Rolling Stone: A Code Similarity Detection System

Albert (Yu) Jiang, Joe Mirza, Ricardo Jénez

School of Information
University of California, Berkeley
{yu.albert.jiang,mirza2020,rrj}@ischool.berkeley.edu

Abstract

Software plagiarism is a growing concern in academic and commercial software development environments. The state of the art in these NLP-PL systems (Natural Language Processing - Programming Languages) has not received as much attention and has not advanced as rapidly as it has for more conventional NLP tasks. Our contribution in this paper is developing an NLP system that performs code similarity detection on software code using state of the art deep learning architectures. This system allows the detection of similar code programs (not just snippets) and determines if there is sufficient similarity between code examples to warrant further review for plagiarism. Our objective was to outperform the industry-standard similarity detection system, named MOSS ("Measure Of Software Similarity") which uses text fingerprinting[1] to identify software similarity. We appear to have achieved this for a large labeled corpus of C programming assignments.

1 Introduction

In many universities, there is an increased incidence of plagiarism in code implemented by students in programming assignments [2]. Similarly, in industry, commercial code is frequently "infected" by code snippets taken from the open-source or on popular sites like StackOverflow (SO), leading to liabilities for companies who have made unauthorized use of that code.

Additionally, copy and paste code leads to more brittle code bases. Detecting similar code fragments that are swept up into common functions or classes would significantly reduce code complexity. "Recent studies have shown that developers use SO snippets in their software projects, regardless of maintainability, security, and licensing implications...". [3].

The most frequently used software plagiarism detection tool is named MOSS ("Measure Of Software Similarity" [1]), developed at UC Berkeley by Alex Aiken, et al. based upon the paper[1]. MOSS has been in use for over 20 years and is still the gold standard. Given the advances in NLP and Deep Learning, we were curious to see if we might be able to develop a deep learning plagiarism detector that was better than MOSS.

The advent of attention-oriented architectures such as Transformers [4] and the language models they've enabled, such as BERT[5] and T5[6], would seem to provide an opportunity to advance the state of the art in this area. Programming Language (PL) variants, such as CodeBERT, trained on large code corpora, seem even more promising.

2 Data Acquisition

There are several factors to consider when determining if two code files, or parts of those files, are similar and whether that similarity is potentially the result of plagiarism. Neither syntactic nor semantic similarity alone is sufficient to conclude that plagiarism has occurred. Two pieces of code can be semantically quite similar (i.e. they solve the same problem), but arrive at the solution using different approaches (e.g. a while loop vs a for loop for iteration). Syntactic similarity, where the code is the same word for word, may not be flagged as being copied, if variable names are modified or functions are ordered differently.

Given this, we decided that more stringent similarity thresholds were needed in order to make a plagiarism determination between two pieces of code. It lies between pure syntactic similarity and semantic similarity. We looked at work done by Oscar Karnalim, et. al. [7], to provide some of the needed context for this task. By leveraging a labeled plagiarization dataset, we can attempt to assess both syntactic and semantic similarity and perhaps get insights into intent to plagiarize as well.

In searching for a robust plagiarism dataset, we looked at several options, including datasets for code generation from text (CodeXGlue) which has several different datasets from CodeBert trained with a variety of coding tasks[8] as well as StatQC [9] which looked at Stack Overflow code snippets. We felt that we needed to better control the experience and align it with actual software practices in universities or commercial settings, where we could confirm plagiarism with the code authors as in the dataset compiled by Ljubovic [10]. We will go into more detail below about the op-

tions and concerns for each case. (See Appendix for an overview of the datasets we chose from in Table 8.)

2.1 Data Sets

Our primary dataset is Programming Homework Dataset for Plagiarism Detection Dataset [10]. It is used to examine how student programming styles and IDE usage differs between students who plagiarize their homework and those who do not. Ljubovic uses code from an introductory programming class with $\sim$ 500 students per class. When determining whether a particular assignment has been plagiarized, they not only look at code similarity but also offer students the opportunity to refute a determination of plagiarism. Those students who do not mount a defense are determined to have plagiarized their code.

2.2 Data Set Preparation

The first challenge with this task is determining similarity and the second is determining plagiarism, which implies intent. To do this we augmented the Homework Dataset [10] (which contains the binary plagiarization label) with MOSS's similarity measures. We did this by running each source code pair for a particular assignment through MOSS, which provided us with MOSS's evaluation of percent similarity between each file pair and the number of lines that were similar. This provided us with a MOSS baseline for each document pair.

MOSS produces output of the top N pairs that are similar in HTML files that we download. We set N = 1000 for each assignment. We then processed these HTML files and joined them with the plagiarism labels from the Homework dataset. We generated training and test sets from these top 1000 pairs for each assignment and there were between sixteen and twenty assignments per year. These training/test examples are put in csv files with each line having percent similarity, both sources from the files, the filename identifiers, the percent similarity, and a label indicating whether or not the source code pairs were plagiarized. The source code from each file ranges between 79 bytes up to 11K bytes. Our training and test set reflected an 80/20 split of the 17,280 samples. We viewed this as sufficient data to fine-tune these pre-trained models and determine if our model was better than MOSS at identifying plagiarism.

3 Initial Approach

Before creating our models, we attempted to replicate one of the small number of papers in this space from Karnalim et al. [7]. They had a relatively small Java dataset, which contained 105 non-plagiarized files and 355 plagiarized ones, and used a Vector Space Model (VSM) and Language Model (LM) to predict plagiarism.

For the VSM techniques, we vectorized each file's token count with respect to the collection's token count, then compared the vector of each original code file to the vectors of its related files that were either plagiarized or not plagiarized to get cosine similarity values. For the LM techniques, we compared the token count of each original code file against the token count of each of its related files and of the entire collection to get similarity values.

