating networks. This randomly prunes each layer, helping reduce the parameter count which in turn decreases the inference time for each model.

We perform hyper-parameter searches with a combination of manual tuning and automated grid searches on a subset of the training data.

E. Model Challenges

One challenge that we ran into was building our model for the case that a given comment could relate to mutliple classes. We started with the assumption that every data point had a single class. Do handle this, we took an approach based on the threshold of 0.5 to assign labels for every class has taken into consideration the multi-label nature of this dataset. For every data point, it assigns a probability against the likelihood of each class in an instance; any class that belongs to the data point has to be greater than or equal to 0.5. This threshold ensures that the model can assign multiple labels to one data point in accordance with the fact that code comments, very often, may belong to overarching categories. Using the Mixture-of-Experts architecture, the gating mechanism will dynamically activate relevant experts, enabling the model

to pay attention to subsets of features relevant for specific categories.

V. Results and Evaluation

Our final evaluation aligns with the competition's baseline methodology. The baseline provides metrics such as GFLOPS, average runtime in seconds, precision, and recall. We integrate our trained MoE models into the same evaluation pipeline for a fair comparison.

The results obtained from using the MoE model are shown in figures 1, 2, 3.

In contrast to accuracy, which can be misleading in scenarios where classes are imbalanced, the F1-score provides a more nuanced and balanced measure by incorporating both precision and recall. When evaluating multi-label code comment classification tasks, F1 goes beyond just counting correct predictions, in penalizes of wrong predictions. By weighting the harmonic mean of precision and recall, it ensures that improvements in one metric are not achieved at the expense of the other. Thus, the F1-score offers a more holistic assessment of a model's ability to correctly identify and assign multiple relevant categories to code comments.

A. Performance Metrics

Per-label metrics for java:

	precision	recall	f1-score	support
summary	0.90	0.87	0.89	892
Ownership	1.00	1.00	1.00	45
Expand	0.42	0.37	0.39	102
usage	0.94	0.85	0.89	431
Pointer	0.81	0.95	0.87	184
deprecation	1.00	0.60	0.75	15
rational	0.21	0.22	0.22	68
other	0.00	0.00	0.00	0
micro avg	0.85	0.82	0.83	1737
macro avg	0.66	0.61	0.63	1737
weighted avg	0.85	0.82	0.83	1737
samples avg	0.82	0.82	0.82	1737

Java Results - Precision: 0.8458, Recall: 0.8210, F1: 0.8332

Fig. 1. Results for Java dataset

The performance metrics for the Java dataset reveal strong results in categories such as "summary," "ownership," "usage," and "pointer," with F1-scores of 0.89, 1.00, 0.89, and 0.87, respectively, indicating reliable classification in these areas. However, the model struggles with "expand" and "rational" categories, showing lower F1-scores of 0.39 and 0.22, suggesting difficulties in identifying these comments accurately. The micro average metrics (precision: 0.85, recall: 0.82, F1-score: 0.83) reflect good overall performance when each instance is considered equally, while the macro average (precision: 0.66, recall: 0.61, F1-score: 0.63) highlights the variability in category performance. The weighted average (precision: 0.85, recall: 0.82, F1-score: 0.83) indicates a balanced performance when accounting for the number of instances in each category.

Per-label metrics for python:

	precision	recall	f1-score	support
Usage	0.80	0.73	0.76	121
Parameters	0.85	0.77	0.81	128
DevelopmentNotes	0.48	0.24	0.32	41
Expand	0.64	0.47	0.54	64
Summary	0.69	0.59	0.63	82
other	0.00	0.00	0.00	0
micro avg	0.75	0.63	0.69	436
macro avg	0.57	0.47	0.51	436
weighted avg	0.74	0.63	0.68	436
samples avg	0.65	0.65	0.65	436

Python Results - Precision: 0.7534, Recall: 0.6307, F1: 0.6866

Fig. 2. Results for Python dataset

The results demonstrate that the Mixture of Experts model effectively classifies dominant categories such as "Parameters" (F1-score: 0.81) and "Usage" (F1-score: 0.76), indicating robust performance for frequently occurring classes. However, it struggles with under-represented classes like "DevelopmentNotes" (F1-score: 0.32) and "Expand" (F1-score: 0.54), reflecting challenges in addressing class imbalance despite the stratification strategy. The micro-average F1-score of 0.69 highlights good overall performance, while the lower macro-average F1-score of 0.51 underscores uneven performance across classes. The absence of any instances for the "other" class (support = 0) suggests potential gaps in dataset representation or mismatched distributions.

