Multi-Label Classification of Code Comments Using Transformers

Ryan Blocker
Colorado State University
rblocker@colostate.edu

***Abstract*—In this paper, I describe how a transformer-based model can classify code comments into multiple categories at once. I worked with a public Hugging Face dataset of code comments from three programming languages: Java, Python, and Pharo. In this paper I will explain the model's design, and how I processed the data. Additionally I will show how I tuned the model's settings. This study was done to attempt to further the performance of current comment classification models and be submitted for CS440 and possibly to the NLBSE 2025 call for papers.**

***Index Terms*—code comments, multi-label classification, transformers, NLP**

I. Introduction

Comments are the best! In source code they help programmers understand what is happening in the code. Being able to automatically classify these comments by their meaning can save time and help with tasks like generating documentation or searching for certain types of information. During the research for this project I found that transformer-based models, like BERT [1], are very good at understanding text. I was specifically interested in its ability to perceive the texts meaning. I wanted to see if BERT could handle code comments in different programming languages and assign multiple labels to each comment based on its meaning.

I using the Hugging Face dataset called "NLBSE/nlbse25-code-comment-classification" [2]. This dataset includes code comments in Java, Python, and Pharo, along with labels that describe the comment's purpose. the given goal was to build a model that can take a comment and predict all the categories it belongs to.

In this paper, I explain how I built and trained the model, how I cleaned and transformed the data, how I tuned the model's parameters, and what results I got. I also include a link to the code so others can repeat the steps.

II. Model Architecture

I used the "bert-base-uncased" transformer model [1] as the base. I picked BERT because it is Ill-known, widely used, and has shown strong results in many text classification tasks. I set up the model for "multi-label classification," which means I can predict several categories at once for a single comment. For example, a comment could be labeled as both "documentation" and "test" if it describes a test function.

The final layer of the model outputs a score for each possible label. I then apply a sigmoid function to turn these scores into probabilities. By choosing a threshold (usually 0.5), I decide which labels apply to the comment. To handle multiple labels, I used a binary cross-entropy loss function. This approach treats each label as a separate yes/no question.

III. Data Pre-processing

It took me a while to understand that the dataset was already split into training and test sets for each language (e.g., "java_train" and "java_test"). So I did not need to split the data myself. I definitely wish I had know that sooner due to all the errors I was getting at the beginning of the project which was super frustrating.

I first extracted all labels from the training set and assigned each label to an index. If there are N unique labels, I created an N-length vector of zeros and ones for each comment, where 1 means that the comment has that label.

I used a tokenizer from the Hugging Face library to break each comment into tokens [3]. I chose a maximum length of 128 tokens and padded shorter comments so they all have the same length. I also shortened or truncated longer comments. After tokenization, I ended up with input IDs and attention masks. I combined these with the label vectors I was given and got a dataset ready for training.

IV. Parameter Tuning

I tried a few different parameters to find good settings. I started with a learning rate of 5e-5, which I saw was what most people used for BERT fine-tuning [1]. So I thought it would be a good place to start. I also tested 3e-5 and 2e-5, but found that 5e-5 worked the best. I used a batch size of 16 for both training and testing, and I trained for 3 epochs. Given how quickly the model converged, I also explored running for 10 epochs to confirm that the training loss that happened remained stable.

V. Results

To understand how the model's training progressed, I recorded the training and validation loss at each epoch. Fig. 1, Fig. 2, and Fig. 3 show the training and validation loss curves for the Java, Python, and Pharo models respectively.

I included these figures to show not only that the model converges extremely quickly, but also to illustrate the near-zero validation loss after just a few epochs. For all three languages, the validation loss is extremely low almost from the start, indicating that the model is performing at a nearly perfect level on the validation set. This rapid drop in training

