Iyer, S. et al. (Eds.) (2022). Proceedings of the 30th International Conference on Computers in Education.
Asia-Pacific Society for Computers in Education

Learning Algorithm Implementation Structures for Multilabel Classification via CodeBERT

Karl Frederick ROLDAN[a], Gerd Lowell JANA[b], John Kenneth LESABA[c] & Joshua MARTINEZ[d]
[a]*Undergraduate, Ateneo de Naga University, Philippines*
[b]*Undergraduate, Ateneo de Naga University, Philippines*
[c]*Undergraduate. Ateneo de Naga University, Philippines*
[d]*Thesis Advisor, Ateneo de Naga University, Philippines*
[a]kroldan@gbox.adnu.edu.ph

Abstract: Task constraint feedback is the collective name for any kind of feedback system that checks whether problem-defined constraints were fulfilled by students upon submission of work. This can be as simple as checking if certain programming constructs exist, or if a specific algorithm or data structure required by the problem is fulfilled. Most of these systems use static analysis (Fischer, 2006; Gotel, 2008) or natural language processing techniques (Lane, 2005) to generate feedback. A transformer is a neural network for sequence processing, such as natural languages. Previous work has shown that transformers can be generalized for programming language tasks such as code summarization. In this study, we used the CodeBERT transformer to classify or tag algorithms implemented in some code snippets to check constraint satisfaction. Using a custom dataset containing source code aiming to implement algorithms, we show that CodeBERT is capable of learning structures of how code is implemented regardless of how a programmer names the code. Averaging each label's f1-score, the model was able to obtain an average of 0.85, which showed promising results in the dataset.

Keywords: Deep Learning, Programming Language Processing, Sequence Models, Multilabel Classification, Transfer Learning

1. Introduction

Code summarization is one of the heavily researched areas in programming language processing. (Alon, 2019; Yahav, 2019) It aims to give a semantic description of the input code snippet. For example, a *merge sort* function may be described by a code summarization model as a *sort*. This paper deals with a subset of code summarization that the study calls *algorithm detection*, which focuses on classifying functions more specifically. Given a code snippet, an algorithm detection model attempts to find all the included algorithms and label them as such. This model will form the basis for developing a part of an intelligent tutoring system (ITS) for Project CodeC, a competitive programming application currently under development at Ateneo de Naga University. Specifically, a task-constraint based feedback system that analyzes an input whether it contains the required algorithms by the problem setter.

In this paper, the researchers aim to develop a sequence model that acts as a multilabel classifier to tag seven introductory algorithms and two introductory data structures from code snippets. The included programming languages in the study are *Python*, *Java*, and *JavaScript,* which are covered by the ITS being developed. Furthermore, the included algorithms and data structures would be *insertion sort*, *merge sort*, *selection sort*, *quick sort*, *bubble sort*, *linear search*, *binary search*, *linked lists*, and *hash maps*. Since this paper focuses on algorithm detection, evaluation of real-world student submissions will not be studied, but hopes to serve as a foundational study for a future ITS.

2. Review of Related Literature

2.1 Programming Language Processing

Sequence models are machine learning algorithms that are designed to input and/or output sequential data such as time series, audio, signals, texts, and videos. Multiple-sequence neural networks have been proposed with varying performances on different tasks in the past decades. In this paper, the researchers make use of the transformer that is currently the state-of-the-art sequence model. (Vaswani, 2017) Transformer models do not process the input sequentially, which renders it easily parallelizable. The auto-encoding transformer BERT (Jacob Devlin, 2019) was pre-trained on a corpus of 16GB of data, while RoBERTa (Liu, 2019) had a larger corpus of 160GB and achieved better results than the former. Recent breakthroughs have shown that sequence models can be generalized to programming languages as well. Central themes in research include code summarization (Rui Xie, 2021) which can help in automatically commenting on a snippet of code, code-clone detection (Hui-Hui Wei, 2017) which can be used for finding functional similarities for two code snippets that look distinct from each other, and code completion (Alon, 2020) which can be used for assistance when coding. Furthermore, these models can learn programming language semantics by leveraging the Abstract Syntax Trees (AST), (Lili Mou, 2016; Long Chen, 2019) encoding AST paths into tokens which can be processed by traditional networks (Alon, 2019; Yahav, 2019) and by Hoare triples of a program. (Piech, 2015). Furthermore, the CodeBERT autoencoding transformer is a replica of RoBERTa but trained on an entirely different dataset, which is composed of *Java*, *Python*, and *JavaScript* snippets. (Feng, 2020)

2.2 Task-Constraint Feedback

Task-constraint feedback refers to systems that propagate feedback on whether some required tasks have been met by the student. Some of the research done includes WeBWork, which attempts to perform tests that mimic software quality assurance testing (Gotel, et al., 2008) and ProPL, which is a dialogue-based system that aims to help programmers learn program planning by asking questions (Lane, H. C. 2005). Both of these systems give a response on whether the student has correctly satisfied the given constraints. This paper aims to provide a system that is capable of detecting algorithms in a submitted snippet that can be used for teaching algorithms and related classes.