For both VSM and LM, we used the similarity measures that Karnalim et al. used. We then ranked the files by similarity and then applied mean average precision (mAP) to the file rankings. On balance, our mAP values were similar to Karnalim et al.

We then attempted to use the VSM and LM based techniques on a portion of our main dataset. Although the VSM based techniques consistently outperformed LM based techniques, neither technique attained mAP scores significantly greater than 0.5 on any of the variants of each method we applied. We concluded these techniques did not perform well enough to pursue further.

4 Main Approach

As mentioned in section 2, we generated a dataset that had MOSS scores along with annotation/labels from the Homework Plagiarization dataset. This provided us with a MOSS baseline for all of our experiments. We started by applying simple techniques such as Sentence Embeddings and Cosine Similarity to assess similarity for plagiarized and unplagiarized code pairs.

We then compared these results with more specifically trained models. One significant challenge we encountered was that models like BERT and CodeBERT have token limits of 512, while many of the code files would require 10x that many tokens or more to represent in their entirety. That led us to experiment with models capable of longer token inputs, such as LongFormer and BigBird.

4.1 Experimentation with Embeddings

To start, we encoded each document using BERT, CodeBERT, LongFormer, and T5Large as-is (without fine tuning on our task) and applied either max or mean pooling to get a single vector for each document. Next, we calculated cosine similarity for each file pair using those embeddings. We then plotted Accuracy, Precision, Recall and F1 against ascending values of those similarity scores. In Table 1 on page 3, we show the performance of these models along these metrics, with MOSS performance as our base case. In that table, the value maximized is accuracy. At the similarity that maximizes accuracy for each model, the other three

Table 1: Moss As Measure Of Plagiarism Vs.Cosine Similarity Of Embeddings

Model	Experiment Details	Accuracy	Precision	Recall	F1-Score
MOSS	Code pairs, similarity, 50 setting for code common block. Evaluate at similarity above 73.1 pcnt to get max accuracy	**0.6970**	**0.6840**	**0.7325**	**0.7074**
BERT	bert-base-uncased, average embeddings over 256 Tokens from sentence-transformer, cosine sim = 0.9674 and above, max pooling	0.6519	0.6505	0.6566	0.6536
CodeBERT	microsoft/codebert-base, average embeddings over 256 Tokens from sentence-transformer, cosine sim = 0.9747 and above, max pooling	0.6603	0.6796	0.6066	0.6410
LongFormer	allenai/longformer-base-4096,2K tokens, from sentence transformer, cosine sim 0.9367 and above, max pooling	0.6307	0.6164	0.6922	0.6521
T5Large	sentence-transformers/sentence-t5-large, 2K tokens, from sentence transformers,mean pooling, min similarity 0.29, mean pooling	0.5619	0.5712	0.4967	0.5313

metrics are provided. (The data-set is balanced between plagiarized and not plagiarized.)

MOSS outperforms these simple, embeddings-based models across all four metrics. At 73.1% similarity, MOSS sees its maximum accuracy of 69.7%. We were surprised that LongFormer and T5Large performed so poorly relative to BERT and CodeBERT. We had expected those models' larger maximum input lengths to capture more signal and result in outperformance.

4.2 Experimentation with Model Training

Our next step was to fine-tune these transformer-based models on our plagiarism classification task. We started with BERT, CodeBERT, the BERT Chinese model, BigBird 1K and 2K as well as Longformer 4K. This work is summarized in Table 2 on page 4. We trained each of our models for 4 Epochs and used an 11K/3.5K/3.5K split for training, validation and test (after randomly over sampling the minority class we had 21K/7K/7K). We used a BERT based Tokenizer (Roberta in the case of CodeBERT), and Models based upon Trainers for SequenceClassification.

All of the trained models outperformed MOSS by Accuracy and all of the models except Longformer 4K outperformed MOSS by F1-Score. So training is effective. BERT and CodeBERT outperformed the models with larger maximum input lengths (BigBird 1K, 2K and Longformer 4K). Strangely, the BERT Chinese model performed about as well as BERT and CodeBERT. Why this was the case provided us with a useful insight on the importance of tokenization methods, which we'll explore in the next section.

4.3 Experimentation with Whole Word Masking

As noted in the last section, the Chinese BERT model had the highest F1-Score. Why would a model trained on Chinese perform so well? We learned that the Chinese BERT model uses whole word masking, while BERT models use wordpiece tokenization, which can split each word into token parts (e.g. "surfing" becomes "surf" and "ing"). Furthermore, we know that normal BERT models only mask on a token level. This method can help reduce the size of the vocabulary, while increasing its expressiveness. However, we hypothesized that it does not perform as well when dealing with programming language code elements. To deal with such documents, masking each code element in its entirety instead of masking tokenized parts helps retain the integrity of the vocabulary. To address this, we trained on an uncased BERT Large model that uses whole word masking. This model produced our best results yet, beating the BERT model trained on Chinese. This is summarized in Table 3 on page 4. While whole word masking did not play a part in our best model, understanding why the Chinese BERT model performed so well provided us with a better understanding of the importance of the right choice of tokenization methods.

4.4 Experimentation with Preprocessing

To attempt to further improve our models, we analyzed some of the false negatives and false positives of our best models. In the course of this analysis, we observed that whitespace and comments represented a large fraction of the tokens. We suspected they had little predictive power and, given the maximum input limits on our best models (512 tokens), these tokens were *displacing* tokens that might actually possess some predictive power.

