Influence of Contrastive Learning on Source Code Plagiarism Detection through Recursive Neural Networks

*Note: This paper was peer-reviewed and presented for the 3rd edition of the IMITEC on the 24th Novemember 2021.

Manuel A. Fokam
School of Computer Science and Applied Mathematics
The University of the Witwatersrand
Johannesburg, South Africa
2490967@students.wits.ac.za

Ritesh Ajoodha
School of Computer Science and Applied Mathematics
The University of the Witwatersrand
Johannesburg, South Africa
ritesh.ajoodha@wits.ac.za

***Abstract*—Plagiarism during programming assignments is a problem in academia. It hinders the ability of academic instructors to truly judge students' performance and thus, prevents students from receiving adequate help from their instructors. In cases where the number of code submissions for a particular assignment is relatively small, the instructor can inspect each code submission to determine whether they are similar. But as the number of code submissions grows, it becomes difficult to detect similarities between them. Therefore, this induces the need for an automatic source code plagiarism detector.**

Previous studies showed that we could use the abstract syntax tree (AST) of a source code to get an accurate representation of the source code for neural network computations. Although a study even presented a recursive artificial neural network named Abstract Syntax Tree-based Neural Network (ASTNN) that could represent source codes into vector embeddings using their ASTs, it does not use contrastive learning paradigms, shown to increase the performance of Siamese networks in similarity detection tasks.

Therefore, this paper presents an improved version of the ASTNN for code clone detection, where we modify the original model for contrastive learning. Experiments demonstrated that we outperform the original ASTNN model in code clone detection tasks, with a +5% improvement in the F1-score of our model. This study aims at improving the way we perform similarity detection tasks involving programming languages.

***Index Terms*—Plagiarism detection, Programming Languages, Abstract Syntax Trees, Deep Learning**

I. Introduction

Plagiarism can be defined as the act of using someone's work without attribution. As a result of the democratization of online platforms such as GitHub, Stack Overflow and Medium, the incidence of plagiarism in academia and industry has risen to a point where it is no longer necessary to have a detailed understanding of a particular concept to solve a programming task. The only thing required is to know the relevant platforms from which we can obtain the snippet of code that will accomplish the desired task.

In the industrial field, there is no need to pay keen attention to the author of a particular source code, as long as this code works and respects the software development standards established by the concerned company. However, source code plagiarism is a threat in academia because this prevents instructors from evaluating their students' performance. Therefore, there is a strong need to build computer systems to detect plagiarism in students' work.

A research work particularly caught our attention because it proposed a recursive neural network to efficiently exploit the structure of an AST, known to be an accurate syntactical representation of a source code [1]. The work presented a model which could compute the vector representation of a source code by recursively calculating the output at some node locations of its AST. Then, after encoding the AST into a vector representation, this representation was passed through a bidirectional Gated Rectifier Unit (GRU) to generate the final source code embedding. Additionally, the model could predict the similarity between two source codes through Siamese network architecture. Compared to previous traditional and network-based approaches [2], [3], [4], this approach showed superior results.

However, the method proposed with ASTNN uses binary cross-entropy as its loss function though studies showed that supervised contrastive loss functions outperform their cross-entropic counterparts in image similarity detection tasks [5]. Therefore, this paper aims at validating this claim for representation learning tasks such as source code similarity detection.

The main contribution of our work is the modification of the ASTNN model to detect source code similarity by contrastive learning rather than binary cross-entropic learning.

Our paper is structured as follows. Section II gives a brief overview of the work done towards source code plagiarism detection. Section III provides the background needed to understand our work, and Section IV gives a thorough description of our approach for code clone detection tasks using contrastive learning and the experiments we conducted to validate the claim made in this study. Section V provides the results of the experiments conducted and a discussion about the obtained results. Finally, we conclude our work in Section VI.

II. Related Work

Source code plagiarism detection is not a novel task in the computing field. Researchers have succeeded in providing methods for detecting similarities between source codes. There are two main challenges towards solving this problem, namely the process that transforms source codes into computable forms (Source code representation) and the technique used for finding similarities between these processed forms (Similarity detection) [6].

