Convolutional Neural-Networks for Cross-Language Comment Classification

Table of Contents

Abstract

We aimed to experiment with Keras Convolution Neural Networks (CNN) and improve their accuracy, precision and recall on multilabel text classification. While achieving these goals we also aimed to optimize the models testing time and floating-point operations per second (FLOPS). We chose the Keras model for its wide use in multi-label classification problems. Our model was tuned using data preprocessing, dropout regularization, and batch normalization. These tuning methods helped us reduce overfitting and improve our model's generalization techniques. While convolutional layers are exceptional at detecting patterns in datasets, they were a detriment to our final scoring of our model. Convolutional layers increased the total number of operations/FLOPS required of the model by a drastic amount, and despite our model's efficiency and accuracy metrics, it was not able to overcome the number of operations taken. CNN models are known to be computationally intensive; we chose Keras for its simplicity within the realm of CNN models. Despite that, the high number of required computations is a serious drawback in using these types of AI models. When creating multi-label classifiers for text classification future researchers should extensively exam the aim of their projects, as the computational complexity of these models may cause them to be suboptimal for their given purpose.

Introduction

As software development projects grow, the importance of well-documented code becomes paramount. Comments in code help to improve the readability and usability of a project. However, automating the classification of comments remains a challenging task due to the diversity in programming languages and writing styles. This project aims to automate classification of code comments by developing a machine learning pipeline to classify source code comments across Java, Python, and Pharo.

The primary focus of this study is to build a classification system capable of processing and analyzing code comments at the sentence level. We created three CNN models using the Keras CNN architecture, specifically Keras for sequential classification. The datasets have been pre-processed to remove noise, normalize text, and segment comments into individual sentences. We sought to experiment with our Keras CNN, as CNN models are not known to be the most efficient at text classification, we wanted to both see this in action for ourselves and strive to make a higher performing CNN model for text code comment classification in the languages of Java, Python, and Pharo.

Related Work

The NLBSE'25 competition provides participants with a comprehensive dataset and baseline model which leverages the SetFit framework [1]. The baseline model sets a high standard for classification accuracy and models an efficient solution to comment classification. This process can be replicated by utilizing their Google Colab notebook.

Rani et al. (2021) proposed a multi-language approach for identifying class comment types, introducing a taxonomy of comment categories across programming languages [2]. Their study highlighted the challenges of cross-language classification.

Our model aims to improve upon the NLBSE'25's baseline model for comment classification by utilizing a Convolution Neural Network. We plan to make our CNN perform well by applying additional data processing and other techniques.

Method

Dataset

The dataset consists of **13,216** rows across three programming languages: Java, Python, and Pharo. The data is split into training and testing subsets for each language, with the following distributions:

- **Training Set**: Comprising **10,796 rows**, **82%** of the dataset.
- **Testing Set**: Comprising **2,420 rows**, **18%** of the dataset.

Language Distribution

- **Java**: The largest portion of the dataset, consisting of **7,614 training rows** and **1,725 testing rows**, accounting for **71%** of the total dataset.
- **Python**: Consists of **1,884 training rows** and **406 testing rows**, making up **17%** of the dataset.
- **Pharo**: The smallest subset, with **1,298 training rows** and **289 testing rows**, consisting of **12%** of the dataset.

Features

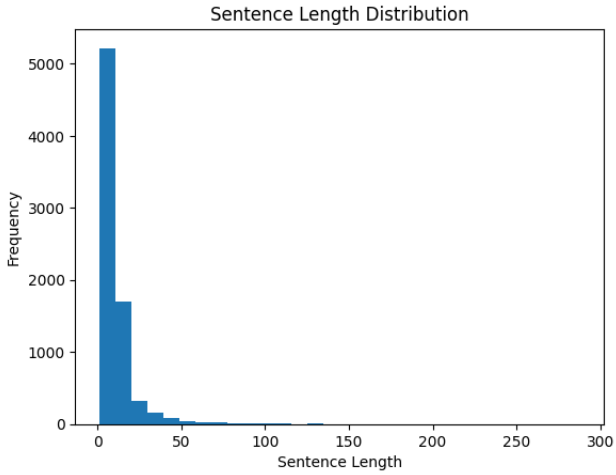
The dataset includes the following key features:

- **`index`**: A unique identifier for each data point.
- **`class`**: The class name from which the comment was extracted.
- **`comment_sentence`**: The actual text of the comments, segmented into individual sentences.
- **`partition`**: A value indicating the dataset split; for this project, all values for `partition` are 0.
- **`combo`**: A combination of `comment_sentence` and `class`, formatted as `comment_sentence | class`.
- **`labels`**: The target variable for classification tasks, represented as a binary list of 0s and 1s, indicating category membership.

Preprocessing Steps

To prepare the dataset for analysis, significant preprocessing was applied:

1. **Lowercasing**: All text was converted to lowercase to ensure case insensitivity.
2. **Special Character Removal**: Non-alphanumeric characters were removed to eliminate noise and simplify processing.
3. **Sentence Splitting**: Comments were segmented into individual sentences to provide more targeted analysis.
4. **Removing overly large sample sizes:** We examined the sequence length of the sentences in our combo column and determined that there was overly large sample size that could negatively impact our model. Removing samples in our training data reduced our input parameters and memory usage while optimizing our training time.



5. **Tokenization:** CNN such as the Keras architecture are not made process text. Tokenizing our data broke down the sentences in the combo column into numerical representations required for the model to function. Our model looks for patterns among these numerical representations in the convolutional layers that can help better inform its predictions. Padding provided by our tokenization provides uniform throughput to every layer of the model, ensuring more optimal accuracy with consistent samples.

Potential Considerations

- **Dataset Imbalance**: Python and Pharo are underrepresented compared to Java, which may affect the model's ability to generalize to these languages. Between the train and test splits for each language we found there to be 9,339 Java samples, 2,290 Python samples, and 1,587 Pharo samples.
- **Impact of Preprocessing**: While preprocessing steps have reduced noise and simplified the data, they may have inadvertently removed language-specific nuances that could contribute to better classification. For example, removing too many samples may improve the overall performance but could negatively impact specific category classification, such as the summary category for Java.

Model Architecture

- **Keras Sequential Architecture**
    - The Keras Sequential model is a simple CNN designed in a linear fashion. It works when each layer has on input and output tensor. At its base the Keras architecture is typically comprised of three layers.
        - **Input Layer:** The layer where model input is received, and data shape can either be inferred or input.
        - **Hidden Layers:** Where the processing of data is done using operations and activation functions.
        - **Output Layer:** Where the final processing of data occurs using a logistic function.
- **Layers**
    - **Embedding/input layer:** Out data was put into our model through an embedding layer. The embedding layer converts tokenized sentences to vectors. This gave our model the ability to create relationships between words. We used a typical input dimension of 15,000 which seemed to be optimal when tuning our model. With an output dimension (where the tokenized words are mapped to) of 128. Our output in this layer is equal to the matrix of vector created for each input or $y = W_x$

- **2 convolutional layers:** We gave our convolutional layers 256 kernels of size 3 and the Rectified Linear Unit (ReLU) activation function. Here our layers apply our 256, size 3 kernels/filters to each input, while our activation function introduces non-linearity to our dataset. We introduce a second convolutional layer so that we can gain more complex insights into the features of our data, that hopefully increases our model's accuracy. The output of these layers would be equal to our size 3 kernels applied to our input $y_i = (x * K)_i$.
- **2 pooling layers:** This layer performs down-sampling on our training data in samples of 4. It selects the highest number from our samples of 4 and outputs it $y_i = \max(x_i, x_{i+1}, x_{i+2}, x_{i+3})$. This reduces the size of our data and consequently, the complexity of it. This can help with overfitting near the end of training.
- **Dropout layer:** We gave our dropout layer a typical rate of 50%. The dropout layer simply decreases the chances of overfitting by randomly setting half of its inputs to zero. It does this to reduce reliance on specific, possibly incorrectly trained neurons and improve model generalization.
- **Flatten layer:** Our flatten layer is used as a transition layer from our other hidden layers to our dense output layer. This ensures that the data enters the dense layer as a one-dimensional vector as it is the only input this layer can receive for classification.
- **Dense/output layer:** We matched the output dimensions to the language specific amount of labels and gave a sigmoid activation function. The sigmoid function allows for multi-label classification by outputting a probability for each class in a sample. The dense layer forms a fully connected layer that connects all of our previous neurons and layers to the current one for making classifications.