To address this, we used ANTLR, a customizable

Table 2: Trained Model Performance

Model	Experiment Details	Accuracy	Precision	Recall	F1-Score
MOSS	Code pairs, similarity, 50 setting for code common block. Evaluate at similarity above 73.1 % to get max accuracy	0.6970	0.6840	0.7325	0.7074
BERT	bert-base-uncased,512 tokens max ,learning rate 1e-5	0.7833	0.8032	0.7505	0.7760
BERT - Chinese Model	bert-base-chinese,512 tokens max,learning rate 1e-5	0.7676	0.7460	0.8114	0.7774
CodeBERT	codebert-base,512 tokens, learning rate= 1e-5	0.7845	0.8144	0.7367	0.7737
BigBird 1K	google/bigbird-roberta-base, 1K tokens, $attention_typeoriginal_full, learningrate1e-5$	0.7369	0.6992	0.8318	0.7597
BigBird 2K	google/bigbird-roberta-base, 2K tokens,learning rate 1e-5)	0.7585	0.8369	0.6421	0.7261
Longformer 4K	allenai/longformer-base-4096, 4K tokens, learning rate 1e-5	0.7140	0.8322	0.5361	0.6521

Table 3: Trained Model Performance - Whole Word Masking

Model	Experiment Details	Accuracy	Precision	Recall	F1-Score
MOSS	Code pairs, similarity, 50 setting for code common block. Evaluate at similarity above 73.1 % to get max accuracy	0.6970	0.6840	0.7325	0.7074
BERT	bert-base-uncased,512 tokens max ,learning rate 1e-5	0.7833	0.8032	0.7505	0.7760
BERT - Chinese Model	bert-base-chinese,512 tokens max,learning rate 1e-5	0.7676	0.7460	**0.8114**	0.7774
BERT w/Whole Word	bert-large-uncased-whole-word-masking, 512 tokens,learning rate 1e-5	**0.8194**	**0.8809**	0.7386	**0.8035**
CodeBERT	codebert-base,512 tokens, learning rate= 1e-5	0.7845	0.8144	0.7367	0.7737

parsing tool we had used to preprocess the code files in the VSM/LM models mentioned previously, to ignore whitespace and comments and lower case all tokens. When our models were rerun on the code files with this preprocessing, we saw a noticeable improvement to the BERT Large model that uses whole word masking. CodeBERT performed even better, despite not using whole word masking. In fact, the CodeBERT model with ANTLR preprocessing was our best overall model. This is summarized in Table 4 on page 5, and an example of the before and after tokens can be seen in Figure 5 in the Appendix.

5 Discussion

Our initial expectation was that using cosine similarity of embeddings that were not fine-tuned and to which either max or mean pooling was applied would yield similar results to what MOSS was able to generate. It became apparent, across a range of untrained embeddings, that an untrained model was unable to fully address the layers of noise in the dataset. This can be seen in Table 1 on page 3.

Almost all of our *trained* models, however, outperformed MOSS based on F1-score (except for Longformer 4K). This can be observed in Table 2 on page 4. For example, by F1-score, BERT outperformed MOSS by .0686 (.7760 vs .7074). Accuracy is also quite a bit higher, 0.7833 vs 0.6970, a difference of .0863. What leads to such an increase in accuracy? What does the model learn in 512 tokens that MOSS sees over entire files? We'll attempt to understand that a bit better in our error analysis section.

Additionally, that Longformer 4K result highlights another finding: architectures with longer maximum input lengths, such as BigBird 1K, 2K and Longformer 4K, which enable these models to capture more of the input documents, consistently underperformed models like BERT and CodeBERT, which have smaller maximum input lengths. It appears that critical distinctions between plagiarized and unplagiarized files occur early in these documents. Furthermore, it appears the benefits of capturing more of the document for these longer input architectures is outweighed by their inability to capture critical classification distinctions, relative to models like BERT or CodeBERT.

Lastly, our best performing model resulted from using CodeBERT along with text preprocessing (using ANTLR) that removed whitespace and comments. This combination had an F1-Score of 0.8675, .0915 better than a trained BERT model. The improvement in accuracy was similar. We speculate that this perfor-

Table 4: Trained Model Performance with Text Preprocessing

Model	Experiment Details	Accuracy	Precision	Recall	F1-Score
MOSS	Code pairs, similarity, 50 setting for code common block. Evaluate at similarity above 73.1 % to get max accuracy	0.6970	0.6840	0.7325	0.7074
BERT	bert-base-uncased,512 tokens max ,learning rate 1e-5	0.7833	0.8032	0.7505	0.7760
BERT w/Whole Word	bert-large-uncased-whole-word-masking, 512 tokens,learning rate 1e-5	0.8194	**0.8809**	0.7386	0.8035
BERT w/Whole Word ANTLR	bert-large-uncased-whole-word-masking, ANTLR preprocessing remove whitespaces and comments, 512 tokens, learning rate 1e-5	0.8195	0.8041	0.8449	0.8240
CodeBERT	codebert-base,512 tokens, learning rate= 1e-5	0.7845	0.8144	0.7367	0.7737
CodeBERT ANTLR	codebert-base, ANTLR preprocessing remove whitespaces and comments, lower case tokens, 512 tokens, learning rate 5e-6	**0.8634**	0.8424	**0.8940**	**0.8675**

mance is due to the combination of using a transformer trained on code elements that was better able to identify similar syntactic arrangements and the input preprocessing that allowed more of those code elements to get into input, which is especially important given the small maximum input lengths CodeBERT can accept relative to the length of many of these documents. (The preprocessing effectively increased the signal to noise ratio.) These results are summarized in Table 4 on page 5.

