Code Clone Detection

Siddharth Yadav, Aamir T. Ahmad
siddharth16268,aamir16001@iiitd.ac.in
IIIT-Delhi

ABSTRACT

Our problem statement is to find if two given code fragments are a clone or not using Tree and Graph-based Neural Networks. There, it's a classification problem and we treat it as a graph/tree classification/comparison task.

1 INTRODUCTION

Clone Detection is an important problem in the Software Engineering community. One of the approaches of writing software is to copy-paste relevant snippets followed by changing a few things to meet our own demands. These code copies are usually syntactically different, but functionally the same. In large code-base, developers often end up writing similar functions multiple times unknowingly. Code reuse is a very important part of software engineering but it leads to the problem of code duplication due to the copy-paste working flow of developers. Code duplication is a major problem as it makes updating/maintaining/refactoring a code-base incredibly tedious and hard. To rectify this problem, we need methods that can identify functions that do similar things. Identifying clones can be beneficial in finding out malicious functions by comparing them with a set of existing similar functions. It's also used in detecting plagiarism an finding out cases of copyright infringements.
Previous works deal with usually four types of clones. Look at Figure below for clarity.

(1) Type-1 is exact same code except for variations in comments and layout.
(2) Type-2 is exact same code except for different identifier names and literal values plus Type-1 differences.
(3) Type-3 is syntactically similar code snippets that differ at the statement level(insertion, deletion, and modification of statements w.r.t each other) plus Type-2 difference.
(4) Type-4 is syntactically dissimilar code snippets that implement the same functionality.

Source code	Type-1 clone	Type-2 clone
int main() { int x = 1; int y = x + 5; return y; }	int main() { int x = 1; int y = x + 5; return y;//output }	int func2() { int p = 1; int q = p + 5; return q; }

Type-3 clone	Type-4 clone
int main() { int s = 1; int t = s + 5; t = t/++s; return t; }	int func4() { int n = 5; return ++n; }

Figure 1: Types of Code Clones

A lot of research has been done in the domain of clone detection. Some broad categories of techniques are a text-based comparison of code fragments, token-wise comparison of code fragments, comparison of AST of code, and comparison of program dependencies graph and control flow graphs. All these approaches require special features engineering followed by generic algorithms. In this project, we focus on learning algorithms that learn from examples rather than having to require special feature engineering.

2 METHODS

2.1 Treating Code as Graph

Source code is very structured data and they follow a very strict grammar(Context Free Grammar). Code is often represented in the form of Abstract Syntax Tree(AST). AST can be treated as Directed Graph. Each node is this tree/graph is either a syntax node(the node belonging to non-terminal sets in the CFG) or token node(the node belonging to the terminal set in the CFG) and the edges describe the relationship between the nodes. The edges types belong to the relations set of the CFG. It's possible to introduce extra edges in the graph using control flow dependencies or the data flow model of the program. We can also add edges to represent what is the next token, what nodes this value depends on(kind of dataflow model), etc. All this information can be obtained from the static analysis of the program.

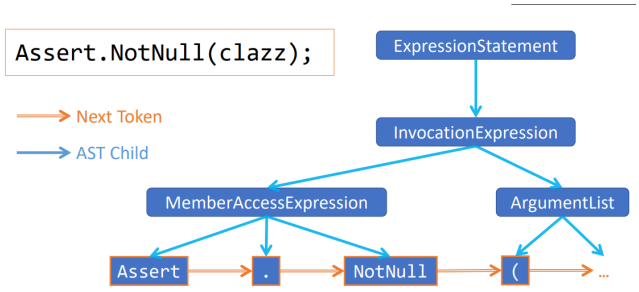

Figure 2: In the given Figure the AST is a part of the statement "Assert.NotNull(), " where blue rounded boxes are syntax nodes, blue rectangular boxes with orange edges are syntax tokens, blue edges child edges and orange edges NextToken edges.