3. Technical Background

3.1 CodeBERT

A transformer is an encoder-decoder sequence model that is non-recurrent, that is, it takes the input sequence without the need for loops like Recurrent Neural Networks would use. (Vaswani, 2017) The BERT family of transformers are categorized as autoencoders, that is, they only contain the encoder part, which can be used for compressing the input data. (Devlin, 2019)

The CodeBERT transformer is an exact replica of RoBERTa trained in six different programming languages. (Feng, 2020) Furthermore, CodeBERT was trained using unimodal data, consisting solely of code snippets, and bimodal data, comprising a pair of a code snippet and an English language description of the snippet. Since CodeBERT is already a trained transformer, this study utilizes the learned features to make better predictions from a small dataset. To optimize the model, the researchers fine-tuned the last three out of twelve encoding layers of CodeBERT. This enables CodeBERT to learn the dataset gathered by the researchers, instead of relying solely on the artificial neural network built on top of CodeBERT.

4. Methodology

4.1 Dataset and Labeling

Since there are no publicly available datasets on programming language processing that aim to classify algorithms, the researchers gathered this study's dataset from public GitHub repositories containing Python, Java, and Javascript source codes which aim to implement included algorithms and data structures. Figure 1 shows that *linear search* is the most-gathered algorithm in the custom dataset and *binary search* has the fewest. Most of the files in the dataset implement only one of the algorithms in the study. After labeling, features that do not contribute to code semantics, such as extra whitespaces, comments, and docstrings were removed. The researchers achieved an almost perfect agreement of 0.829 during labeling according to Kraemer's extension of the Kappa coefficient. (McHugh, 2012)

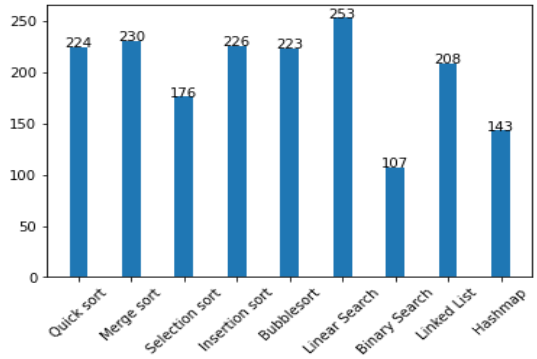

Figure 1. Number of Labels per Algorithm/Data Structure.

4.2 Training

Initially, the model was trained with frozen CodeBERT weights. That is, CodeBERT was used as a feature extractor under the assumption that it had learned some basic algorithms during its pre-training tasks. However, this was not the case, and the model had poor performance. Two main problems were found. First, CodeBERT did not learn these algorithms at all during the pre-training. Second, the model merely memorized identifier names and function orders.

For the first problem, the researchers fine-tuned the first three encoding layers of CodeBERT. This enabled CodeBERT to learn algorithm implementations on top of programming languages. The output is used as an input to a feedforward neural network. For the second problem, the dataset was augmented as described in the next subsection.

4.3 Data Augmentation

To address the problem where the model memorized identifier names, the dataset was augmented by swapping identifier names (function names, class names, and variable names) with randomly generated strings of lengths ranging from 1 to 3. The short length maximizes the probability that CodeBERT will not tokenize the identifier as unknown. Otherwise, the model might have a bias towards unknown tokens. The researchers took steps to ensure that the mappings are bijective. Each file was augmented between 0 to 6 times to ensure randomness in identifier names. This will ensure that the model used in the study learns the structure rather than names. However, language keywords and common functions were not augmented as these are usually crucial to how the algorithm and the programming language work.

Another problem found was that the model was memorizing the ordering of functions in the file. To mitigate this problem, the researchers concatenated two files of different programming languages that do not have common algorithms implemented and added both concatenation orders into the dataset under some probability. The resulting files were then augmented further with random strings from the first augmentation technique discussed

4.4 Evaluation

Precision, recall, and f1-score were recorded for each label for model evaluation purposes. In this paper, the researchers maximized the precision to ensure that an unmet task will not pass the constraint checker.

5. Results

5.1 Experimental Setup

The first experiment result shown in this paper combines the use of augmented and original datasets for the train set. The test set's augmentation does not use random identifiers. The second experiment removes all the unaugmented code snippets from the train set and moves them to the test set. Hence, the distribution of data between the train set and test set in the second experiment is completely different. The goal is to have the model generalize well.