5.1 Error Analysis

To better understand the differences between MOSS and our BERT and CodeBERT models, we first examined the differences in their errors in Table 5. Both BERT and CodeBERT have far more true negatives than MOSS (or, equivalently, fewer false positives). Similarly MOSS has a higher false negative count when compared to both CodeBERT and BERT.

When examining the differences between CodeBERT and BERT, there's little difference between their true negative (false positive) counts. What distinguishes CodeBERT is its superior performance with true positives/false negatives. We have detailed some of the reasons above and we've included a specific example in the appendix in Figure 4 that highlights the power of preprocessing and a model trained on programming languages.

Specifically, we saw CodeBERT perform significantly better than BERT at recognizing plagiarism from basic code replacements or structure changes. In that example, we see that CodeBERT was not fooled when a for loop was replaced with a while loop. Some other cases where CodeBERT demonstrated this ability include recognizing different formats for conditions (e.g. if else) and comparisons (e.g. >=<). We theorize that CodeBERT might recognize when these code structures have similar meanings, even if the tokens are different. After analyzing BERT's tokenization vs CodeBERT's tokenization, we theorize that CodeBERT is doing a better job tokenizing code since it was trained on code. As seen by the highlighted example in Figure 6, CodeBERT understands that certain code elements like "++" have inherent meaning and should remain as a single token whereas BERT separates the "+"s. This improved tokenization almost gives CodeBERT a pseudo-whole word masking effect, just by identifying tokens more effectively. Furthermore, CodeBERT is able to recognize statements like the beginning of for loops as an entire construct whereas BERT splits them apart. These two factors further serve to help CodeBERT detect plagiarism.

5.2 Ablation Study

To better understand the effects of the ANTLR preprocessing that removed whitespace and comments, we ran CodeBERT with just whitespaces removed and just comments removed. Although each was a significant improvement over the base CodeBERT model, neither were close to the performance of having both as shown in Table 6. By only removing comments, CodeBERT seemed to be able to better ignore non-code elements that don't effect code plagiarism. By only removing whitespace, CodeBERT seemed to be able to use more effective tokens by wasting less processing on just spacing. Together, they allowed CodeBERT to better focus on the code that determines plagiarism in a way that was more than the sum of their parts.

6 Conclusion

We found that multiple transformer-based models could be trained to predict plagiarism more effectively than MOSS, using a combination of syntactic and semantic pattern recognition that was beyond MOSS's capabilities. Furthermore, CodeBERT, a transformer trained on code, produced our best model. It appears CodeBERT has two advantages. First, during tokenization,

Table 5: Comparison Of Results

Model Comparison	tn	tn-diff	fp	fp-diff	fn	fn-diff	tp	tp-diff
MOSS-BERT	2135	168	**1159**	**767**	**845**	**577**	2449	422
BERT-MOSS	2734	**767**	560	168	690	422	**2604**	**577**
CodeBERT-BERT	**2742**	172	552	164	349	102	945	443
BERT-CodeBERT	2734	164	560	172	690	443	2604	102

Table 6: CodeBERT Ablation

Model	Experiment Details	Accuracy	Precision	Recall	F1-Score	F1 Delta
CodeBERT	codebert-base,512 tokens, learning rate= 1e-5	0.7845	0.8144	0.7367	0.7737	0.0
CodeBERT ANTLR Remove Comments Only	codebert-base, ANTLR preprocessing remove comments, lower case tokens, 512 tokens, learning rate 5e-6	0.8045	0.7856	0.8376	0.8108	0.0371
CodeBERT ANTLR Remove Whitespace Only	codebert-base, ANTLR preprocessing remove whitespaces, lower case tokens, 512 tokens, learning rate 5e-6	0.8268	0.8307	0.8209	0.8258	0.0521
CodeBERT ANTLR Remove Both	codebert-base, ANTLR preprocessing remove whitespaces and comments, lower case tokens, 512 tokens, learning rate 5e-6	**0.8634**	**0.8424**	**0.8940**	**0.8675**	0.0938

it preserves code integrity in a way that other transformers do not. Second, CodeBERT appears to be better at understanding the types of simple changes in code that might be indicative of plagiarism (e.g. swapping a for loop with while). Lastly, we found that preprocessing our input to remove whitespace and comments improved performance by, we believe, increasing the amount of meaningful code available in input. MOSS's performance on large datasets, however, was extraordinary in regard to speed and accuracy, whereas our models were somewhat slower to produce results. So this represents an opportunity for continued improvement.

6.1 Future Work

As is often the case, the work that's yet to be done greatly exceeds what we have done to this point:

- Additional preprocessing. Removing text shared across a large number of files. For example, removing the template text or code shared in common across every assignment.
- Pursuing methods of ingesting the entire document, without suffering from the shortcomings we saw with long input models such as BigBird, Longformer, etc.
- Increase the size of the training data from balanced 21K data points to over 100K to see what the impact is on performance.
- Using the improved model in our Sentence Transformer framework for creating embeddings and averaging cosine similarity of those blocks over the entire document.
- Should we be able to find suitable labeled data, test these models on other programming languages such as Python or Java.
- Better understand which features are being generated by going in depth using AI explainability techniques like LIME.
- Better latency. We want to extend our models to perform as well a MOSS, which means being able to do comparisons on the order of a few tens of milliseconds, which at this point is not possible.

References

[1] Saul Schleimer, Daniel S. Wilkerson, Alex Aiken. **Winnowing: Local Algorithms for Document Fingerprinting**. 2003.

[2] Kevin W. Bowyer, Lawrence O. Hall. **Experience Using "MOSS" to Detect Cheating On Programming Assignments**. 1999.

[3] Sebastian Baltes, Christoph Treude, Stephan Diehl, Lorik Dumani. **SOTorrent: Reconstructing and Analyzing the Evolution of Stack Overflow Posts**. 2018.