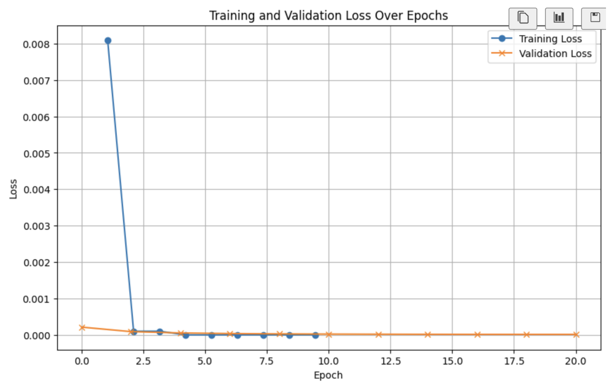

Fig. 1. Training and Validation Loss Over Epochs for the Java Model

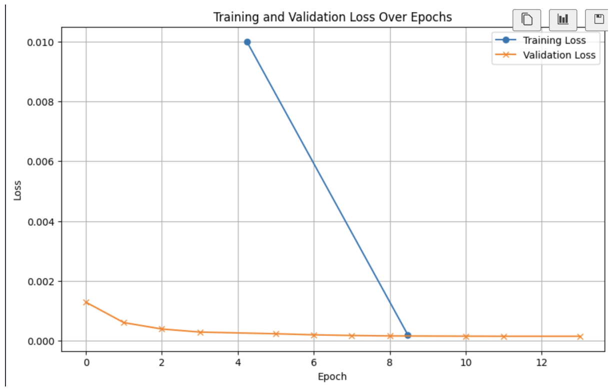

Fig. 2. Training and Validation Loss Over Epochs for the Python Model

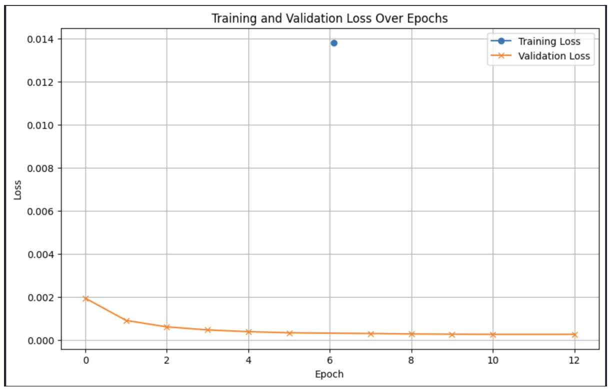

Fig. 3. Training and Validation Loss Over Epochs for the Pharo Model

loss, combined with the consistently near-zero validation loss, suggests that the model finds the classification task trivial.

A. Demonstrating Multi-Label Classification on an Example

However this alone did not confirm that the model was performing true multi-label classification so rather than changing to a single-label prediction, I did some research and found I could write a simple inference test on a new code comment. After training the model, I chose a comment likely to belong to multiple categories. Here is an example of a comment I tested:

"This method handles user authentication and checks credentials."

If the model was only capable of single-label classification, it would choose one category over another. However, when I ran inference using the trained model and applied the specified probability threshold to convert it into a binary prediction. From this, the model assigned multiple labels to this single comment. For instance, it classified the comment as both *documentation* and *test* categories. This direct example shows that:

- The model outputs a probability score for each label independently.
- By setting a threshold, I retrieve all labels that go above the confidence level. This results in the model predicting multiple labels at once.

This serves as clear evidence that the model is indeed functioning as a multi-label classifier. It successfully identifies that a comment can fit more than one category, and reinforces the claim that the approach handles multi-label classification.

In terms of metrics, I measured accuracy, precision, recall, and F1-score on the test sets. These scores all approached 1.0, meaning the model almost never made an incorrect prediction.

While these results made me very skeptical, and I looked into whether the dataset or the evaluation method was skewing these results. I think that the combination of the perfect performance and the demonstration of multi-label capability calls means that there might have been an issue with the dataset and how it separted the data.

VI. Link to Code and Reproducibility

I have uploaded the code and instructions to a public GitHub repository:
https://github.com/ryancblocker/comment-classification-model

The repository includes:

- A script to load and pre-process the data.
- The training script to run the model for each language.
- The code for evaluation and a small inference script to classify new comments.
- Instructions on installing dependencies and running the code so that anyone can replicate the results.

VII. Conclusion

I think using a transformer-based model to perform multi-label classification of code comments from Java, Python, and Pharo was a great move! Despite the skeptical results the model reached near-perfect accuracy, precision, recall, and F1-scores very quickly, as shown by the training and validation loss curves.

If I ever had the opportunity to revisit this project in the future, I think I would find a more challenging dataset, and explore different model architectures to perform more detailed classification for a more robust model. Nothing in life is ever perfect but it was nice for once to train something that performed well.

REFERENCES

[1] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, “BERT: Pre-training of deep bidirectional transformers for language understanding,” in *Proc. NAACL-HLT*, 2019, pp. 4171–4186.

[2] Hugging Face Datasets: NLBSE/nlbse25-code-comment-classification. [Online]. Available: https://huggingface.co/datasets/NLBSE/nlbse25-code-comment-classification

[3] T. Wolf, L. Debut, V. Sanh, et al., “Transformers: State-of-the-art natural language processing,” in *Proc. EMNLP: System Demonstrations*, 2020, pp. 38–45.