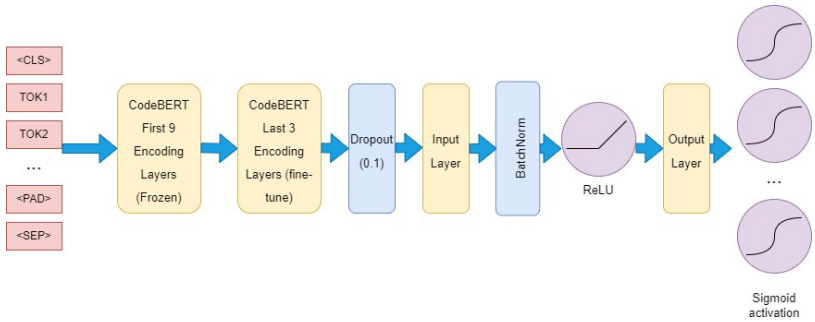

Figure 2. Model architecture.

Figure 2 illustrates the model architecture. To prevent overfitting the dataset, a dropout layer was used prior to feeding the output of CodeBERT into the artificial neural network. Preliminary experiments had all CodeBERT weights frozen. However, this proved ineffective and incapable of overfitting even the training set. Fine-tuning the last three layers proved to be enough to learn the training set and generalize well on the test set. The output layer is activated by nine sigmoid functions, each pertaining to one class. The researchers used binary cross-entropy loss as the loss function.

5.2 Training results

The first experiment makes use of traditional data augmentation techniques wherein the augmented files are used as additional data points for the dataset in the hopes that this will increase the dataset size. Table 3 shows that the model was able to achieve very good results, achieving almost perfect f1 metrics. However, upon experimentation, one problem that was found was implementing an empty function named after one of the algorithms that the model incorrectly classifies. This shows that the model still uses identifiers to make predictions instead of relying purely on code structure.

Table 1. *Evaluation Metrics. (COM) refers to combined augmented and unaugmented dataset during training. (AUG) refers to having only the augmented dataset for training.*

	Recall		Precision		F1 Score	
	COM	AUG	COM	AUG	COM	AUG
Quick Sort	0.938	0.78	0.988	0.957	0.963	0.86
Merge Sort	0.936	0.74	0.994	0.962	0.964	0.836
Selection Sort	0.935	0.81	0.944	0.85	0.939	0.829
Insertion Sort	0.931	0.816	0.974	0.926	0.952	0.867
Bubble Sort	0.919	0.824	0.989	0.836	0.953	0.83
Linear Search	0.936	0.792	0.986	0.82	0.961	0.806
Binary Search	0.914	0.911	0.991	0.886	0.951	0.899
Linked List	0.93	0.876	0.981	0.838	0.955	0.906
Hash Map	0.939	0.779	0.963	0.881	0.951	0.827

To mitigate the bias towards identifier names, the unaugmented dataset was removed from the training set and moved to the test set. Therefore, CodeBERT is forced to learn only the structural information since only files with random identifiers remain in the training set. As shown in Table 1, the model was able to generalize well despite having different data distributions between the train set, composed of augmented files, and the test set, composed of unaugmented files.

Table 1 shows that the first experiment yielded better metrics. However, note that this is due to CodeBERT using identifier names to make predictions and cannot generalize well on experimental inputs. The second experiment shows slightly lower metrics. This is because the model does not make use of identifier names to create predictions but solely leverages on the function structure. Hence, the model was confused for some of the function names but still displayed good predictive power as demonstrated by the high precision value.

5.3 Examples

To experimentally evaluate each trained model, the researchers created a simple web-based application that tags algorithms present in a code snippet. To demonstrate the sanity of the model, Figure 3 shows a code snippet that implements a selection sort algorithm but is named "binary search". Since the model was trained on random identifiers, it was able to correctly identify "selection sort" on structure alone.

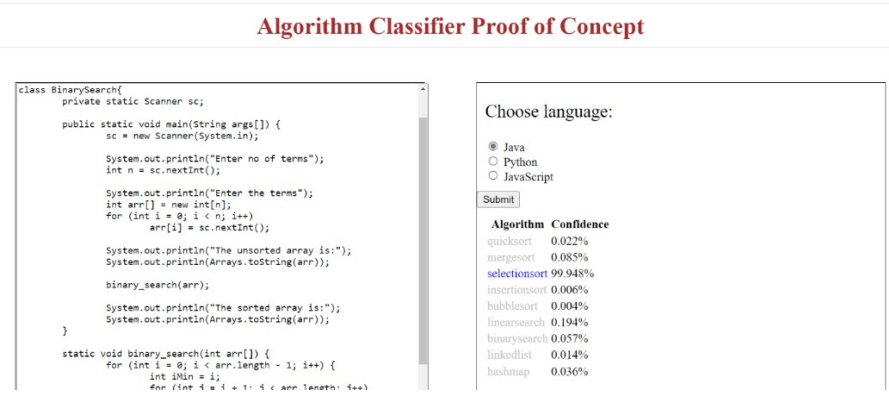

Figure 3. Implementing *selection sort* but *binary search* as identifier.

