Code Clone Detection using Siamese CodeBert

Himanshu Singhal
Virginia Polytechnic and State University
Falls Church, USA
himanshusinghal@vt.edu

Tej Mukeshbhai Patel
Virginia Polytechnic and State University
Falls Church, USA
tejp98@vt.edu

ABSTRACT

Cloning of code refers to the copy and pasting of code without any modifications. Even though this has led to an increase in the productivity of developers, this has often led to bugs and security vulnerabilities introduced in the production code of many firms that are a serious cause of concern. This is why there is widespread interest among many researchers to solve this problem and detect the cloning of code. In this work, we attempt to solve this problem through the use of Siamese networks. First various preprocessing steps were applied on the BigCloneBench dataset such as random sampling and tokenization. Then, We used CodeBert and GraphCodeBert transformers as a transfer learning model and introduce them into the Siamese network architecture and evaluated if that leads to an increase in the performance of the model for Code Clone detection. We also experimented with different loss functions such as Triplet Loss function and Contrastive Loss and tried to gain insights into which works best for this task by evaluating various metrics such as precision, recall, F-score, and accuracy. Upon running the experiments with adam optimizer for 10 epochs, we achieved a F-score of 0.57 with CodeBert and 0.56 with GraphCodeBert.

CCS CONCEPTS

• **Applied Computing** → **Document Management and text processing**; • **Software and its engineering** → **Software creation and management**; • **Computing Methodologies** → **Machine Learning**.

KEYWORDS

code clone detection, neural networks, codebert, graphcodebert, siamese networks, similarity learning

ACM Reference Format:
Himanshu Singhal and Tej Mukeshbhai Patel. 2022. Code Clone Detection using Siamese CodeBert. In *Proceedings of Intro to Deep Learning (CS5814)*. ACM, Falls Church, VA, USA, 9 pages. https://doi.org/20220401.20220401

1 INTRODUCTION

With many recent advancements in the field of Deep Learning and a continuous increase in computational power, neural networks

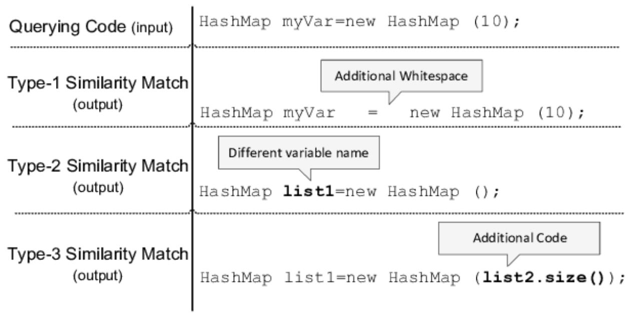

Querying Code (input)
HashMap myVar=new HashMap (10);
Type-1 Similarity Match (output)
Additional Whitespace
HashMap myVar = new HashMap (10);
Type-2 Similarity Match (output)
Different variable name
HashMap list1=new HashMap ();
Type-3 Similarity Match (output)
Additional Code
HashMap list1=new HashMap (list2.size());

Figure 1: Code Clone Detection Types [13]

are increasingly finding many use cases in different fields like Biology, Finance, Automotive, Software, Cybersecurity, etc. In software field, there are many applications of neural networks like code summarization[1][15], source code generation[17][5], bug detection and fixing[11][8], etc. Among these use cases, one of the most popular ones is Code Clone Detection[22][25].

With online collaboration between software engineers continuously increasing with the use of open-source repository code and various websites like StackOverflow, Reddit, Quora, Medium, etc the number of clones is also increasing. Software Engineers by copying and pasting some code from these sources are increasing programming productivity but with this copy-pasting, they might be introducing bugs in their source code. Researchers[3] mainly divide source code clones into four main types:

- Type 1, in this type everything is the same except the number of white spaces and the comments
- Type 2, in this type changes in the name of the variable, type of the variable, function name changes are included
- Type 3, in this type, statements are added, changed, or removed
- Type 4, in this type, there is a semantic similarity between codes

For Code Clone Detection, before passing through the neural network, the source code is converted to a different code representation. These code representations can be a Control Flow Graph (CFG), Abstract Syntax Tree (AST), data flow, bytecode, etc. Each of these representations provides a different abstraction and relationship for source code like AST represents the structure of the source code, CFG provides control flow of the source code using graph representation. [20] provides a comprehensive study on different source code representations. For this project, we would be experimenting with code clone detection using a Siamese neural network.

Siamese neural networks are also known as twin neural networks. Siamese Network[7] is a type of artificial neural network that uses

CS5814, Apr 01, 2022, Falls Church, VA
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-2022-04/01/22…$15.00
https://doi.org/20220401.20220401