A. Source Code Representation

A crucial step in developing a source code plagiarism detector is the representation of the source codes. Many approaches differ in the way they represent source code for plagiarism detection. ASTs are not the only representations that give syntactical information about a source code. Some methods use control-flow graphs (CFGs) as structural representations of source code [7]. Nonetheless, while most methods use either the ASTs or CFGs of source codes to represent them as code vectors [1], [8], some create weighted bags of words the represent function names present in the source code using only word embeddings [9]. Also, a popular approach is to use N-grams which are widely used in natural language processing tasks for source code representation [10]. Finally, to exploit the dependencies between functions and variables in a source code, there were promising results regarding the use of recurrent neural networks as it gave the ability to represent source codes as a sequence of vectors [11].

B. Similarity Detection

Another aspect of developing a source code plagiarism detector is the algorithm or model used to compare and detect plagiarism based on the representation of the source code used. Firstly, some approaches which opted for representing source codes as vectors calculate the similarities between these source codes through a standard K-Means clustering method [12]. Other studies proposed Siamese networks for code clone detection [9], [1], based on their success in detecting face clones [13].

More formally, given a Siamese network for a similarity detection task, Let us assume this Siamese network takes in two inputs, namely the two source codes representation from which we wish to predict the similarity score $S \in [0, 1]$. We can define the similarity function as:

$$S = F(s_1, s_2),$$

where s_1, s_2 are the source codes' representation which serves as input to the Siamese network. Thus, this method aims at teaching the Siamese model how to map a pair of inputs (s_1, s_2) to the correct similarity score S by using a binary cross-entropy loss function.

III. Background

Throughout this section, we will provide a detailed explanation of two concepts important for understanding the foundations of this study. We will first briefly define recursive neural networks and explain why their architecture is suitable for source code analysis. Finally, we will also introduce contrastive learning and the role they play in representation learning.

A. Recursive Neural Networks

These are artificial neural networks that can operate on structured data. In addition to being used for natural language processing, they are also suitable networks to learn tree data representations. Recurrent neural networks are examples of recursive neural networks. Studies have shown that deep recursive neural networks are more effective than shallow ones for natural language processing tasks such as sentiment analysis on text data [14]. Given a tree layer structure with parent node p, the weight of the entire tree as W, the weight at the output as W^{score} and children node c_1, c_2 as in the figure below,

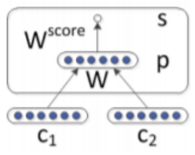


Fig. 1. Recursive layer with a parent node p and two children nodes, c_1, c_2

we can obtain the output s of this layer using this formula,

$$s = W^{score} p,$$

$$p = W[c_1; c_2] + b.$$

In this particular case, the tree structure to encode has a height of 1. For arbitrarily large trees, such as ASTs, this computation is applied recursively on the top of the tree p to get an output s.

B. Contrastive Learning

Using cross-entropic loss functions to train neural networks for classification and similarity detection tasks tend to make them learn the following subtasks:

- How to accurately map inputs into a representation space.
- The decision boundaries we can use to group inputs with similar representations in a latent space.

While this approach works well for image classification, the neural network only cares about making sure it correctly separates image embeddings from different classes. Thus, they only learn how to represent inputs accurately enough to separate them by decision boundaries. But, the model must have been exposed to all the possible representations of the networks' inputs to draw accurate decision boundaries. This way of learning requires a broad dataset of possible inputs. Thus, some of these models cannot faithfully reproduce their training performance when evaluated. It would be better to focus on learning the discriminative features of these inputs. Therefore, it would be much simpler to draw decision boundaries across different inputs.

Contrastive learning is the appropriate learning method for such tasks. Through this learning method, the model can predict the distance between these inputs in their latent space. Contrastive learning was shown to outperform cross entropic methods of image classification [5].

Therefore, contrastive learning is suitable for code clone detection tasks because of its ability to make a model learn how to contrast between two inputs.