6. Conclusion

CodeBERT was originally trained for tasks such as code summarization and code search. However, the study was able to demonstrate that CodeBERT is not only capable of learning the meaning of a given code snippet, which can be used for automatic commenting, but can also learn structural information and make predictions from this. Furthermore, the model was able to generalize well and will mostly lead to correct predictions if the input code snippet is in the same distribution as the test set.

Since the study has proven that CodeBERT is capable of learning structural information, a more enhanced model with more classes and datasets geared towards student submissions can be used for algorithm tagging. This model can eventually be used for a task-constrained feedback system. Both the code (Roldan, 2022a) and the dataset (Roldan, 2022b) for this study can be freely accessed on GitHub.

6.1 Recommendations

This study only investigates whether CodeBERT can learn code structure without the aid of abstract syntax trees or graphs. This does not prove that the model can properly predict algorithms from student submissions. Therefore, the researchers recommend that this area be studied and how useful the model will be in aiding task-constraint feedback. To do this, the researchers recommend gathering the dataset from competitive programming websites, such as Project CodeC, to eliminate expected bias against student submissions. To mitigate CodeBERT limitations, such as the 512 max token length and limitations of language choices, other models such as graph neural networks can be investigated.

Acknowledgements

We would like to thank all the people who prepared and revised previous versions of this document.

References

Alon, U. S. (2020). Structural language models of code. *ICML'20: Proceedings of the 37th International Conference on Machine Learning*, 245-256.

Alon, U. Z. (2019). Code2Vec: Learning Distributed Representations of Code. *Proc. ACM Program. Lang.*

Feng, E. A. (2020). CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *Findings of the Association for Computational Linguistics: EMNLP 2020*, 1536–1547.

Fischer, G. (2006). Distributed intelligence: extending the power of the unaided, individual human mind. *AVI '06: Proceedings of the working conference on Advanced visual interfaces*, 7-14.

Gotel, E. A. (2008). Global perceptions on the use of WeBWorK as an online tutor for computer science. *2008 38th Annual Frontiers in Education Conference*.

Hui-Hui Wei, M. L. (2017). Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. *IJCAI'17: Proceedings of the 26th International Joint Conference on Artificial Intelligence*, 3034–3040.

Devlin, (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 4171-4186). Minneapolis, Minnesota: Association for Computational Linguistics.

Lane, H. C. (2005). Teaching the tacit knowledge of programming to noviceswith natural language tutoring. *Computer Science Education, 15*, 183-201.

Lili Mou, G. L. (2016). Convolutional neural networks over tree structures for programming language processing. *AAAI'16: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, 1287-1293.

Liu, Y. a. (2019). *RoBERTa: A Robustly Optimized BERT Pretraining Approach.* Retrieved from arXiv: https://arxiv.org/abs/1907.11692

Long Chen, W. Y. (2019). Capturing source code semantics via tree-based convolution over API-enhanced AST. *Proceedings of the 16th ACM International Conference on Computing Frontiers*, (pp. 174-182).

McHugh, M. L. (2012). Interrater reliability: the kappa statistic. *Biochemia medica 22, 3*, 276-282.

Piech, C. H. (2015). Learning program embeddings to propagate feedback on student code. *ICML'15: Proceedings of the 32nd International Conference on International Conference on Machine Learning*, 1093-1102.

Roldan (2022b). *Algorithm Multilabel Classifier dataset*. Retrieved from GitHub:n https://github.com/gerdiedoo/alg-dataset

Roldan (2022a). *Algorithm Multilabel Classifier source code*. Retrieved from GitHub: https://github.com/karlfroldan/algorithm-detection

Rui Xie, W. Y. (2021). Exploiting Method Names to Improve Code Summarization: A Deliberation Multi-Task Learning Approach. *2021 IEEE/ACM 29th International Conference on Program Comprehension*, 138-148. Vaswani, e. a. (2017). Attention is all you need. *Proceedings of the 31st International Conference on Neural Information Processing Systems* (pp. 6000-6010). Red Hook: Curran Associates Inc.

Yahav, U. A. (2019). code2seq: Generating Sequences from Structured Representations of Code. *International Conference on Learning Representations.*