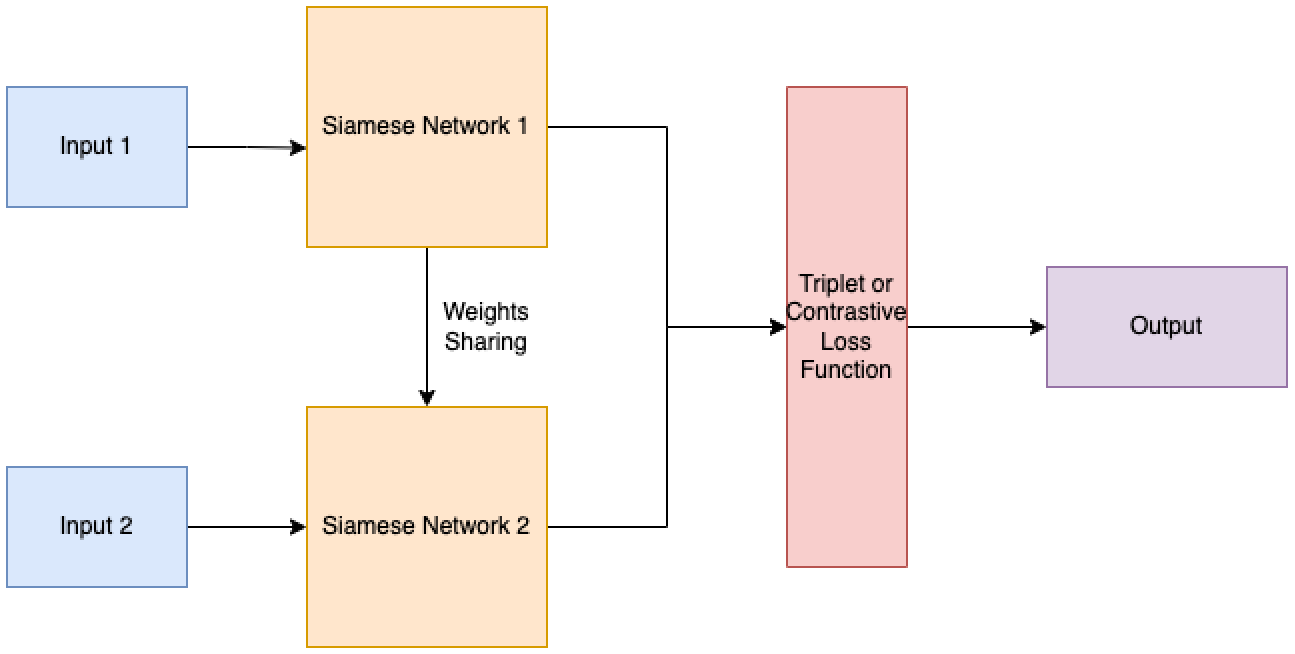


Figure 2: Basic Siamese Network/ Generic Siamese Architecture

the same weights and parameters while working in tandem on two or more different input vectors to compute comparable output vectors. These comparable output vectors are compared on the basis of distance measure and based on that classification is done.

Siamese networks[4] are the same parameters and weights neural networks applied on two or more inputs. Parameters and weights updating is done once two or more inputs are passed through. They are natural tools for comparing entities. Intuitively, a Siamese Network learns to differentiate between inputs, learning their similarity, instead of trying to classify inputs. The loss functions that can be used are Contrastive loss or Triplet loss.

A few advantages of the Siamese Network would be:

- It can learn on less data much better than a normal model.
- As Siamese Network learns on the basis of similarity function, it is good at handling class imbalance as well as it can learn semantic similarity.

A few disadvantages of the Siamese Network would be:

- It takes more time to train a Siamese Network model than a normal model
- The output of the model is an output vector, not probabilities using softmax, so the classification needs to be done on the basis of a distance measure between output vectors.

The figure Fig. 2 shows basic structure of a Siamese Neural Network or a generic Siamese Architecture. As shown at a time there a two or more inputs in a Siamese Network, these simultaneously inputs are the things between which similarity is being learned and compared. As shown in the figure both networks use same weights. And the finally the output of both the networks is feed to Triplet or Contrastive loss function, based on what works best for a given use case. Based on the output of the loss function the backpropagation in the training phase is done.

More details about Siamese Network/Architecture would be discussed in the Models subsection.

Attention[18] is one of the most important concepts in the field of deep learning. Inspired by the human biological systems, it is used to model long-range interaction in neural networks. The attention mechanism was first introduced in the paper by Bahdanau et al.[2] and applied to the task of machine translation. The attention mechanism primarily builds a path between the inputs and the context vectors so as to give more importance to the different parts.

Transformers[21] are the state of the art encoder-decoder models that use the attention mechanism to find the global dependencies between the input and the output tensors provided to the model. They use stacks of self-attention and fully connected layers for the encoder and decoder and map a query and set of key-value pairs to the output. Since they lack recurrence relations, positional embeddings are used to provide the relative position of the tokens in the sequence. In this work, we utilize the power of the transformers to obtain contextual embeddings in the Siamese architecture to solve the problem of code clone detection.

2 LITERATURE SURVEY

There have been a lot of studies and work attempting to solve the problem of code clone detection. These solutions used various methods ranging from abstract syntax trees to complex neural networks. In this section, we look at these some of the existing works.

Jiang et al.[12] uses abstract syntax trees to compute vectors for each code that depict the structural information of the code. These computed vectors are then clustered using Locality Sensitive

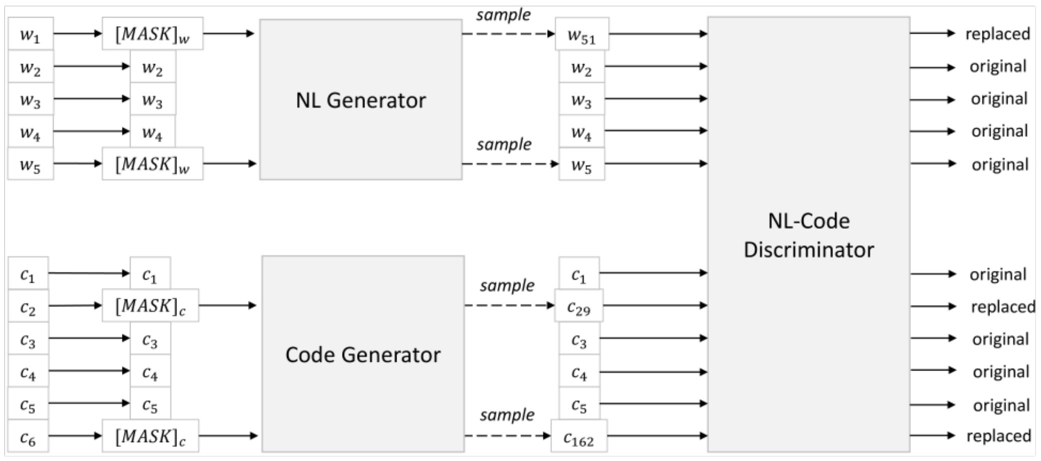


Figure 3: CodeBert architecture [9]

