Context survey

200007779

Supervisor: Rosa Filgueira

March 27, 2023

1 Introduction

Code similarity is an important topic in Computer Science. It can be used to detect plagiarism, to find similar code fragments, to find bug fixes, to detect malware, etc.

In this survey, we will discuss different methods of comparing cross-language or single-language code similarity, based on various features of the source code, such as their AST representations, call graphs, control flow graphs, code embeddings, as well as training models based on different architectures like RNN, LSTM and transformers.

2 Background

2.1 Type of clone

As proposed in [8], we often divide types of clones into four categories:

- Type I: Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.
- Type II: Structurally/syntactically identical fragments except for variations in iden- tifiers, literals, types, layout and comments.
- Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.
- Type IV: Two or more code fragments that perform the same computation but implemented through different syntactic variants.

In this project, our aim is to detect Type IV clones. Therefore the training data will contain code pairs that are semantically similar but syntactically different.

2.2 Abstract Syntax Tree

Abstract Syntax Tree or AST in short is a tree representation of the abstract syntactic structure of text (often source code) written in a formal language. Each node of the tree denotes a construct occurring in the text[12]. It contains semantics and syntactic information of the program, therefore is commonly used to analyze the similarity between source codes.

2.3 Control Flow Graph

Control flow graph or CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution[14]. It captures more comprehensive semantic information such as branch, loop and calling relationshops(represented by call graph) in the code, but much coarser syntactic information compared to AST[3]. It can be combined with AST to better represent the source code.

2.4 Embedding

In the context of machine learning, embedding is the process of mapping high-dimensional data to low-dimensional vectors, such that the low-dimensional vectors are close to each other if they are similar in some way. In natural language processing, embedding techniques is commonly used to represent words in a vector space, where the semantic similarity between words can be measured by the distance between their vectors. Programming language are similar to natural language in many ways, therefore embedding techniques could also be adopted to measure code similarity.

2.5 Recurrent Neural Network

Recurrent Neural Network or RNN in short is a type of neural network where connections between nodes can create a cycle, allowing output from some nodes to affect subsequent input to the same nodes[15]. RNN is good at processing sequential data or time series data and is commonly used in natural language processing tasks such as machine translation, speech recognition, etc.

There are two main disadvantages of RNN:

- It suffers from the vanishing/exploding gradient problem due to the nature of backpropagation. Although there are some solutions to this problem, such as gradient clipping and LSTM model, it is still not good at processing long sequences of data.
- As RNN relies on previous output to compute the next output when training, it cannot take advantage of parallel computing, which makes the traiing process slow.

2.6 Transformer

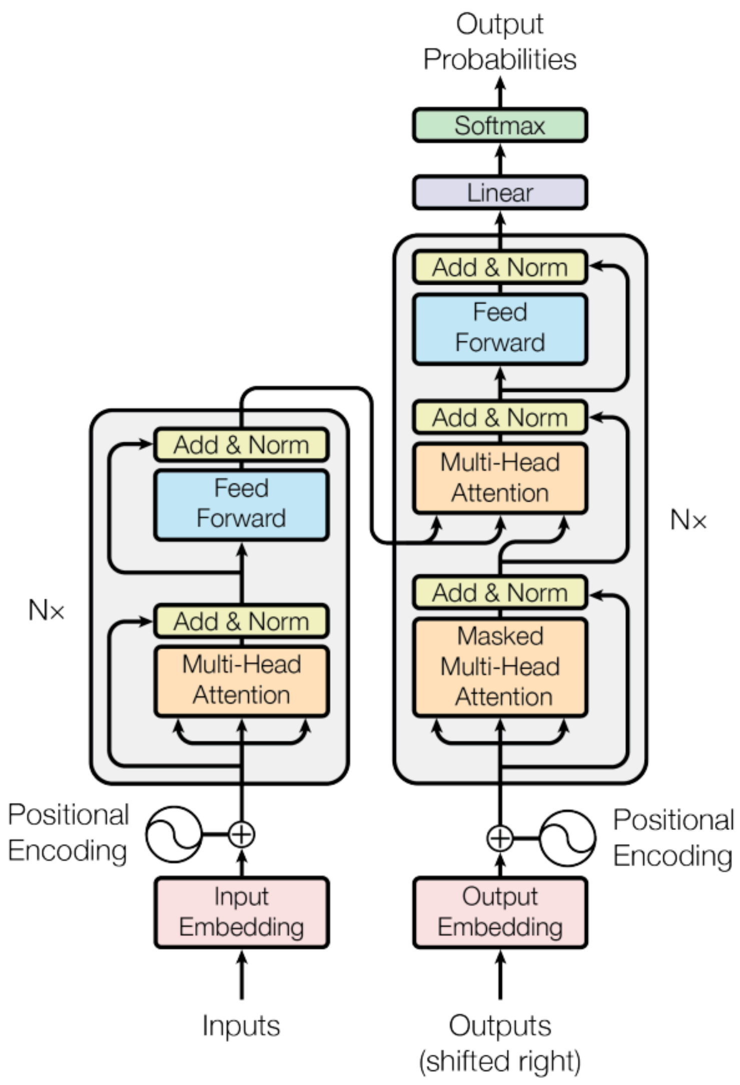


Figure 1: Transformer Architecture[9]

A transformer is a deep learning model that adopts the mechanism of self-attention, differentially weighting the significance of each part of the input data[16]. It is also designed to process sequential data but does not rely on recurrence, therefore it is able to process input data in parallel, which takes advantage of modern GPU's parallel computing power and makes the training process more efficient.