C. Siamese Networks

Mainly used for one-shot image recognition [13], Siamese Neural Networks are an arrangement of twin networks, usually trained using shared weights to rank the similarity between two inputs. They are used in identification tasks such as face recognition and signature identification.

By applying a contrastive loss function to the distance between the outputs of the twin networks, Siamese networks can learn how to differentiate between two inputs. Therefore, the objective of this Siamese network would be to minimize the output distance for similar inputs and maximize this distance when they are different.

IV. Methodology

In this section, we provide a systematic description of the methodology we used to obtain the findings of this study. Firstly, we describe the dataset used and outline the steps we followed to convert the source codes from this dataset into a suitable form for our proposed deep learning model. Secondly, we explain the minor modifications done to the ASTNN model for contrastive learning and provide a detailed explanation of the loss function used. Finally, we outline the process followed to validate the claim of this study regarding contrastive learning's superiority over binary cross-entropic learning for code clone detection tasks.

A. Data Acquisition

The dataset consists of code fragments made in the C programming language and extracted from the Open Judge System (OJS) [15]. These code fragments are subclassed into 104 classes, where each class represents a task the code fragments belonging to that class solve. However, we use a preprocessed version of the dataset where 6.6% of the code fragments in the dataset are authentic clone pairs [1]. We label as clones any two code fragments which solve the same task. We use OJS data in this study because it contains C source codes that have already been stripped out of macros and other tokens which could produce errors during parsing.

B. Data Pre-processing

Before feeding the proposed model with our dataset, we must transform each source code into a sequence of statement trees, with the root of each tree being a statement block such as a method declaration or a control flow statement. We carry out this process through the following steps.

1) Retrieving the AST of a Source Code: In this step, we use a C parser [16] to extract the source code's AST. Also, we remove the directives from the source codes because the chosen parser does not support C directives. This action will not hinder the efficiency of our model in detecting code clones as almost all directives follow the same pattern irrespective of the author of the source code.

2) Splitting the AST into a Sequence of Statement Trees: In this step, we extract every statement block from our source code as shown in Fig. 2. At the end of this step, for each statement block identified in the AST of the source code, we extract this block and transform it into a multidimensional list data structure consisting of token indexes present in this statement block. These token indexes represent particular syntax tokens in the source code, such as **if**, **for**, **int** and **while**. We obtain these token indexes from a word dictionary generated by training a word2vec [17] model with the tokens of the code samples. We use token indexes instead of the actual tokens because neural networks work with numerical inputs, but tokens are strings. After transforming each statement block into a multidimensional list, we join these lists together to form a sequence of preprocessed statement trees.