Results

We evaluated the performance of two classification models: the baseline SetFit-based model and a Convolutional Neural Network. The models were tested on their ability to classify code comment sentences across multiple categories in three programming languages: Java, Python, and Pharo. Performance was measured using precision, recall, and F1-score metrics for each language and category.

The baseline model demonstrated strong overall performance, particularly for the Java and Pharo categories.

Key Observations:

- **Java**: The baseline model achieved strong results in categories such as Ownership (F1: 1.00) and Pointer (F1: 0.87). However, it struggled with other categories like Expand (F1: 0.37) and Rational (F1: 0.23).
- **Python**: The model performed decently with the Parameters category scoring (F1: 0.8) while categories such as Development Notes scored low (F1: 0.33).
- **Pharo**: The model performed well in categories like Example (F1: 0.89) and Key Messages (F1: 0.74). However, it struggled with Class References (F1: 0.29) and Collaborators (F1: 0.36).
- **Final Scores:** The baseline model achieved a very harmonious relationship between a GFLOPS score of 945.54 and average runtime1.118865 seconds. This gave a good final score 0.7

Baseline Model Performance

Index	Language	Category	Precision	Recall	F1
0	java	summary	0.87839	0.83408	0.85566
1	java	Ownership	1.00000	1.00000	1.00000
2	java	Expand	0.32353	0.43137	0.36975

3	java	usage	0.92506	0.83063	0.87531
4	java	Pointer	0.79018	0.96196	0.86765
5	java	deprecation	0.81818	0.60000	0.69231
6	java	rational	0.17647	0.30882	0.22460
7	python	Usage	0.70079	0.73554	0.71774
8	python	Parameters	0.79389	0.81250	0.80309
9	python	Development Notes	0.24390	0.48780	0.32520
10	python	Expand	0.43363	0.76563	0.55367
11	python	Summary	0.64865	0.58537	0.61538
12	pharo	Key Implementation Points	0.62222	0.65116	0.63636
13	pharo	Example	0.87805	0.90756	0.89256
14	pharo	Responsibilities	0.59615	0.59615	0.59615
15	pharo	Class References	0.20000	0.50000	0.28571
16	pharo	Intent	0.71875	0.76667	0.74194
17	pharo	Key Messages	0.68000	0.79070	0.73118
18	pharo	Collaborators	0.26087	0.60000	0.36364

The CNN model was designed to improve the accuracy, precision, and recall of the baseline model. However, the results indicated a significant performance gap when compared:

- **Java**: The CNN struggled across most categories, with many F1-scores near zero.
- **Python**: The CNN also struggled here scoring the highest in the Parameters category (F1: .17).
- **Pharo**: Example (F1: 0.73) and Intent (F1: 0.59) showed moderate performance. However, the overall results were far less competitive than the baseline.
- **Final Scores:** Our model had better performance in average runtime than the base model with 0.417695 of second. Sadly the GFLOPS score was terrible due to computational complexity with 1,331,866.8. This doomed our score giving us a terrible –52.76 score.

Despite our efforts to improve on CNN's generalization the model consistently underperformed.