[4] Ashish Vaswani, Noam Shazeer, Niki Parmar,J akob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser. **Attention Is All You Need**. 2017.

[5] Jacob Devlin, Ming-Wei Chang, Kenton Lee ,Kristina Toutanova. **BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding**. 2019.

[6] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, Peter J. Liu. **Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer**. 2020.

[7] Oscar Karnali, Setia Budi, Hapnes Toba, Mike Joy. **Source Code Plagiarism Detection in Academia with Information Retrieval: Dataset and the Observation**. 2019.

[8] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, S hujie Liu. **CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation**. 2021.

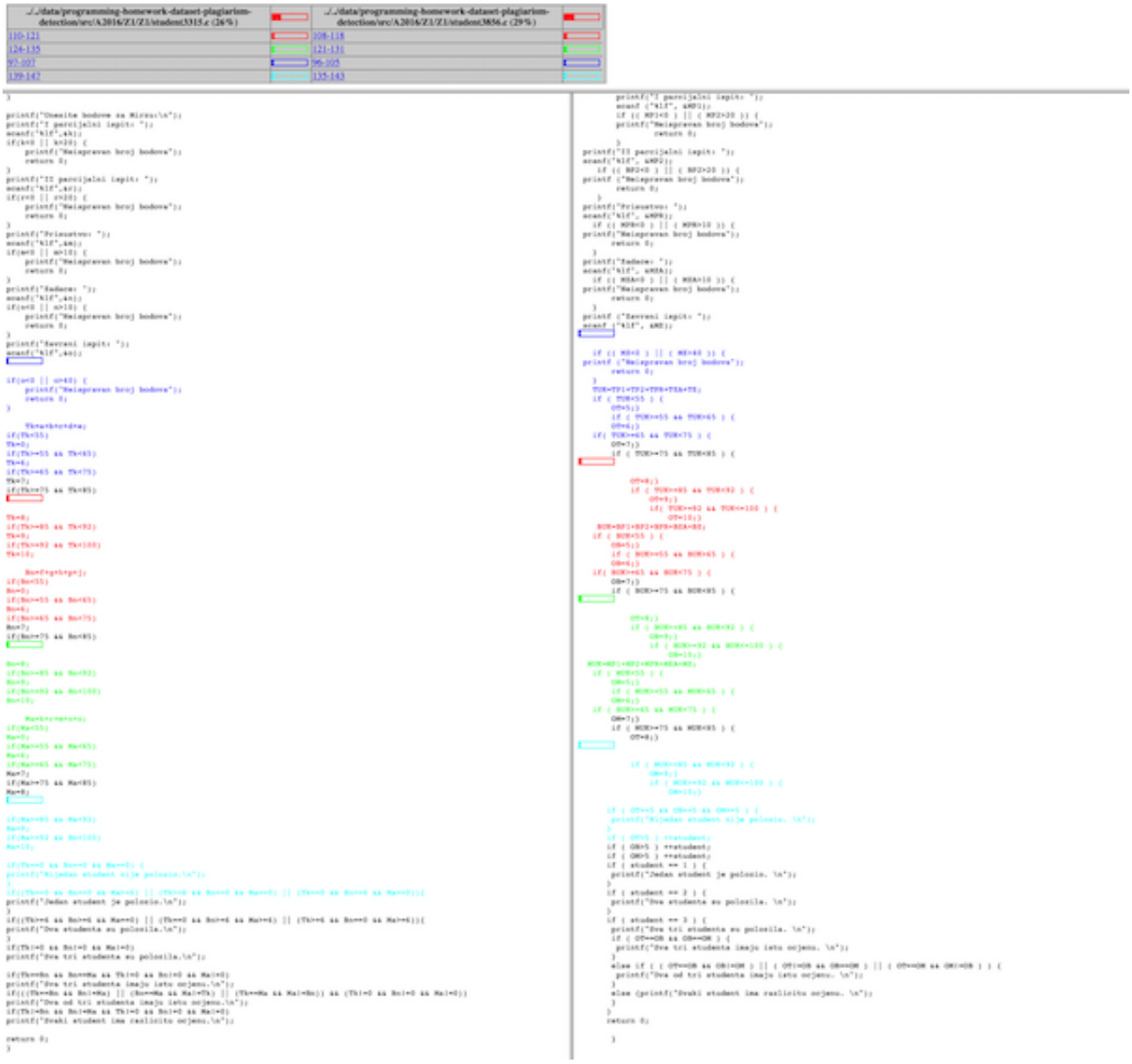
[9] Pengcheng Yin, Bowen Deng, Edgar Chen, Bogdan Vasilescu, Graham Neubigu. **Learning to Mine Aligned Code and Natural Language Pairs from Stack Overflow**. 2018.

[10] Vedran Ljubovic. **Programming Homework Dataset for Plagiarism Detection**. 2020.