Hashing and similar vectors are found and correspondingly cloned codes. This method was highly scalable, however, was found to give a high number of false negatives and hence gave poor recall and thereby F-scores.

White et al.[25] used deep learning to solve the problem of code clone detection. The authors of the paper use a recurrent neural network to obtain the embeddings for the code at a lexical level as well as obtain vector representations of the code using its syntactical methods such as abstract syntax trees and greedy algorithms. These embeddings and the vector representations obtained are then passed through another RNN to determine whether the codes are a clone or not. However, this method also suffers from the same problems as L. Jiang et al.[12].

Wei et al.[24] transform code into ASTs and then pass them to LSTMs(long short term memory) that help obtain a vector representation for the code. Post this, the representations are encoded into binary hash codes so that similar codes can be identified using the hamming distance between the code hashes. This method worked very well giving an F-score of 0.82.

Zhang et al.[27] state that the works discussed above often use large abstract syntax trees. This particular paper divides these ASTs into smaller trees and creates a sequence of statement vectors. These vectors are then passed through a bidirectional RNN to obtain fragment representations. These representations are then used to perform binary classification and the results from this framework give close to 98% test accuracy.

Wang et al.[22] in their work make use of a graph representation of the programs which are called flow-augmented abstract syntax trees. These FA-ASTs are then passed through graph neural networks and the similarity of code pairs is measured. The approach used by the authors beats the state-of-the-art solutions on both the Google Code Jam and the BigCloneBench datasets.

Yu et al.[26] propose an approach using tree-based convolution to detect semantic clones in order to capture both structural and lexical information from the code. For this, they introduce an embedding technique called position-aware character embedding that considers a token as a position-weighted combination of character one-hot embeddings. This approach is highly computationally efficient along with beating the SOTA on the code clone detection datasets.

Feng et al.[9] uses a Transformer-based architecture that not only is used to obtain an effective code representation but also incorporates the task of token detection. The network architecture of CodeBert uses a multi-layer bidirectional transformer. They use both unimodal and bimodal data that helps them provide better tokens as well as generators. The architecture of CodeBert is the same as that of RoBERTa-Base[16]. The CodeBert model produces contextual representation for each token as well as the representation of CLS. The model was primarily trained for 2 objectives namely Masked Language Modeling and Replaced Token Detection. The trained model was then tested on multiple tasks such as Natural Language Code Search, NL-PL Probing, Code Documentation generation and generalization to the languages not trained upon. The architecture of the CodeBert model is shown in the Figure Fig.3. Given the wide nature of tasks it has been observed to perform well, we hypothesize that the embeddings produced by the

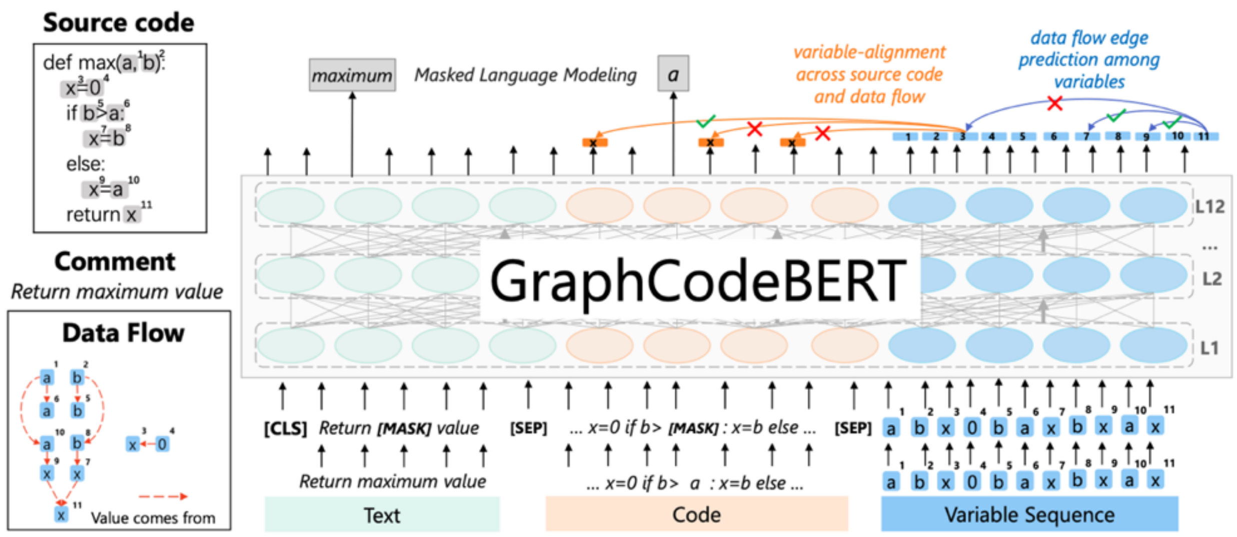


Figure 4: GraphCodeBert architecture [10]

CodeBert model should produce good results in a Siamese network architecture for the task of Code Clone detection.