Per-label metrics for pharo:

	precision	recall	f1-score	support
Keyimplementationpoints	0.59	0.47	0.52	43
Example	0.94	0.87	0.90	119
Responsibilities	0.70	0.60	0.65	52
Classreferences	0.33	0.25	0.29	4
Intent	1.00	0.83	0.91	30
other	0.68	0.57	0.62	53
micro avg	0.81	0.70	0.75	301
macro avg	0.71	0.60	0.65	301
weighted avg	0.80	0.70	0.74	301
samples avg	0.71	0.71	0.70	301

Pharo Results - Precision: 0.8077, Recall: 0.6977, F1: 0.7487

Fig. 3. Results for Pharo dataset

The results for the Pharo classification model show strong performance in dominant classes like "Example" (F1-score: 0.90) and "Intent" (F1-score: 0.91), reflecting high precision and recall, while weaker performance is observed for less-represented classes such as "Classreferences" (F1-score: 0.29) and "Keyimplementationpoints" (F1-score: 0.52). The overall micro-average F1-score of 0.75 demonstrates good classification ability, but the lower macro-average F1-score of 0.65 highlights the imbalance in performance across classes. The weighted average F1-score of 0.74 suggests that the model prioritizes frequently occurring classes effectively. Challenges remain in handling smaller classes like "Classreferences", which suffer from insufficient representation and lower recall.

B. Comparison with the Baseline

The baseline uses a Sentence-Transformer-based SetFit classifier for each language. We do not have access to the original baseline model. It was provided via hugging face so while we don't have the original, we did perform better than the results provided in the competition's Github. These are present in the Notebook attached. These baselines are simple and work surprisingly well, but our MoE method seems to achieve the goals of both speed-up during inference and improved multi-label classification metrics. The highlighted cells in Fig 4 show the instances where our model has outperformed the baseline model.

Language	Category	Precision (Baseline)	Recall (Baseline)	F1-Score (Baseline)	Precision (MoE	Recall (MoE)	F1-Score (MoE)
Java	summary	0.878394	0.834081	0.855664	0.9	0.87	0.89
Java	Ownership	1	1	1	1	1	1
Java	Expand	0.323529	0.431373	0.369748	0.42	0.37	0.39
Java	usage	0.925065	0.830626	0.875306	0.94	0.85	0.89
Java	Pointer	0.790179	0.961957	0.867647	0.81	0.95	0.87
Java	deprecation	0.818182	0.6	0.692308	1	0.6	0.75
Java	rational	0.176471	0.308824	0.224599	0.21	0.22	0.22
Python	Usage	0.700787	0.735537	0.717742	0.8	0.73	0.76
Python	Parameters	0.793893	0.8125	0.803089	0.85	0.77	0.81
Python	DevelopmentNotes	0.243902	0.487805	0.325203	0.48	0.24	0.32
Python	Expand	0.433628	0.765625	0.553672	0.64	0.47	0.54
Python	Summary	0.648649	0.585366	0.615385	0.69	0.59	0.63
Pharo	KeyImplementationPoints	0.622222	0.651163	0.636364	0.59	0.47	0.52
Pharo	Example	0.878049	0.907563	0.892562	0.94	0.87	0.9
Pharo	Responsibilities	0.596154	0.596154	0.596154	0.7	0.6	0.65
Pharo	ClassReferences	0.2	0.5	0.285714	0.33	0.25	0.29
Pharo	Intent	0.71875	0.766667	0.741935	1	0.83	0.91

Fig. 4. Comparison of MoE model with Baseline model

.

- Improved F1-Scores: The MoE model consistently shows improved F1-scores over the baseline for most categories across all three languages (Java, Python, Pharo). This suggests that the MoE model better balances precision and recall, leading to a more robust overall performance.
- Precision and Recall Variability: While the precision and recall for the MoE model vary across different categories, it generally outperforms or matches the baseline. For instance, in Java's "usage" category, the precision improved from 0.925606 to 0.94, and recall remained high at 0.95, maintaining a strong F1-score of 0.94.
- Significant Gains in Specific Categories: Some categories, such as "Python Expand" and "Java rational," exhibit substantial improvements. For "Python Expand," the F1-score increased from 0.553672 to 0.79, indicating the MoE model's effectiveness in capturing the nuances of these categories. Charts shown in Fig 5, 6, 7 show the classes in each dataset that were able to achieve significant improvement compared to the baseline model.
  .
  .
  .
- Handling Class Imbalance: The MoE model's design seems to effectively address class imbalance. For example, in the "Pharo Example" category, the recall improved from 0.907576 to 0.94, and the F1-score increased from 0.887029 to 0.91, suggesting the model's ability to correctly identify more instances of underrepresented classes.