7 Appendix

MOSS Output Example

FIGURE 1: Moss Example



Karnalim VSM and LM Techniques Across Levels of Plagiarism

Table 7: Levels Of Plagiarism

Level	Attack Signatures	Example
1	Comment and whitespace modification	Removing all comments from given source code
2	Identifier modification (i.e., changing lexical name from one to another)	Renaming all local variables
3	Component declaration relocation	Moving all variable declarations to the beginning of main method
4	Method structure change	Replacing all method invocations with their respective invoked- method's content
5	Program statement replacement (i.e., changing statements with other statements that share similar semantic yet different syntactic form)	Replacing while statement with for statement
6	Logic change (i.e., changing statements with other statements that share no similarity in terms of syntactic and semantic form)	Replacing an iterative traversal with the recursive one that generates similar result

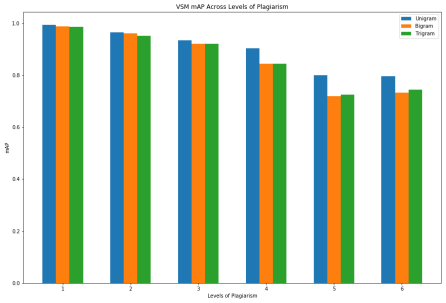


Figure 2: VSM mAP Across Levels of Plagiarism

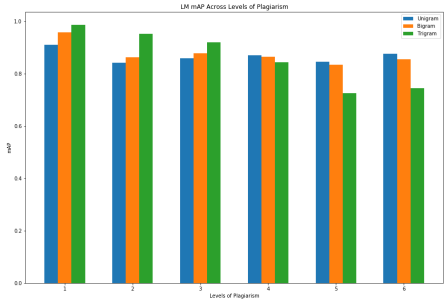


Figure 3: LM mAP Across Levels of Plagiarism

Figure 4: Error Analysis False Negative By BERT not in CodeBERT

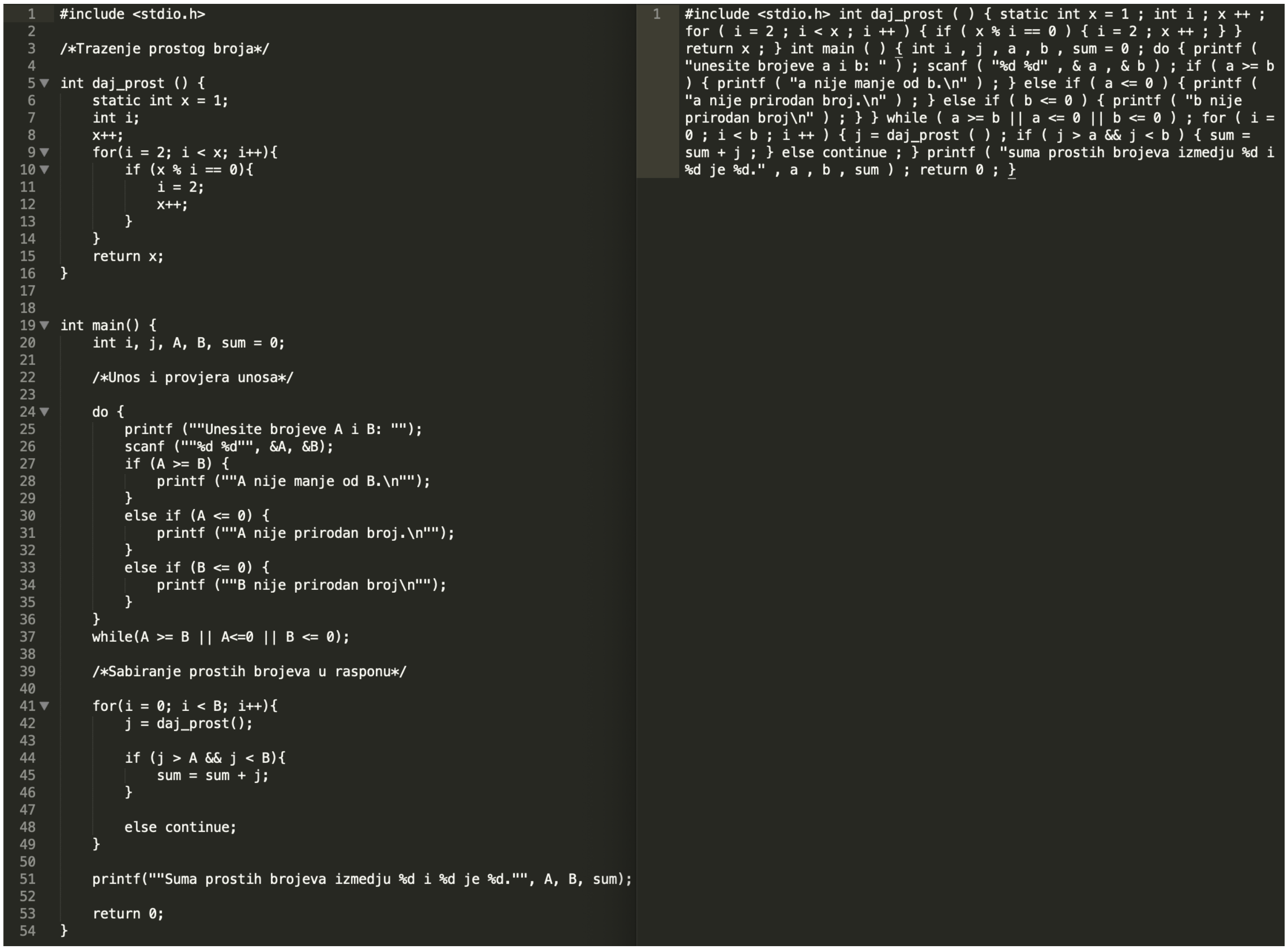
```
440 plagiarize=1 percent=56, source0 length = 1279 source1 length = 1364 ----------------------------------------------------

#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.1415926
#define eps 0.0000000000000000000000000000000000000000000001
int main()
{
        int i, j, brUglova, sekF, ste, min;

        double niz[500] = {0}, sek;

        printf ("Unesite broj uglova: ");

        scanf ("%d", &brUglova);

        for (i = 0; i < brUglova; i++)
        {
                scanf ("%lf", &niz[i]);
                niz[i]*=180./PI;
        }

        for (i = 0; i < brUglova; i++)
        {

                sek = fabs(60 * ((60*(niz[i] - (int)(niz[i]))) - (int)(60*(niz[i] - (int)(niz[i])))));

                sek += 0.5;
```