The work of Gaya et al.[10] proposes an approach to use pre-trained models that use the structure of the code. The GraphCodeBert model uses data flow that represents code in a semantic-structural manner and accounts for the relationship between the variables. The data flow is a dependency graph that represents the relationship between variables. A transformer based architecture based upon data flow is less complex than one based on abstract syntax trees and hence performs better and is more efficient. The figure Fig. 4 demonstrates the architecture of the GraphCodeBert model. It shows a bidirectional transformer that not only uses the source code as input but also the paired comments to train the model that helps it to be used for natural language code search tasks. The graph structure is incorporated into the code by using a graph-guided masked attention. The tasks for pretraining the model include Masked Language Modeling, Edge Prediction and Node Alignment. The GraphCodeBert model was experimented with Natural Language Code Search, Code Clone Detection, Code Translation and Code Refinement and was shown to perform excellently on all these tasks.

3 METHODOLOGY

This section talks about the dataset that is used, the model architecture, and the evaluation techniques that will be performed for judging the model performance.

3.1 Dataset

The dataset used for this project is BigCloneBench[19]. The code in the dataset is in Java programming language. The whole project would basically have four files, mentioned below.

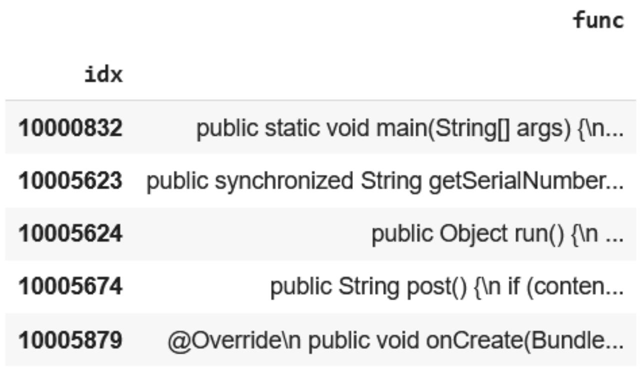
- data.jsonl, this file is of jsonlines format. Each row represents a json object with two keys, func and idx where func key has implemetation/code of a function and idx key represents index/id/unique identifier of that function
- train.txt, each line is of the format: idx1-idx2-label
- validate.txt, each line is of the format: idx1-idx2-label
- test.txt, each line is of the format: idx1-idx2-label

Dataset size is mentioned in the Table 1.

Table 1: No. of Samples

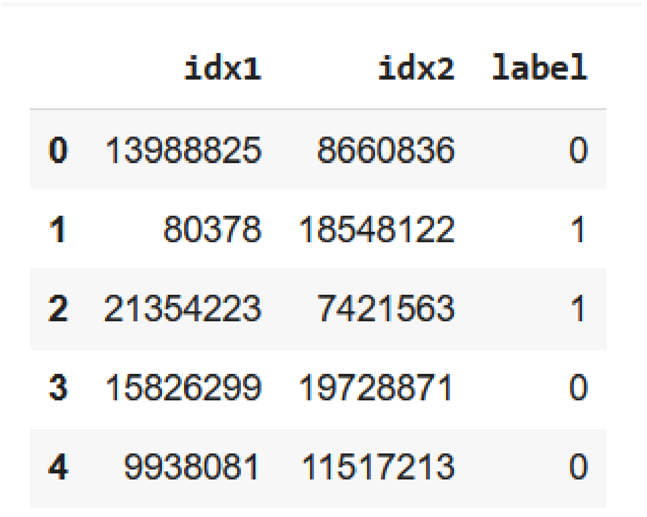
File Name	Samples
train.txt	901028
validate.txt	415416
test.txt	415416

We would be downsampling from the above samples based on the computational power we have access to. We are using approximately 70/15/15 train/validate/test combination.



idx	func
10000832	public static void main(String[] args) {\n...
10005623	public synchronized String getSerialNumber...
10005624	public Object run() {\n ...
10005674	public String post() {\n if (conten...
10005879	@Override\n public void onCreate(Bundle...

Figure 5: Sample code examples



	idx1	idx2	label
0	13988825	8660836	0
1	80378	18548122	1
2	21354223	7421563	1
3	15826299	19728871	0
4	9938081	11517213	0

Figure 6: Sample training data for code clone detection

3.2 Data Preprocessing

First, the data.jsonl file is parsed to obtain a dataframe of the mappings between the code indices/ids and the codes. Similarly, the train.txt, val.txt and test.txt is parsed to obtain the pair of codes and the labels which tell whether the codes are a clone or not. Next step in the data preprocessing was down sampling from the whole dataset due to computational power restrictions. This down-sampling of dataset was done using random sampling. The dataset was down-sampled to below mentioned dimensions which is approximately 70/15/15 train/validation/test split.

Table 2: Size of data used

Type	No. of Samples
train	5000
validate	1000
test	1000

There two labels 0 and 1, 1 shows equivalence between the two functions. Label distribution for all training data is shown below. The distribution of the training data is balanced and is not skewed towards a particular class. The label distribution of validation and test data is skewed, and should help us evaluate the performance of the model better and observe if it has learnt the important features to distinguish between the labels.