2.2 Methods 1 - LSTM/CNN-LSTM:

We use Node2Vec method for source code called, Code2Vec, to get embeddings for each token node(present at the terminal of the AST) in our graph. Then these embeddings for token nodes can be passed through an LSTM/CNN-LSTM in order of their occurrence in the original source code to get a single vector for a given code fragment. These vector representations can be compared using Siamese Network to get the final score. The final table shows the

difference in result when using one-hot encoding vs Code2Vec encoding for the ast nodes.

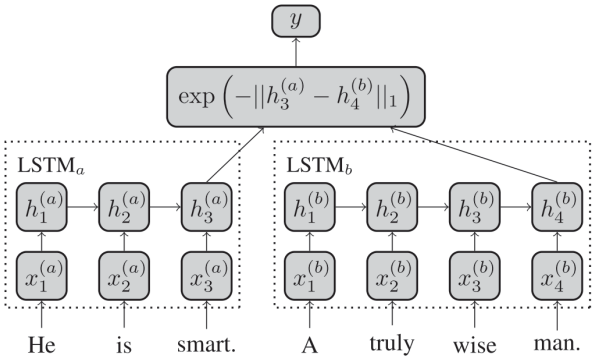

Figure 3: Siamese RNN architecture.

2.3 Method 2 - Graph Convolution:

Graph convolution is a technique inspired from convolution neural networks. Similar to the previous method, we use Code2Vec embedding for the ast nodes. Let us call this initial set of features X. Then we propagate these features through the network using the Graph Convolution Network (GCN) Propagation Rule :

$$H^{(l+1)} = \sigma(D^{-1/2}AD^{-1/2}H^{(l)}W^{l})$$

where,
H $^{(l+1)}$ - Hidden features at next layer
D- Degree Matrix of A
A - Adjacency Matrix with Self loops
H$^{(l)}$ - Hidden features of current layer
W - Weight matrix for current layer
$\sigma - Non - linear activation function(Relu)$
Note : H^0 will be X (initial representation of node)

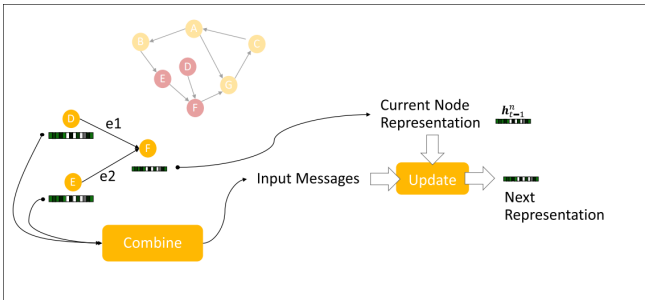

Figure 4: GCN Propagation Rule Diagram

We saw that graph convolution alone wasn't sufficient in improving accuracy. Instead we decreased our accuracy by using only Graph Convolution. So we added Top K pooling layers which showed real progress and improved accuracy.

2.4 Method 3 - Graph Convolution With Top-K Pooling:

Same as the previous method, graph pooling takes inspiration from pooling layers of CNNs. Top-k Pooling utilizes a trainable projection vector p$^{(l)}$R^f and then takes the top-k indices of the projection of X$^{(l)}$p$^{(l)}$ and the corresponding edges in A$^{(l)}$. Here X$^{(l)}$ becomes H$^{(l)}$ for future layers. Below is a figure showing the transformation due to pooling.

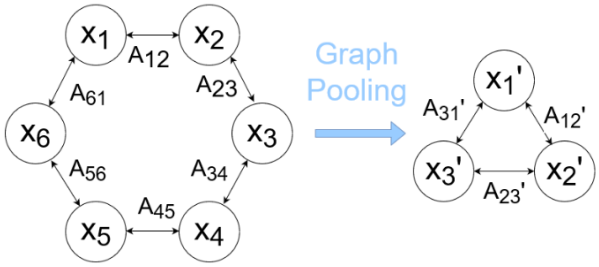

Figure 5: Effect of pooling on a graph with k=3. A new feature vector and adjacency matrix is created for the new graph

While the accuracy jumped back up, we found it was still lacking, so we implemented different variations of ast compositions to be passed to the model

2.5 Method 4 - Variations of AST composing passed to the Network

To improve accuracy tried to change the way we were composing and passing our AST to the model. Let us take a simple program adding one to any given integer. The two variations of this program in AST form can be seen below Fig 6 and 7.