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#define PI 3.1415926
#define eps 0.0000000000000000000000000000000000000000000001
int main()
{
        int i,j,sekF,ste,min;

        int brUglova;

    double uglovi[500] = {0},sek,a;

        printf("Unesite broj uglova: ");

    scanf("%d", &brUglova);
        i=0;
        while(i<brUglova)
              {
                  scanf("%lf",&uglovi[i]);
            uglovi[i]*=180./PI;
            i++;
        }
        i=0;
              while(i<brUglova)
           {
                a=((60*(uglovi[i]-(int)(uglovi[i])))-(int)(60*(uglovi[i
```

Table 8: Dataset Options

Dataset	Features	Challenges
Programming Homework Dataset for Plagiarism Detection	Actual Programming Assignments Of Varying Lengths. Reviewed by Professors and Challenge to students	40,000 examples of C/C++ with only about 1,300 plagiarizations
StaQC	Code Snippets in Python From Stack OverFlow. Over 148K Examples)	Annotation with code and text pairs. No verification of plagiarism.
IR-Plag	Canonically mapped based upon type of plagiarism	Small size and only cover introductory programming.
CodeXGlue	Clone Detection (POJ-104) Train 32K/Dev 8K/Test 12K examples spread amounst 64K problems	Mostly small programming problems, not as variable as other programming assignments.

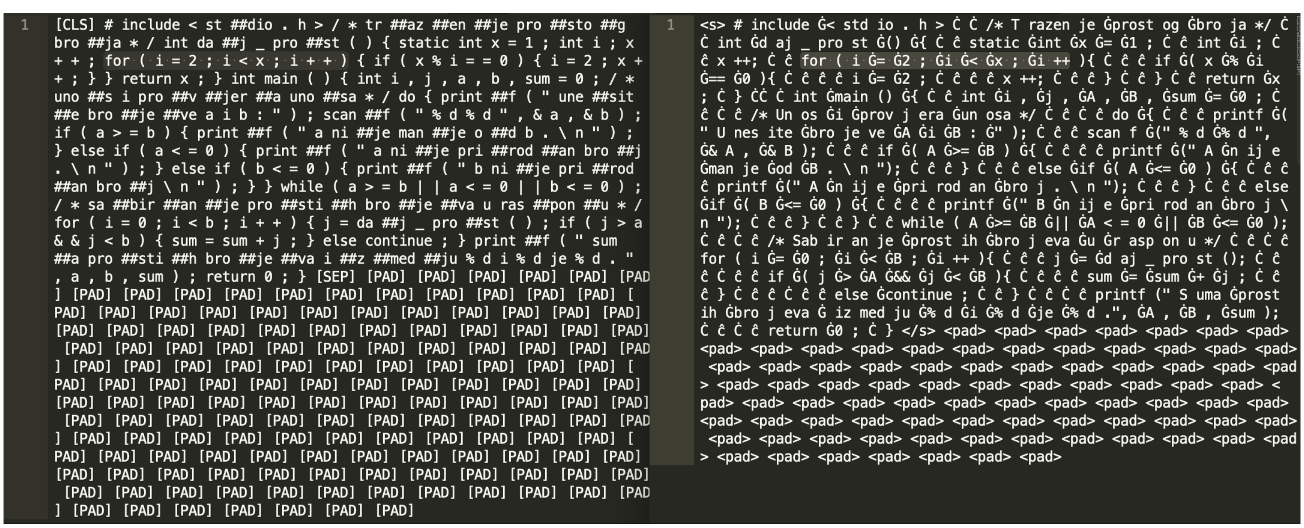
Figure 5: Tokens Before and After ANTLR Preprocessing



```c
#include <stdio.h>

/*Trazenje prostog broja*/

int daj_prost () {
    static int x = 1;
    int i;
    x++;
    for(i = 2; i < x; i++){
        if (x % i == 0){
            i = 2;
            x++;
        }
    }
    return x;
}


int main() {
    int i, j, A, B, sum = 0;

    /*Unos i provjera unosa*/

    do {
        printf (""Unesite brojeve A i B: "");
        scanf (""%d %d"", &A, &B);
        if (A >= B) {
            printf (""A nije manje od B.\n"");
        }
        else if (A <= 0) {
            printf (""A nije prirodan broj.\n"");
        }
        else if (B <= 0) {
            printf (""B nije prirodan broj\n"");
        }
    }
    while(A >= B || A<=0 || B <= 0);

    /*Sabiranje prostih brojeva u rasponu*/

    for(i = 0; i < B; i++){
        j = daj_prost();

        if (j > A && j < B){
            sum = sum + j;
        }

        else continue;
    }

    printf(""Suma prostih brojeva izmedju %d i %d je %d."", A, B, sum);

    return 0;
}
```

```
#include <stdio.h> int daj_prost ( ) { static int x = 1 ; int i ; x ++ ;
for ( i = 2 ; i < x ; i ++ ) { if ( x % i == 0 ) { i = 2 ; x ++ ; } }
return x ; } int main ( ) { int i , j , a , b , sum = 0 ; do { printf (
"unesite brojeve a i b: " ) ; scanf ( "%d %d" , & a , & b ) ; if ( a >= b
) { printf ( "a nije manje od b.\n" ) ; } else if ( a <= 0 ) { printf (
"a nije prirodan broj.\n" ) ; } else if ( b <= 0 ) { printf ( "b nije
prirodan broj\n" ) ; } } while ( a >= b || a <= 0 || b <= 0 ) ; for ( i =
0 ; i < b ; i ++ ) { j = daj_prost ( ) ; if ( j > a && j < b ) { sum =
sum + j ; } else continue ; } printf ( "suma prostih brojeva izmedju %d i
%d je %d." , a , b , sum ) ; return 0 ; }
```

Figure 6: BERT Tokenization vs CodeBERT Tokenization