Table 3: Label distribution of the training data

Label	No. of Samples
0	2509
1	2491

Table 4: Label distribution of the validation data

Label	No. of Samples
0	866
1	134

Table 5: Label distribution of the test data

Label	No. of Samples
0	856
1	144

The next step used was tokenization. The dataframe which holds the mappings of id and code is not useful currently, as we cannot pass the code through the model directly, so the code columns of the dataframe is tokenized using CodeBert[9] tokenizer model and stored in the dataframe. CodeBert tokenizer model is pre-trained model which helps generate a representation of code that can be further used for a specific use-case.

After tokenization is done, the length/dimension of each tokenized code is calculated and stored in the dataframe. The length of different tokenized codes would be different, so a fixed sequence length is chosen based on the 0.7 percentile of tokenized code length. For this project, the chosen max sequence length was 724. So, the tokenized codes are truncated if their length is greater than the chosen sequence length or padded if their length is less than the chosen sequence length. After this process, all the tokenized code would be of same length/dimension which is 724, now they can be directly passed through the model.

Once the whole tokenization process of the code is performed, the id of each token is extracted and stored. These token ids are then used as an input to the Siamese models.

3.3 Models

The figure Fig. 7 describes the model architecture that we have used for our project. The network uses two CodeBert transformers that take in the two codes to be compared as the inputs and then produce a representation of the codes as a feature vector. The CodeBert transformers are used as pre-trained models and used for transfer learning in our approach. We will however unfreeze the last few layers of the model so that the weights of the transformers are updated to learn the similarity between the codes. The CodeBert output feature has been taken of dimension 200. The feature vectors obtained from CodeBert are then passed through 2 fully connected layers and the ReLU activation function is applied to it after the first linear layer of size (100, 50). The second linear has a dimension of (50, 2). We also tried GraphCodeBert instead of CodeBert as the pretrained model for obtaining the contextual embeddings. The networks that both the codes are passed to are identical. The final representations obtained from both the subnetworks are then compared using a Triplet Loss as referenced by Eq. 1 or a Contrastive Loss function depicted in Eq. 2.

$$L(A, P, N) = max(||f(A) - f(P)||^2 - ||f(A) - f(N)||^2 + \alpha, 0) \quad (1)$$

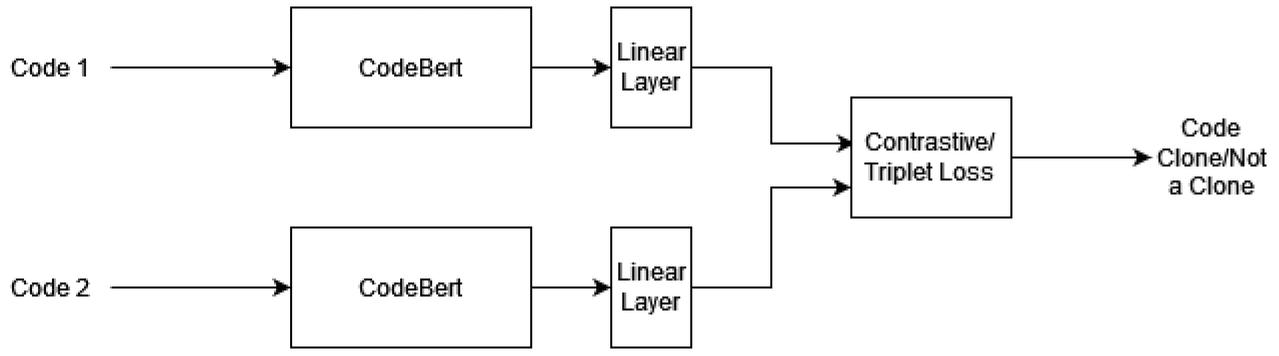
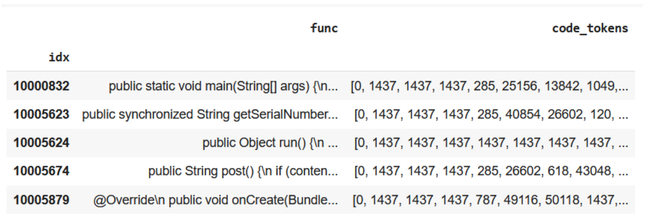


Figure 7: Siamese CodeBert architecture



	func	code_tokens
idx		
10000832	public static void main(String[] args) {\n...	[0, 1437, 1437, 1437, 285, 25156, 13842, 1049,...
10005623	public synchronized String getSerialNumber...	[0, 1437, 1437, 1437, 285, 40854, 26602, 120, ...
10005624	public Object run() {\n ...	[0, 1437, 1437, 1437, 1437, 1437, 1437, 1437, ...
10005674	public String post() {\n if (conten...	[0, 1437, 1437, 1437, 285, 26602, 618, 43048, ...
10005879	@Override\n public void onCreate(Bundle...	[0, 1437, 1437, 1437, 787, 49116, 50118, 1437,...

Figure 8: Sample token ids generated post tokenization

$$L = (1 - Y)\frac{1}{2}(D_w)^2 + (Y)\frac{1}{2}max(0, m - D_w)^2 \tag{2}$$

$$D_w = \sqrt{G_w(X_1) - G_w(X_2)^2} \tag{3}$$

Triplet loss is a type loss function where a reference/baseline input (anchor) is compared to a matching/true input (positive) and a non-matching/false input (negative). This distance from the baseline to the positive is minimized while the distance from the anchor to the negative input is maximized, as shown in Equation 1

Contrastive Loss function is a distance-based loss function. This loss function helps learning weights/embeddings in such a way that when two similar inputs are used then they should have low Euclidean distance and if two not similar inputs are used then the Euclidean distance between them should be high.

The use of these loss functions should ensure that a semantic similarity function is learned between the two representative vectors and the model can predict whether the codes are clones or not.