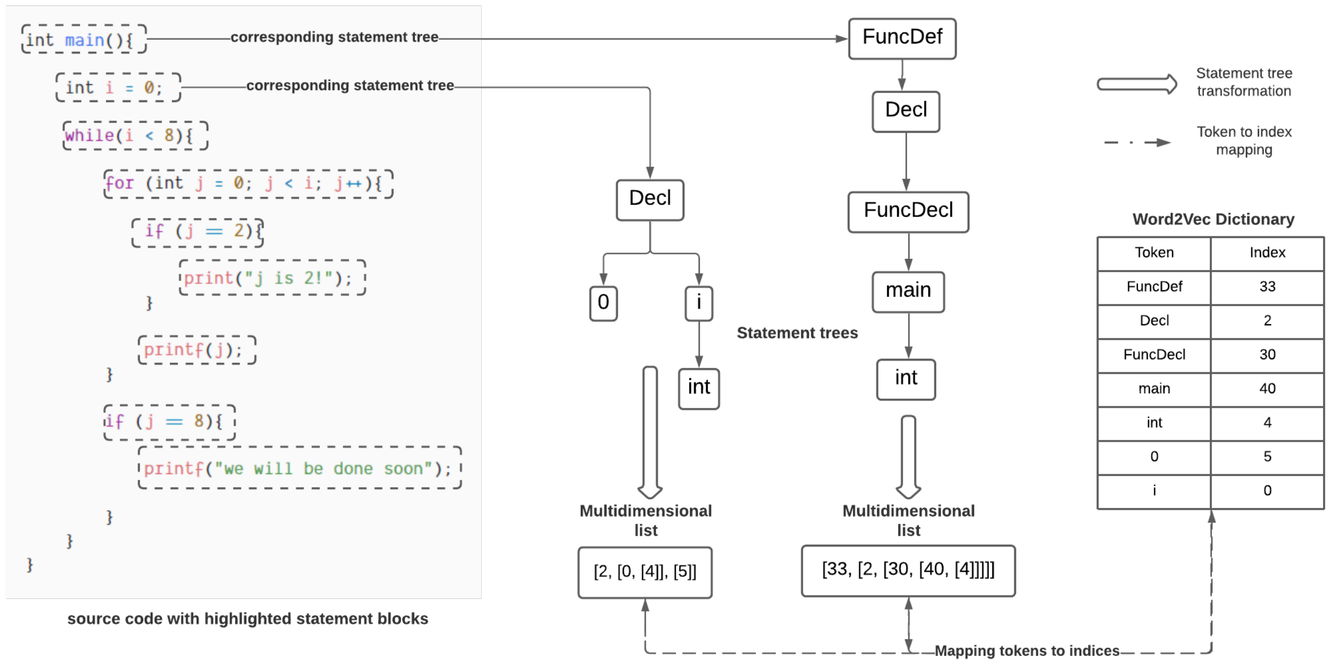
C. Clone Detection Model

In this section, we provide a thorough description of the model used to detect code clones. The model is a Siamese network of statement tree encoders designed to predict the distance between two source codes through contrastive learning.

The statement tree encoder is the model responsible for encoding the sequence of statement trees into a vector representing the source code. Firstly, we have a batch tree encoder whose job is to transform statement trees into a set of vectors, where each vector is the output from a recursive function call on each statement tree.

For example, given a sequence of statement trees representing a particular source code, each statement tree is encoded into an embedding vector through recursive calls on the root of the statement tree. So, imagine we pass 3 statement trees to the encoder, knowing that the batch tree encoder has an embedding layer with an embedding dimension of 5. Therefore, the batch tree encoder will output a set of 3 5-D vectors, with each vector representing the output for each statement tree.

Next, this set of vectors from the batch tree encoder serves as input to a bidirectional GRU, which outputs the vector representation of the entire code fragment. Bidirectional GRU is used to learn the natural dependencies between the statement blocks of the source code [1]. Finally, the output of the GRU serves as the input to a densely connected layer of neurons,



```
int main(){
   int i = 0;
    while(i < 8){
      for (int j = 0; j < i; j++){
         if (j == 2){
            print("j is 2!");
         }
         printf(j);
      }
      if (j == 8){
         printf("we will be done soon");
      }
    }
}
```

Token	Index
FuncDef	33
Decl	2
FuncDecl	30
main	40
int	4
0	5
i	0

Fig. 2. Preprocessing framework of the input source codes. The statement nodes (framed in rectangles with dotted borders) are extracted into their corresponding statement tree then, these statement trees are transformed into a multidimensional list where the tokens (Decl, FuncDef, main, int, etc.) are replaced by their indexes from a Word2Vec model which has been trained with these tokens.