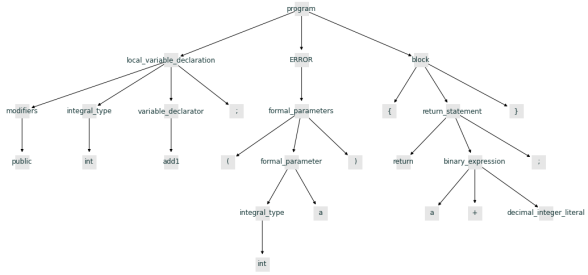

Figure 6: Program 1 : Adding 1 by doing addition (a+=1)

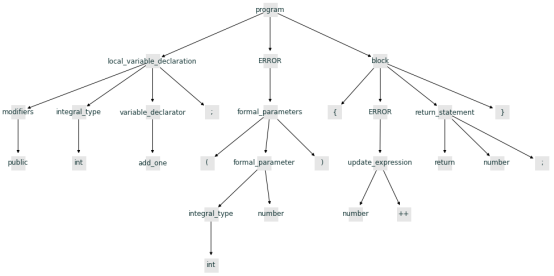

Figure 7: Program 2 : Adding 1 with incremental operator (a++)

There are three ways we have composed the AST. The first method is AST merged by tokes and their location (Fig 8). The second method is based on AST merged by their structure (Fig 9). And the third method is passing disjoint ASTs (Fig 10). The appendix depecits overlapping AST for two real world programs.

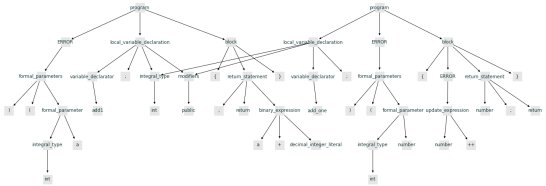

Figure 8: AST merged by tokes and their location

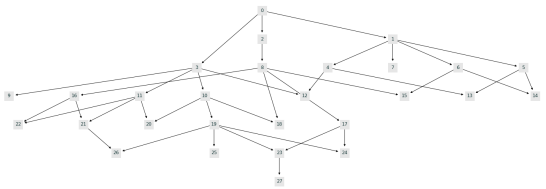

Figure 9: AST merged by their structure

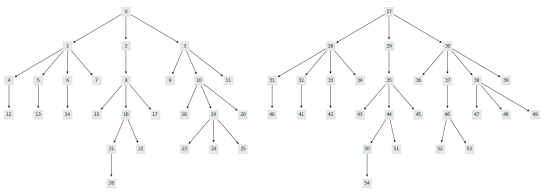

Figure 10: Disjoint AST

We were using Code2Vec embeddings up till this point but weren't getting any significant improvements. So we decided to train our own custom embeddings using FastText embeddings, which showed great improvement. We then combined all the previous knowledge and put it together in a Siamese Network

2.6 Method 5 - Siamese Graph Convolution

For this model, we initially tried with Code2Vec embeddings but failed due to incorrect parameter initialization by Code2Vec which lead to a lot of OOV (Out-of-Vocabulary) embeddings. Then we switched to our custom embeddings made using FastText. The Siamese Network included three repetitions of Graph Convolution and Top-K Pooling followed by three fully connected layers.