4 EVALUATION AND RESULTS

For this project, we tried different tokenizer/transfer learning models like CodeBert and GraphCodeBert with a Siamese model to see how well/bad their performance is on Code Clone Detection task. We used the validation set to fine-tune the hyper-parameters like number of layers, number of neurons, learning rate, number of epochs, etc of the above models. We used the test set that we pre-processed to evaluate their performance on unseen data.

We are doing a classification task using the above model, but since our model is based on Siamese Network, we are not using the softmax function at the end to classify it into two classes. Instead, we are comparing the output vectors(model output) of both the functions/tokenized code using distance measure like Euclidean Distance, and based on this distance measure classification is done.

We started our experiments with using CodeBert as a transformer model which is connected to a Siamese neural network of two hidden linear layers with 100 and 50 hidden neurons respectively. The optimizer that was used for this experiment was Adam optimizer[14]. Along with Adam optimizer, the hyperparameters used were learning rate was set to 0.001 and weight decay was set to 0.0005. Weight Decay was used to help the model generalize better. The figure Fig. 9 shows the training/validation loss of the plotted against the number of epochs.

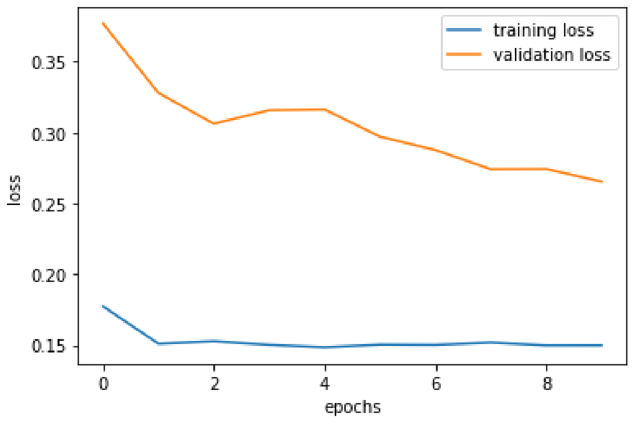


Figure 9: Training/Validation Loss Vs Epochs for CodeBert - Siamese Model

As we can notice from the image the training loss seems to decrease a little bit in the first two epochs but after that it almost remains constant. This indicates that model hasn't learned much from the training step, this might occur due to a number of reasons - less data, less number of epochs, less number of hidden layer, not right learning rate, etc.

We tried to increase the training data from 5K to around 7K, but there was no significant change in the training loss converging in that case as well. Further increase in training data was causing significant memory exceeds issues. We tried a few learning rates-weight decay permutations and changing the loss function to Triplet Loss, but that did not help as well. Some things that we thought would help the model learn better are making the model deeper by adding more hidden layers to it. Also the increase in number of epochs would also have helped the model, each time we tried to do these changes, time taken to train increased exponentially and computational power would get exhausted, it would result into error.

The validation loss plot against number of epochs shown in orange line in the graph seems to decrease with number of epochs. But it was still showing more error than training data and it also does not looked fully converged.

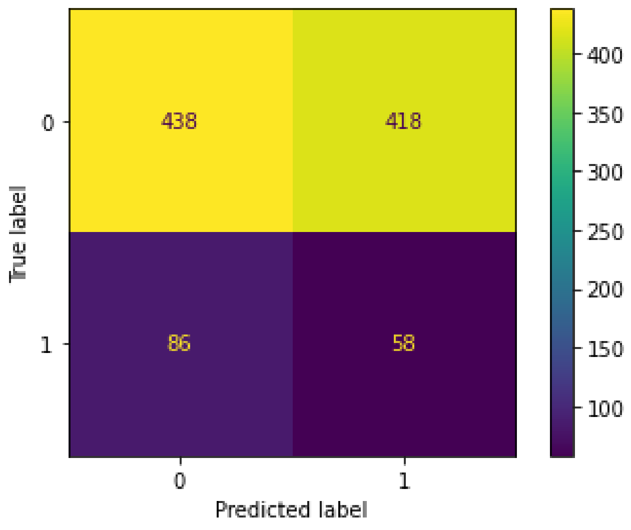


Figure 10: Confusion Matrix for CodeBert

The figure Fig. 10 shows confusion matrix of performance of the CodeBert-Siamese Model on test data. As we interpreted from the training loss versus number of epochs plot that the model has not learned much is made more apparent with this confusion matrix. The Accuracy on test data for the model is 0.49, which is not good.

Below table shows precision, recall and F1-score, precision score is better than other parameters but that can be attributed to class imbalance. Other parameters like Recall and F1-Score say the same story that model is not very good.

Table 6: Classification report of the Siamese Codebert

LABEL	PRECISION	RECALL	F1-SCORE
0	0.84	0.51	0.63
1	0.12	0.40	0.19
Avg	0.73	0.50	0.57

With this not very good model, we wanted to try something that would help the model learn better but without running into computational power and memory issues. So we decided to try changing the transformer we were using from CodeBert to GraphCodeBert. So the architecture of the other parts of the model remains same along with the hyperparameters only thing that changes is CodeBert is replaced with GraphCodeBert.