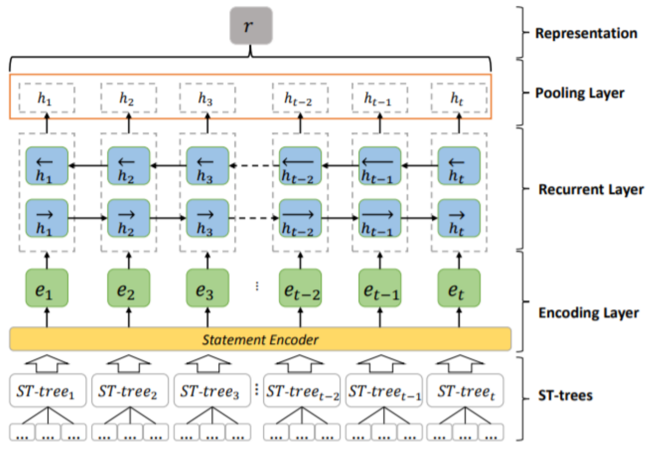

r
Representation
Pooling Layer
Recurrent Layer
Encoding Layer
Statement Encoder
ST-trees
h_1 h_2 h_3 h_{t-2} h_{t-1} h_t
e_1 e_2 e_3 e_{t-2} e_{t-1} e_t
ST-tree$_1$ ST-tree$_2$ ST-tree$_3$ ST-tree$_{t-2}$ ST-tree$_{t-1}$ ST-tree$_t$

Fig. 3. Proposed architecture of the AST Neural Network for source code representation [1].

which outputs the final vector representation of the source code or, specifically, the source code embedding.

As we stated earlier, instead of using a binary cross entropic loss function for this study, we will use a contrastive loss function because it helps the model learn how to contrast two inputs. In other words, it helps predict the relative distance between two inputs. More Formally, given a Siamese network of a source code encoder model, let us assume that this Siamese network outputs two d-dimensional vectors V_1 and V_2, representing the Siamese network's outputs. Let us define contrastive loss function as:

$$L = (1 - Y) \times D^2 + Y \times max(0, m - D)^2, \qquad (1)$$

where $Y \in \{0, 1\}$ is the similarity label between the inputs of the Siamese network, $D \in \mathbb{R}$ is the distance between the embeddings, V_1, V_2 and, $m > 0$ is the variable that defines the minimum distance allowed between the embeddings of dissimilar input source codes. We can also see D as the calculated distance between the vector representation of the two inputs of the Siamese network. We chose Euclidean distance for this study instead of the absolute distance metric because it makes our model converge faster and enables us to transform the source code fragments into vector embeddings which can be clustered through K-Means clustering.

Taking a closer look at (1), we observe that if $y = 1$, this means that the two inputs are similar. Therefore, we wish to minimize D but, if $y = 0$, it means the inputs are different and, as a consequence, we wish to maximize D. Also, it is important to note that the variable m gives the minimum separation between dissimilar inputs. This loss function is bounded below by 0 as one property the distance metric D must have is non-negativity.

D. Evaluation Metrics

To evaluate our approach on unseen data, we use precision and recall. We define precision and recall using the following formula:

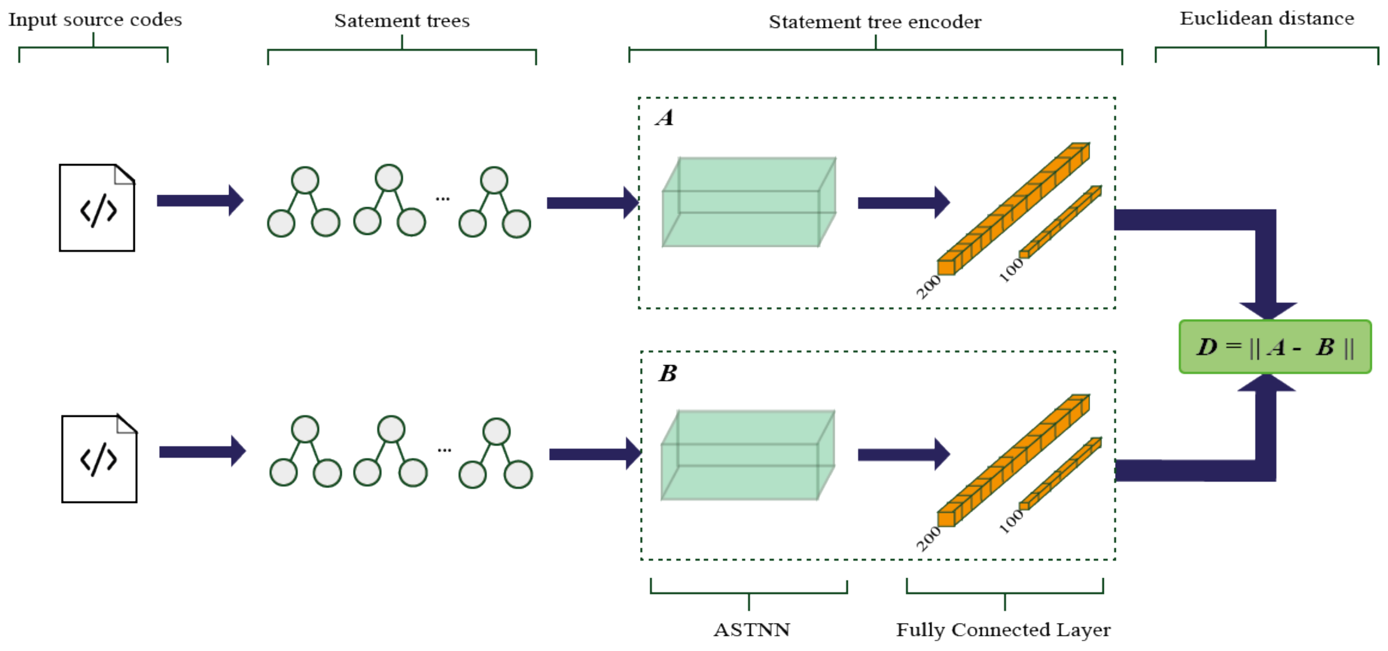


Fig. 4. Code clone detection framework. Each input source code is transformed into a set of tree data structures called statement trees as in Fig. 1, then, these sets are passed into the same statement tree encoder which outputs a 100-D vector for each input source code. The Euclidean distance between the vectors is then calculated and pass to the contrastive loss function.

$$Precision = \frac{t_p}{t_p + f_p},$$

$$Recall = \frac{t_p}{t_p + f_n},$$

where t_p, f_p and f_n are, respectively, the true positives, false positives and false negatives made by our model during its evaluation.

Using only precision and recall to evaluate our model can be misleading because the recall can be 100% if our model only makes positive predictions, but our model would suffer from a low precision. Also, we can have a model with 100% precision if its only positive prediction is correct, but this model will suffer from a poor recall.

A solution to this problem is to use the F1-score metric in conjunction with precision and recall. F1-score is the balance between recall and precision. We define the F1-score F using the following formula:

$$F = 2 \times \frac{Precision \times Recall}{Precision + Recall},$$

E. Experimental Procedure and Analysis

The following section provides a comprehensive evaluation of the approach used in this study. Specifically, our objective is to answer the following questions:

- **(RQ1)** How does the model perform with unseen source codes?
- **(RQ2)** Does contrastive learning perform better than the original cross-entropic learning approach?

To answer these research questions, we split out the dataset and used 60% to train the original ASTNN model and our modified model. We used approximately 26 000 clone pairs to train our model. We reserved 20% for validation and 20% for testing. However, we did specific experiments to get results that would answer our research questions. These experiments are listed below.

*1) **(RQ1) How does the model perform with unseen source codes?:*** To answer this question, we evaluated our model on a test dataset, where we recorded its precision, recall and F1-score. Also, since the margin m in the contrastive loss function (1) also serves as a hyperparameter for our learning algorithm, we decided to train our model with different values of m and observe the impact of these variations on its performance. It is important to mention that the only hyperparameter that was updated throughout the training was the margin m. Therefore, for each margin value, we recorded the top performance (F1-score) of the model at that margin and the threshold distance d used to consider two input source codes as being similar or dissimilar.

*2) **(RQ2) Does contrastive learning perform better than the original cross-entropic learning approach?:*** While comparing the average training time of our models, we also recorded the training and validation loss, as well as the accuracy of the two approaches throughout training. Finally, we calculated the F1-score, recall and precision of each model on the test dataset and compared them.