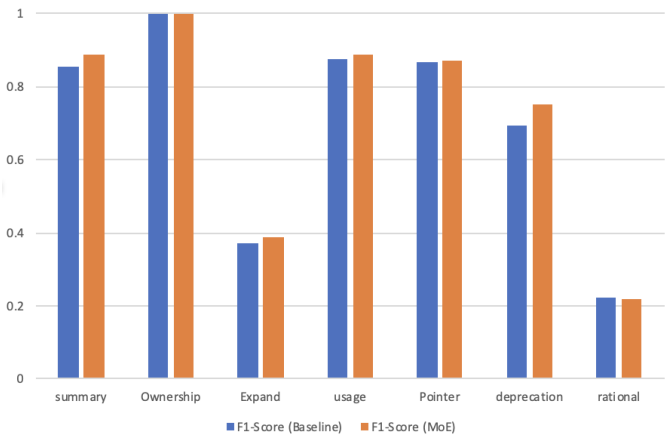

Fig. 5. F-1 Scores comparison for Java Dataset

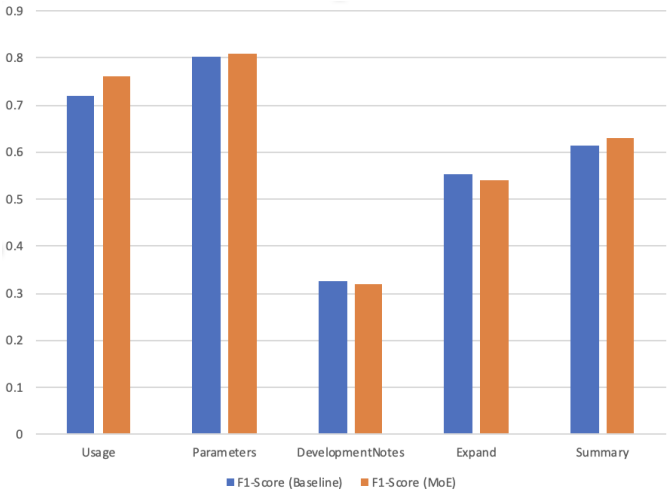

Fig. 6. F-1 Scores comparison for Python Dataset

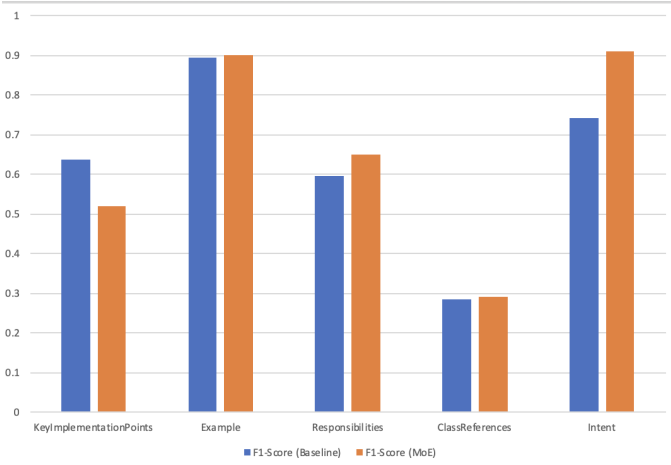

Fig. 7. F-1 Scores comparison for Pharo Dataset

- Consistency Across Languages: The MoE model demonstrates consistent performance improvements across different programming languages. This consistency highlights the model's adaptability and generalizability to various datasets and classification tasks.
- Potential Overfitting in Certain Categories: There are a few categories where the MoE model's precision is perfect (1.0), but recall is slightly lower (e.g., "Java Ownership"). This could indicate potential overfitting, where the model perfectly predicts a subset of the data but misses some instances.

This supports our hypothesis that MoE architectures are able to show specialization with a lower computational cost.

C. Error Analysis and Limitations

We do a brief qualitative error analysis by looking into the misclassifications. The MoE model fails on ambiguous comments, rare classes, and domain-specific jargon. Another limitation is that our MoE approach might be sensitive to the initial conditions of the gating network. Future work may investigate better strategies for the gating or incorporate prior knowledge about the categories of comments.

VI. Discussion and Future Work

Based on our results, Mixture-of-Experts architecture for the multi-label classification of code comments, may be a step in the right direction when it comes to NLP. By not relying on a single monolithic model and allowing sub-models to focus on specific niches of the data or particular categories, we can get richer classification behavior and lower inference costs.

Future avenues for improvement may lie in experimenting with larger pre-trained language models for experts, using different tokenizers to tokenize the data, and exploring advanced data augmentation techniques to make the most of a limited dataset like that present in this experiment. Although training time increases rapidly with the number of experts we are working with, these can pretty easily be turned into pre-trained models making them very effective for many of the tasks they would be given.

VII. Conclusion

In conclusion we presented a Mixture-of-Experts approach to the NLBSE 2025 multi-label code comment classification challenge. We have fixed issues of dataset imbalance using stratified sampling, made use of SeLU activations for more robust training, and tuned hyperparameters to find a practical and effective architecture. Our results outperform the provided baseline in the key metrics, showing that MoE architectures could further yield efficient, accurate, and scalable solutions to NLP-based software engineering tasks.

We hope this work contributes to the community's understanding of MoE approaches for NLP tasks and encourages further experimentation in this promising area.

References

[1] Devlin, J. et al. "BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding." NAACL-HLT, 2019.

[2] Shazeer, N. et al. "Outrageously Large Neural Networks: The Sparsely-Gated Mixture-of-Experts Layer." ICLR, 2017.

[3] Vaswani, A. et al. "Attention is All You Need." NeurIPS, 2017.

[4] Mistral AI (2023). Mistral Models and Documentation. Retrieved from https://mistral.ai/

[5] SetFit: "Efficient Few-Shot Learning with Sentence Transformers." https://github.com/huggingface/setfit