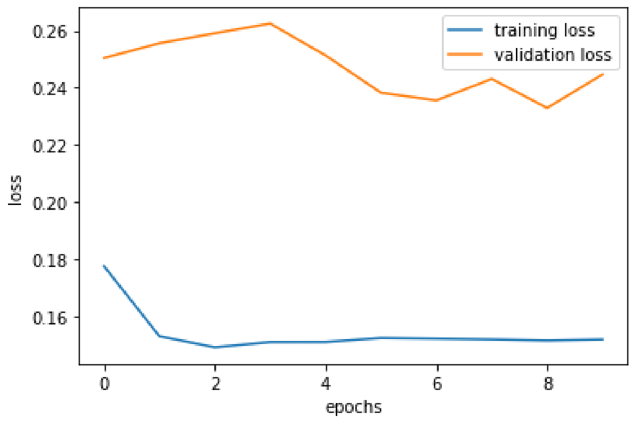


Figure 11: Training/Validation Loss Vs Epochs for GraphCodeBert - Siamese Model

The figure Fig. 11 shows the training loss and validation loss versus the number of epochs. The training loss shown in blue looks like CodeBert-Siamese model training loss plot. It decreases between 0-2 epochs and than stays appropriately same, the training loss neither increases nor decreases with number of epochs after second epoch. The validation loss plot does not give much useful insight as well, it moves between an interval.

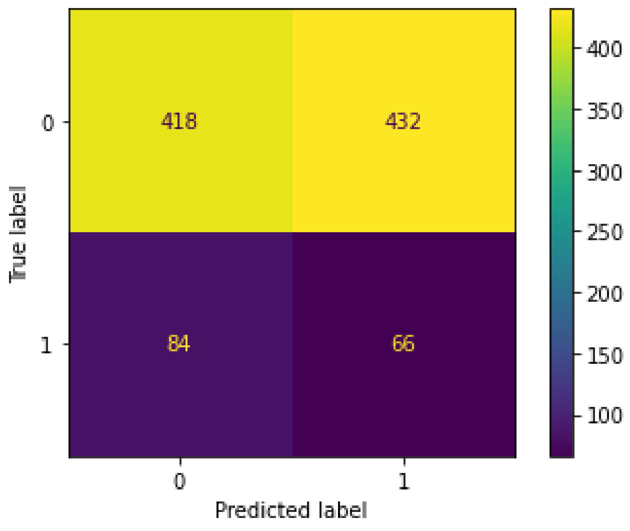


Figure 12: Confusion Matrix for GraphCodeBert

The figure Fig. 12 shows confusion matrix generated by GraphCodeBert - Siamese Model on test data. The Accuracy of this model is approximately 0.48. Confusion matrix shows that model has ways to go to learn to classify Code Clone Detection.

Table 7: Classification report of the Siamese GraphCodebert

LABEL	PRECISION	RECALL	F1-SCORE
0	0.83	0.49	0.62
1	0.13	0.44	0.20
Avg	0.73	0.48	0.56

The table No. 7 shows precision, recall and F1-score. The precision score of 0 class/label is misleading, it is the result of class imbalance not the model's performance. If we take all the parameters into consideration like accuracy, recall, F1-score then it is quite evident that the performance of the model is not good. The learning of the model has a long way to go and we believe that training with a deeper network using more computational resources would solve the issue.

5 LESSONS LEARNED

Working on this project, we learned about Siamese network architecture and how one shot learning can be used in classification tasks with the use of minimal data. We also learned about various aspects of a deep learning project such as preprocessing, visualization, training and evaluation. We also understood the architectures used in transformers and gained practical experience working with the state of the art models like CodeBert and GraphCodeBert.

6 BROADER IMPACTS

This work aims to see if the use of CodeBert and GraphCodeBert transformers in a Siamese Network architecture helps improve the performance and beat the state of the art in solving the problem of Code Clone detection. We will also tried to compare the use of 2 different loss functions namely Triplet and Contrastive losses on the architecture and observe which performs better on the dataset using the model architecture. However, we failed at implementing the Triplet loss function. A better performance by the proposed models would have lead to obtaining better semantic representations for code and could be used by researchers in the field to solve more complicated problems. We would also have liked to experiment with other transfer learning models such as CodeT5[23] and OpenAI Codex[6] in our architecture, which, however, we were not able to experiment with them due to time and resource constraints.

7 CONCLUSION

In summary, we started our project by selecting to tackle Code Clone Detection using Siamese Network and wanting to know how Siamese Network impacts the result when used with CodeBert and GraphCodeBert. First we read many research papers and did research on the nuances of various topics of our project like Code Clonce Detection, CodeBert, Siamese Network, etc. The dataset used was provided by BigCloneBench, the dataset was downsampled using random sampling and tokenized using CodeBert/GraphCodeBert to be fed into the model. The model architecture used was CodeBert/GraphCodeBert as a transformer and then Siamese Network of two fully connected hidden layers. Different combination of hyperparameters, loss function, etc were tried while training to help improve the model performance. Lastly, various types of evaluation was done on the model using test data. While performing the experimentation, we achieved poor results using both the transformer models, however, this can be attributed to the less training and data used due to a lack of computational resources.

8 WORK DISTRIBUTION

The goal for both of us for this project was to learn as much as we can about every step of a end-to-end deep-learning project, so with this goal in mind for presentation slides and report we decided to divide each thing equally. If Tej is handling introduction part in report then Himanshu would have handled the Problem Statement part in the presentation slides, and if Tej is handling presentation slides part for data preprocessing then Himanshu would have handled the data preprocessing part of the report, so that each of us would get to learn about all of the aspects of our project.

Regarding code, data finding, gathering each files and loading was done by Tej. Data preprocessing steps like downsampling, tokenization, etc and loading the data into Pytorch Dataloader were done by Himanshu. Siamese Network class, Contrastive loss function class, Triplet loss function was done by Tej. Himanshu handled the training and evaluation part of the project which is generating various plots, statistics from the evaluation part, etc. So, coding part was also equally divided between both Tej and Himanshu.