V. Results and Discussion

In this section, we use the results obtained from our experiments to answer the research questions listed earlier in Section

TABLE I
PERFORMANCES OF DIFFERENT ASTNN MODELS.

Model	Precision	Recall	F1-Score
ASTNN	0.972	0.899	0.935
ASTNN-c (our model)	**0.975**	**0.992**	**0.983**

TABLE II
PERFORMANCES OF OUR MODEL WITH DIFFERENT MARGIN VALUES.

Margin	F1-score	Optimal Threshold
0.5	0.983	0.3
1.0	0.983	0.65
2.0	0.967	1.35
5.0	0.954	3.20

IV-E

*1) **(RQ1) How does the model perform with unseen source codes?**:* Even though our dataset was highly unbalanced, our model could achieve a far greater F1-score than the baseline approach. We found that our proposed model could achieve a precision, recall and F1-score of up to 97.5%, 99.2% and 98.3% in our test set as shown in TABLE I.

Also, from TABLE II outlining the performance of our model with different margin values, we observe that as the margin m increases, the performance decreases and, the optimal threshold for the top performance at this margin value is slightly above half of the margin value. Therefore, we can observe that suitable margin values for these tasks are reasonably small or in the range [0, 3]

*2) **(RQ2) Does contrastive learning perform better than the original cross-entropic learning approach?**:* From the plot in Fig. 5, we observe a faster convergence in our model. Furthermore, contrastive learning appears to possess a more stable training than binary cross-entropy. Additionally, we observe from the confusion matrices in Fig. 6 that our model detects more code clones than the original model, thereby performing better by 5% (F1-score) as shown in TABLE I.

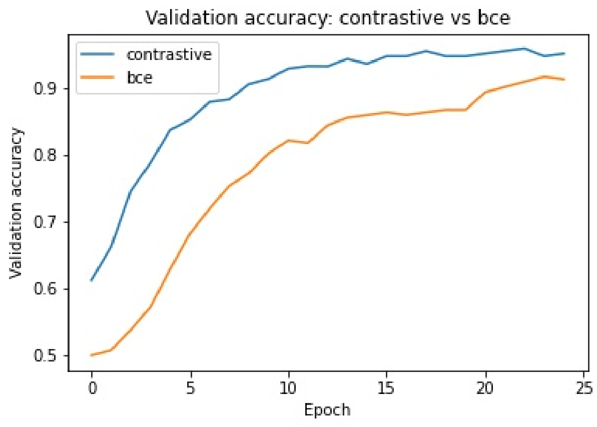


Fig. 5. Validation accuracy through training (bce = binary cross-entropy)

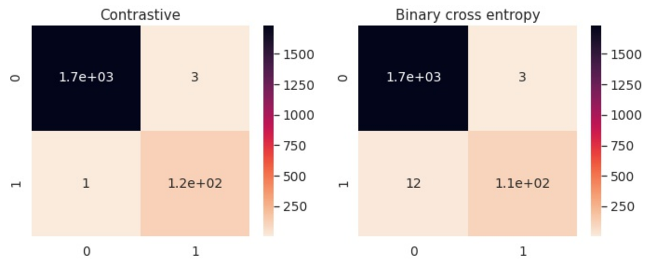


Fig. 6. Confusion matrix of our model (left) and original code clone detection model (right)

Because our models are deep neural networks, the computational cost of training them is higher than their shallow counterparts. It took approximately 20 hours to perform training for each model using the same computational resources. Also, we observed that the time to make a single prediction in each model was about 200 milliseconds.

VI. CONCLUSION

In this paper, we presented a modified version of the ASTNN model proposed for source code similarity detection [1]. Also, we showed that by changing the training objective of the original model and accommodating its architecture to this changeset, we could achieve a higher performance in code clone detection. There are three components to this framework that permits this achievement. First, there is the embedding component, whose job is to map identified tokens in the source code into word embeddings. Secondly, we have a representation component that transforms each source code into a sequence of tree data structures called statement trees and encodes this sequence into a source code embedding. Lastly, we have the most important component, the similarity detection component that calculates the distance between two source code embeddings to maximize this distance if these embeddings come from dissimilar source codes and minimize this distance if the embeddings come from similar source codes.