CNN Model Performance

Index	Language	Category	Precision	Recall	F1
0	java	summary	0.59211	0.35314	0.44242
1	java	Ownership	0.00000	0.00000	0.00000
2	java	Expand	0.03279	0.03922	0.03571
3	java	usage	0.30121	0.11601	0.16750
4	java	Pointer	0.00000	0.00000	0.00000
5	java	deprecation	0.00000	0.00000	0.00000
6	java	rational	0.06061	0.02941	0.03960
7	python	Usage	0.11429	0.03306	0.05128
8	python	Parameters	0.41177	0.10938	0.17284

9	python	Development Notes	0.10526	0.09756	0.10127
10	python	Expand	0.14583	0.10938	0.12500
11	python	Summary	0.07500	0.03659	0.04918
12	pharo	Key Implementation Points	0.41667	0.23256	0.29851
13	pharo	Example	0.61017	0.90756	0.72973
14	pharo	Responsibilities	0.48000	0.23077	0.31169
15	pharo	Class References	0.00000	0.00000	0.00000
16	pharo	Intent	0.92857	0.43333	0.59091
17	pharo	Key Messages	0.18182	0.04651	0.07407
18	pharo	Collaborators	0.0000	0.0000	0.0000

Conclusion

The results highlight the strength of pre-trained models like SetFit for multi-label classification tasks involving code comment sentences. While CNN architectures offer potential for solving other problems, our model's application to this dataset proved ineffective due to the complexity of the task.

Discussion

The results of our experiments highlight key insights into the challenges and successes of code comment classification using both the SetFit and CNN models.

Baseline Model Performance

The SetFit baseline model performed consistently well across all three programming languages. Categories such as Ownership and Pointer in Java and Example in Pharo achieved high F1-scores. However, categories like Expand and Rational in Java, and Development Notes in Python, showed lower F1-scores. This suggests these categories may contain more ambiguity. Overall, the model's performance in Python was weaker than in Java and Pharo.

CNN Model Limitations

The CNN model struggled to match the performance of the baseline across all scores. This highlights its limitations when applied to complex, high-dimensional data. It is also likely that we made some mistakes in our execution of the CNN model.

Impact of Preprocessing

Preprocessing helped reduce noise and standardize the input data. However, this approach may have removed language-specific nuances. It is possible that this impacted classification performance in categories that rely on these differences.

Strengths and Weaknesses

SetFit Baseline: This model was able to generalize comment classification accurately and was relatively computationally efficient. Its weakest point was taking over an hour to train.

CNN: Our CNN model was computationally efficient, achieving an average runtime of .33 seconds. However, it was unable to adapt to the complex task of classifying comments. This model might perform better with additional data or alternative architectures.

Conclusion and Future Work

Summary

We ran an experiment to see if we could overcome the computational complexities of a CNN model to make an efficient and accurate multi-label classifier for Java, Python, and Pharo code comments. Our model had success in Pharo's intent and example categories but seemed to struggle with the rest of the data. Some categories were not recognizable to our model with zeros in precision, recall, and f1 scores. We had tremendous runtime accumulations, with our average runtime being 0.4177 second per prediction. We expected that computational complexity would be an issue from the outset, and had strived to overcome it, but an abysmal final score of –52.76 produced primarily by our FLOPS measurements proved that we were incapable of overcoming it. We have concluded that while this experiment was a failure we can use it to progress forward and potentially overcome this hurdle in the CNN architecture and achieve better overall efficiency

Future Directions

Our CNN model requires significant improvements to be utilized as a code comment classifier in a professional setting. Improvement upon our work could be achieved by:

- **Hyperparameter tuning**: Deep learning models are heavily influenced by hyperparameters, Ie. Learning rate, batch size, kernel size. These parameters could be finetuned using techniques like grid search to increase model performance.
- **Exploring different models:** It is quite possible that a CNN model is not suited to code comment classification and that other models should be explored.
- **Improving accuracy:** We could preprocess our data more effectively using strategies such as stop word removal to help improve the accuracy of this or future CNN multilabel classifiers
- **Reducing computation steps:** The primary thing we could do is to figure how to best optimes our computations with this model to best achieve our desired effect of computationally efficient accuracy

References

[1] P. Rani, A. Al-Kaswan, G. Colavito, and N. Stulova , “NLBSE’25 Tool Competition: Code Comment Classification,” Google colab, https://colab.research.google.com/drive/1GhpyzTYcRs8SGzOMH3Xb6rLfdFVUBN0P#scrollTo=Q738H9EIw5vh (accessed Dec. 2, 2024).

[2] 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, Nov. 2021. doi:10.1016/j.jss.2021.111047