REFERENCES

[1] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. 2015. Suggesting Accurate Method and Class Names. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering* (Bergamo, Italy) *(ESEC/FSE 2015)*. Association for Computing Machinery, New York, NY, USA, 38–49. https://doi.org/10.1145/2786805.2786849

[2] Dzmitry Bahdanau, Kyunghyun Cho, Universite De Montreal, Yoshua Bengio, and Universite De Montreal. 2014. Neural machine translation by jointly learning to align and translate. arXiv preprint arXiv:1409.0473.

[3] Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. 2007. Comparison and Evaluation of Clone Detection Tools. *IEEE Transactions on Software Engineering* 33, 9 (2007), 577–591. https://doi.org/10.1109/TSE.2007.70725

[4] JANE BROMLEY, JAMES W. BENTZ, LÉON BOTTOU, ISABELLE GUYON, YANN LECUN, CLIFF MOORE, EDUARD SÄCKINGER, and ROOPAK SHAH. 1993. SIGNATURE VERIFICATION USING A "SIAMESE" TIME DELAY NEURAL NETWORK. *International Journal of Pattern Recognition and Artificial Intelligence* 07, 04 (1993), 669–688. https://doi.org/10.1142/S0218001493000339 arXiv:https://doi.org/10.1142/S0218001493000339

[5] Chunyang Chen, Zhenchang Xing, Yang Liu, and Kent Ong Long Xiong. 2021. Mining Likely Analogical APIs Across Third-Party Libraries via Large-Scale Unsupervised API Semantics Embedding. *IEEE Transactions on Software Engineering* 47, 3 (2021), 432–447. https://doi.org/10.1109/TSE.2019.2896123

[6] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Josh Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. https://doi.org/10.48550/ARXIV.2107.03374

[7] Davide Chicco. 2021. *Siamese Neural Networks: An Overview.* Springer US, New York, NY, 73–94. https://doi.org/10.1007/978-1-0716-0826-5_3

[8] Jayati Deshmukh, K. M. Annervaz, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. 2017. Towards Accurate Duplicate Bug Retrieval Using Deep

Learning Techniques. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 115–124. https://doi.org/10.1109/ICSME.2017.69

[9] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.18653/v1/2020.findings-emnlp.139

[10] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang (), and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv preprint.

[11] Rahul Gupta, Aditya Kanade, and Shirish Shevade. 2019. Deep Reinforcement Learning for Syntactic Error Repair in Student Programs. *Proceedings of the AAAI Conference on Artificial Intelligence* 33, 01 (Jul. 2019), 930–937. https://doi.org/10.1609/aaai.v33i01.3301930

[12] Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

[13] Iman Keivanloo, Christopher Forbes, and Juergen Rilling. 2012. Similarity search plug-in: Clone detection meets internet-scale code search. In *2012 4th International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation (SUITE)*. 21–22. https://doi.org/10.1109/SUITE.2012.6225474

[14] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. *CoRR* abs/1412.6980 (2015).

[15] Alexander LeClair, Siyuan Jiang, and Collin McMillan. 2019. A Neural Model for Generating Natural Language Summaries of Program Subroutines. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 795–806. https://doi.org/10.1109/ICSE.2019.00087

[16] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. https://doi.org/10.48550/ARXIV.1907.11692

[17] Vijayaraghavan Murali, Letao Qi, Swarat Chaudhuri, and Chris Jermaine. 2018. Neural Sketch Learning for Conditional Program Generation. In *International Conference on Learning Representations*. https://openreview.net/forum?id=HkfXMz-Ab

[18] Zhaoyang Niu, Guoqiang Zhong, and Hui Yu. 2021. A review on the attention mechanism of deep learning. *Neurocomputing* 452 (2021), 48–62. https://doi.org/10.1016/j.neucom.2021.03.091

[19] Jeffrey Svajlenko, Judith F. Islam, Iman Keivanloo, Chanchal K. Roy, and Mohammad Mamun Mia. 2014. Towards a Big Data Curated Benchmark of Inter-project Code Clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. 476–480. https://doi.org/10.1109/ICSME.2014.77

[20] Michele Tufano, Cody Watson, Gabriele Bavota, Massimiliano Di Penta, Martin White, and Denys Poshyvanyk. 2018. Deep Learning Similarities from Different Representations of Source Code. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*. 542–553.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is All You Need. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 6000–6010.

[22] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 2020. Detecting Code Clones with Graph Neural Network and Flow-Augmented Abstract Syntax Tree. In *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 261–271. https://doi.org/10.1109/SANER48275.2020.9054857

[23] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. https://doi.org/10.48550/ARXIV.2109.00859

[24] Hui-Hui Wei and Ming Li. 2017. Supervised Deep Features for Software Functional Clone Detection by Exploiting Lexical and Syntactical Information in Source Code. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence* (Melbourne, Australia) *(IJCAI'17)*. AAAI Press, 3034–3040.

[25] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep learning code fragments for code clone detection. In *2016 31st IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 87–98.

[26] Hao Yu, Wing Lam, Long Chen, Ge Li, Tao Xie, and Qianxiang Wang. 2019. Neural Detection of Semantic Code Clones Via Tree-Based Convolution. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. 70–80. https://doi.org/10.1109/ICPC.2019.00021

[27] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A Novel Neural Source Code Representation Based on Abstract Syntax Tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 783–794. https://doi.org/10.1109/ICSE.2019.00086