We can use our work as the foundation for other applications in big code clone detection tasks. However, recursive neural networks may be computationally expensive. Therefore, further improvements should be made to the recursive algorithm used to encode the source codes. Also, an alternative to using euclidean distance as a distance metric in the contrastive loss function would be the cosine similarity metric due to its robustness against the scale of the vector embeddings representing the source codes.

ACKNOWLEDGEMENT

This work is based on the research supported in part by the National Research Foundation of South Africa (Grant numbers: 121835).

References

[1] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, “A novel neural source code representation based on abstract syntax tree,” in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 783–794.

[2] Y. Kim, “Convolutional neural networks for sentence classification,” *CoRR*, vol. abs/1408.5882, 2014.

[3] W. Zaremba and I. Sutskever, “Learning to execute,” *ArXiv*, vol. abs/1410.4615, 2014.

[4] X. Huo and M. Li, “Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code,” in *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, ser. IJCAI’17. AAAI Press, 2017, p. 1909–1915.

[5] P. Khosla, P. Teterwak, C. Wang, A. Sarna, Y. Tian, P. Isola, A. Maschinot, C. Liu, and D. Krishnan, “Supervised contrastive learning,” *CoRR*, vol. abs/2004.11362, 2020.

[6] M. Ďuračík, E. Kršák, and P. Hrkút, “Current trends in source code analysis, plagiarism detection and issues of analysis big datasets,” *Procedia Engineering*, vol. 192, pp. 136–141, 2017.

[7] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, “Deep learning similarities from different representations of source code,” in *Proceedings of the 15th International Conference on Mining Software Repositories*, ser. MSR ’18. New York, NY, USA: Association for Computing Machinery, 2018, p. 542–553.

[8] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “Code2vec: Learning distributed representations of code,” *Proc. ACM Program. Lang.*, vol. 3, no. POPL, Jan. 2019.

[9] C. Xie, X. Wang, C. Qian, and M. Wang, “A source code similarity based on siamese neural network,” *Applied Sciences*, vol. 10, no. 21, 2020.

[10] M. Ďuračík, E. Kršák, and P. Hrkút, “Source code representations for plagiarism detection,” in *Learning Technology for Education Challenges*, L. Uden, D. Liberona, and J. Ristvej, Eds. Cham: Springer International Publishing, 2018, pp. 61–69.

[11] U. Alon, O. Levy, and E. Yahav, “code2seq: Generating sequences from structured representations of code,” *CoRR*, vol. abs/1808.01400, 2018.

[12] P. Hrkút, M. Ďuračík, M. Mikušová, M. Callejas-Cuervo, and J. Zukowska, “Increasing k-means clustering algorithm effectivity for using in source code plagiarism detection,” in *Smart Technologies, Systems and Applications*, F. R. Narváez, D. F. Vallejo, P. A. Morillo, and J. R. Proaño, Eds. Cham: Springer International Publishing, 2020, pp. 120–131.

[13] N. K. Ahmed, E. E. Hemayed, and M. B. Fayek, “Hybrid siamese network for unconstrained face verification and clustering under limited resources,” *Big Data and Cognitive Computing*, vol. 4, no. 3, 2020.

[14] O. Irsoy and C. Cardie, “Deep recursive neural networks for compositionality in language,” in *Advances in Neural Information Processing Systems*, Z. Ghahramani, M. Welling, C. Cortes, N. Lawrence, and K. Q. Weinberger, Eds., vol. 27. Curran Associates, Inc., 2014.

[15] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, “Convolutional neural networks over tree structures for programming language processing,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI’16. AAAI Press, 2016, p. 1287–1293.

[16] E. Bendersky, “pycparser,” 2010. [Online]. Available: https://github.com/eliben/pycparser

[17] T. Mikolov, K. Chen, G. Corrado, and J. Dean, “Efficient estimation of word representations in vector space,” 2013.