```
[CLS] # include < st ##dio . h > / * tr ##az ##en ##je pro ##sto ##g
bro ##ja * / int da ##j _ pro ##st ( ) { static int x = 1 ; int i ; x
+ + ; for ( i = 2 ; i < x ; i + + ) { if ( x % i = = 0 ) { i = 2 ; x +
+ ; } } return x ; } int main ( ) { int i , j , a , b , sum = 0 ; / *
uno ##s i pro ##v ##jer ##a uno ##sa * / do { print ##f ( " une ##sit
##e bro ##je ##ve a i b : " ) ; scan ##f ( " % d % d " , & a , & b ) ;
if ( a > = b ) { print ##f ( " a ni ##je man ##je o ##d b . \ n " ) ;
} else if ( a < = 0 ) { print ##f ( " a ni ##je pri ##rod ##an bro ##j
. \ n " ) ; } else if ( b < = 0 ) { print ##f ( " b ni ##je pri ##rod
##an bro ##j \ n " ) ; } } while ( a > = b | | a < = 0 | | b < = 0 ) ;
/ * sa ##bir ##an ##je pro ##sti ##h bro ##je ##va u ras ##pon ##u * /
for ( i = 0 ; i < b ; i + + ) { j = da ##j _ pro ##st ( ) ; if ( j > a
& & j < b ) { sum = sum + j ; } else continue ; } print ##f ( " sum
##a pro ##sti ##h bro ##je ##va i ##z ##med ##ju % d i % d je % d . "
, a , b , sum ) ; return 0 ; } [SEP] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD
] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [
PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
 [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD
] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [
PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
 [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD
] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [
PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
[PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
 [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD
] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD] [PAD]
```

```
<s> # include Ġ< std io . h > Ċ Ċ /* T razen je Ġprost og Ġbro ja */ Ċ
Ċ int Ġd aj _ pro st Ġ() Ġ{ Ċ ĉ static Ġint Ġx Ġ= Ġ1 ; Ċ ĉ int Ġi ; Ċ
ĉ x ++; Ċ ĉ for ( i Ġ= Ġ2 ; Ġi Ġ< Ġx ; Ġi ++ ){ Ċ ĉ ĉ if Ġ( x Ġ% Ġi
Ġ== Ġ0 ){ Ċ ĉ ĉ ĉ i Ġ= Ġ2 ; Ċ ĉ ĉ ĉ x ++; Ċ ĉ ĉ } Ċ ĉ } Ċ ĉ return Ġx
; Ċ } ĊĊ Ċ int Ġmain () Ġ{ Ċ ĉ int Ġi , Ġj , ĠA , ĠB , Ġsum Ġ= Ġ0 ; Ċ
ĉ Ċ ĉ /* Un os Ġi Ġprov j era Ġun osa */ Ċ ĉ Ċ ĉ do Ġ{ Ċ ĉ ĉ printf Ġ(
" U nes ite Ġbro je ve ĠA Ġi ĠB : Ġ" ); Ċ ĉ ĉ scan f Ġ(" % d Ġ% d ",
Ġ& A , Ġ& B ); Ċ ĉ ĉ if Ġ( A Ġ>= ĠB ) Ġ{ Ċ ĉ ĉ ĉ printf Ġ(" A Ġn ij e
Ġman je Ġod ĠB . \ n "); Ċ ĉ ĉ } Ċ ĉ ĉ else Ġif Ġ( A Ġ<= Ġ0 ) Ġ{ Ċ ĉ ĉ
ĉ printf Ġ(" A Ġn ij e Ġpri rod an Ġbro j . \ n "); Ċ ĉ ĉ } Ċ ĉ ĉ else
Ġif Ġ( B Ġ<= Ġ0 ) Ġ{ Ċ ĉ ĉ ĉ printf Ġ(" B Ġn ij e Ġpri rod an Ġbro j \
n "); Ċ ĉ ĉ } Ċ ĉ } Ċ ĉ while ( A Ġ>= ĠB Ġ|| ĠA < = 0 Ġ|| ĠB Ġ<= Ġ0 );
Ċ ĉ Ċ ĉ /* Sab ir an je Ġprost ih Ġbro j eva Ġu Ġr asp on u */ Ċ ĉ Ċ ĉ
for ( i Ġ= Ġ0 ; Ġi Ġ< ĠB ; Ġi ++ ){ Ċ ĉ ĉ j Ġ= Ġd aj _ pro st (); Ċ ĉ
ĉ Ċ ĉ ĉ if Ġ( j Ġ> ĠA Ġ&& Ġj Ġ< ĠB ){ Ċ ĉ ĉ ĉ sum Ġ= Ġsum Ġ+ Ġj ; Ċ ĉ
ĉ } Ċ ĉ ĉ Ċ ĉ ĉ else Ġcontinue ; Ċ ĉ } Ċ ĉ Ċ ĉ printf (" S uma Ġprost
ih Ġbro j eva Ġ iz med ju Ġ% d Ġi Ġ% d Ġje Ġ% d .", ĠA , ĠB , Ġsum );
Ċ ĉ Ċ ĉ return Ġ0 ; Ċ } </s> <pad> <pad> <pad> <pad> <pad> <pad> <pad>
<pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>
 <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad
> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <
pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>
<pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad>
 <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad> <pad
> <pad> <pad> <pad> <pad> <pad> <pad> <pad>
```