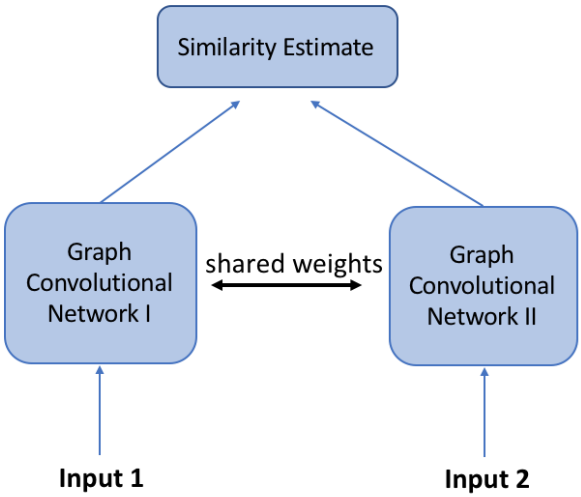

Figure 11: Siamese Graph Convolution Network

The resultant model produced the best test accuracy from all the models trained.

3 DATASET:

We will be using Big Clone Bench(BCB) for training and evaluating our model. Big Clone Bench is a collection of over 8 million validated clones. BCB consists of all the four types of clones, We assign a similarity of 1 to Type-1 and Type-2. Strongly Type-3 is assigned similarity in the range [0.7, 1). Moderately Type-3 is assigned [0.5, 0.7). Type-4 is assigned [0, 0.5).

Pre-processing steps included constructing Code2Vec embedding for dataset, implementing TreeSitter funtions for getting AST from code at any level and also FastText Embedding for dataset. We have trained fasttext embeddings on the inorder ordering of the AST because around 50% unique tokens were OOV when using code2vec

4 RESULTS:

These are the results obtained from the all the models we have implemented. RNN (LSTM) showed quite a high level of accuracy.

Model	Embedding	Accuracy
RNN	Normal	0.83
RNN	Code2Vec	0.86
GraphConv - TopKPooling*	Code2Vec	0.57
GraphConv (AST merged by tokes and their location)	Code2Vec	0.84
GraphConv (AST merged by their structure)	FastText	0.84
GraphConv (Disjoint AST)	FastText	0.87
Siamese of GraphConv	**FastText**	**0.90**

Figure 12: Results obtained for different architectures

Accuracy dropped when only Graph Convolution was used but went back up with Top-K Pooling Layers. AST arrangement did not affect the accuracy much but Disjoint AST outperformed merged AST structure possibly by learning their hidden representations which were expressed better in Disjoint form than merged form. Siamese Networks which are great at comparisons outperformed all the other models we implemented and produced an accuracy of 0.897.

REFERENCES

[1] Miltiadis Allamanis et al, *LaTeX: LEARNING TO REPRESENT PROGRAMS WITH GRAPHS.* https://arxiv.org/pdf/1711.00740.pdf
[2] Arseny Zorin et al, *LaTeX: Recurrent Neural Network for Code Clone Detection.* https://seim-conf.org/media/materials/2018/proceedings/SEIM-2018ShortPapers.pdfpage=48
[3] Yusuke Shido et al, *LaTeX: Automatic Source Code Summarization with Extended Tree-LSTM.* https://arxiv.org/pdf/1906.08094.pdf
[4] Miltos Allamanis et al, *LaTeX: Code Duplication and Machine Learning Models of Code* . https://miltos.allamanis.com/files/slides/2019ml4se.pdf
[5] Miltos Allamanis et al, *LaTeX: Understanding Source Code with Deep Learning* . https://miltos.allamanis.com/files/slides/2019fosdem.pdf
[6] Jian Zhang et al, *LaTeX: A Novel Neural Source Code Representation basedon Abstract Syntax Tree.* http://xuwang.tech/paper/astnnicse2019.pdf
[7] Yujia Li et al, *LaTeX: Graph Matching Networks for Learning the Similarity of Graph Structured Objects.* https://arxiv.org/pdf/1904.12787.pdf
[8] Thomas Kipf, *LaTeX: Graph Convolutional Networks.* https://tkipf.github.io/graph-convolutional-networks
[9] Yujia Li et al, *LaTeX: Graph Matching Networks for Learning the Similarity of Graph Structured Objects.* https://colab.research.google.com/github/deepmind/deepmind-research/blob/master/graph-matching-networks/graph-matching-networks.ipynbscrollTo=F50gn0G-yZ3s
[10] Peter W. Battaglia et al, *LaTeX: Relational inductive biases, deep learning, and graph networks.* https://arxiv.org/abs/1806.01261
[11] Henry Jackson-Flux et al, *LaTeX: Graph Neural Networks in TF2.* https://github.com/microsoft/tf2-gnn/
[12] Kai Sheng Tai et al, *LaTeX: Improved Semantic Representations From Tree-Structured Long Short-Term Memory Networks* . https://arxiv.org/abs/1503.00075
[13] Wang et al, *LaTeX: Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs.* https://arxiv.org/abs/1909.01315
[14] Fey et al, *LaTeX: Fast Graph Representation Learning with PyTorch Geometric.* https://github.com/rusty1s/pytorch-geometric
[15] Josh Vera et al, *LaTeX: Python Tree Sitter module.* https://github.com/tree-sitter/py-tree-sitter/
[16] Chris Thunes et al, *LaTeX: Javalang Library for working with Java source code in Python.* https://github.com/c2nes/javalang/
[17] Kovalenko et al, *LaTeX: PathMiner: a library for mining of path-based representations of code.* https://github.com/JetBrains-Research/astminer/
[18] Kovalenko et al, *LaTeX: PathMiner: a library for mining of path-based representations of code.* https://github.com/JetBrains-Research/astminer/
[19] Zhang et al, *LaTeX: A novel neural source code representation based on abstract syntax tree.* https://github.com/zhangj111/astnn
[20] Mark Cheung et al, *LaTeX: Pooling in Graph Convolutional Neural Networks.* https://arxiv.org/pdf/2004.03519v1.pdf
[21] Yusuke Shido et al, *LaTeX: Automatic Source Code Summarization with Extended Tree-LSTM.* https://arxiv.org/pdf/1906.08094.pdf
[22] Arseny Zorin et al, *LaTeX: Recurrent Neural Network for Code Clone Detection.* https://seim-conf.org/media/materials/2018/proceedings/SEIM-2018-Short-Papers.pdfpage=48
[23] Jain Zhang et al, *LaTeX: A Novel Neural Source Code Representation basedon Abstract Syntax Tree.* http://xuwang.tech/paper/astnn-icse2019.pdf

5 APPENDIX:

Here we show an example of real world programs and their comparison. The 3 following figures show two real world programs and their overlapping AST.

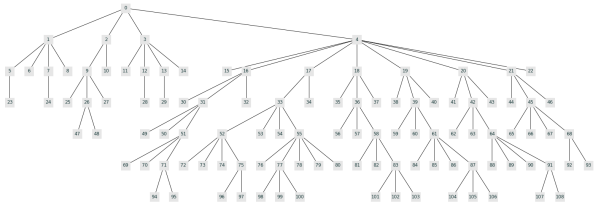

Figure 13: Program 1

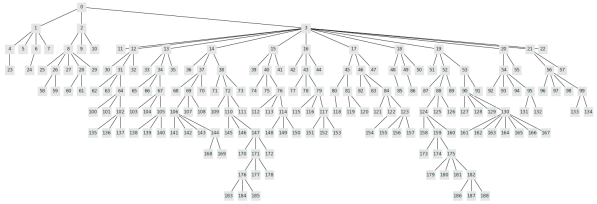

Figure 14: Program 2

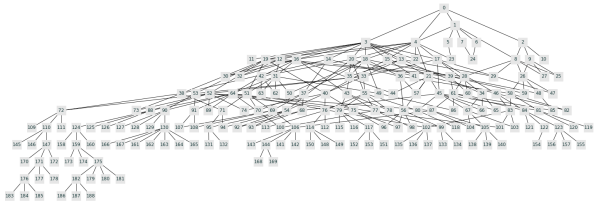

Figure 15: Overlapping AST of the two programs