When training RNNs, input sequences are fed to the model in order, as opposed to transformers where input sequences are fed in parallel. In programming languages, the order of each statements determines the functionality of the code. Therefore, maintaining the order of original statements is crucial for our goal. The way the original transformer uses to preserve order information is by using positional encoding. After mapping inputs to embeddings, each embedding is added with a vector representing their positional information, so that the resultant embeddings are order-aware.

Aside from the positional encoding mechanism which allows the transformer to train more efficiently in parallel without losing input order information, another very important mechanism called self-attention enables the transformer to understand the importance of each part of the input data, as well as adding contextual information to those inputs. It also significantly reduces the computational complexity per layer and increases the amount of computation that can be parallelized as described in [9], making it more efficient to train.

Before transformers were introduced, most state-of-the-art NLP systems relied on gated RNNs[16]. Nowadays transformers are increasingly the model of choice for NLP problems, pre-trained transformers like BERT and GPT were trained with really large datasets[16]. They can be fine-tuned for specific tasks and achieve state-of-the-art results.

3 Methods

3.1 Training with AST

In many proposed methods[7][18][19], RNN such as LSTM are commonly used as encoders. The encoder is trained to convert AST to vectors(embeddings). To predict similarity a classification model is added to learn the similariy from embeddings.

In this paper[7] a tree-based skip-gram model is trained to encode AST to vectors. Skip-gram model is commonly used in natural language processing to turn words into vectors, but they can also be used in programming languages. An LSTM-based neural network is then trained to encode input data further and predict the similarity.

In another paper [18] the authors used a Tree-LSTM network to learn the semantic representation of a function from its AST. The Tree-LSTM network encodes two ASTs to two vectors, the outputs are then passed to a Siamese Network to calculated the similarity.

The above methods typically use labeled data to train the classifier, while the encoders are trained in an unsupervised fashion. There are also approaches based on unsupervised clustering method like [17]. In this approach key information from the function AST is extracted and encapsulated into a custom object. Each program is represented as a list of these objects, and the similarity between two programs is calculated from the similarity between their list representations. Finally, the similarity matrix is clustered and analyzed using unsupervised methods.

3.2 Training with Call Graph and Control Flow Graph

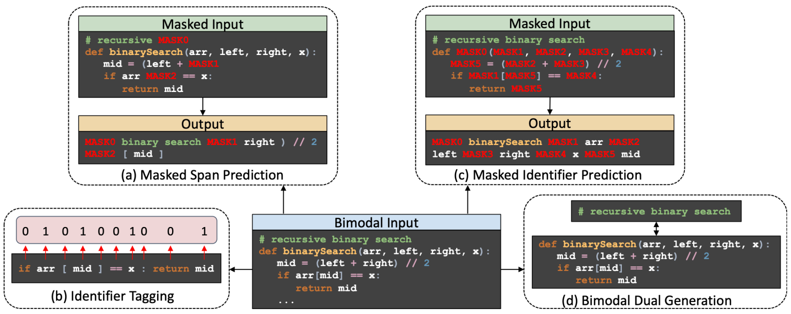
Approaches like [3] utilized the fact that functions with the same functionality often has similar function call graph and control flow graph structure. However, due to the abstract nature of the control flow graph, nodes difference such as use of operators in the control flow graph may not be identified clearly. Therefore, in this paper CFG and AST are both used as complement of each other and combined together to better represent the functionality of the code. Similar to AST approach, the embeddings techniques are used to encode the AST and CFG to extract syntactic and semantic features. Then they are combined to train a Deep Neural Network classifier to predict the similarity of the code snippets.

3.3 Transformer approaches

BERT Bidirectional Encoder Representations from Transformers or BERT in short is a transformer-based machine learning technique for natural language processing (NLP) pre-training developed by Google[13]. It has two pre-training objectives: masked language model and next sentence prediction. The masked language model randomly masks some words in the input sequence and the model is trained to predict the masked words. The next sentence prediction task is to predict if the second sentence is the continuation of the first sentence. Both tasks can be trained in unsupervised fashion which enables BERT to be pre-trained on large datasets like BooksCorpus and English Wikipedia[2]. The pre-trained BERT model is able to have certain level of understanding in the language and can be fine-tuned for specific downstream tasks.

CodeBERT CodeBert[4] is a pre-trained model for programming language, which is a multi-programming-lingual model pre-trained on NL-PL pairs in 6 programming languages (Python, Java, JavaScript, PHP, Ruby, Go). It uses the same architecture as RoBERTa[6](A Robustly Optimized BERT Pretraining Approach) which is one of the variations of BERT. It is the first large NL-PL pre-trained model for multiple programming languages. In the pre-training phase, it trains with both bimodal data which refers to natural language-code pairs, and unimodal data which refers to natural language or code only. The two objectives it has are Masked Language Modeling(as described in the last paragraph) and Replaced Token Detection. In replace token detection objective some input tokens are replaced with plausible alternatives generated from a generator, the goal is to train a model that determines whether each token in the corrupted input was replaced by a generator sample or not[1]. After pre-training and fine-tuning CodeBERT is able to have state-of-the-art performance on both natural language code search and code documentation generation.

GraphCodeBERT Similar to CodeBERT, GraphCodeBERT is the first pre-trained model that leverages the code structure to learn code representation and improve code understanding[5]. It follows BERT model structure and utilize source code with paired comments to pre-train the model. The two training objectives it has are Masked Language Modeling and Edge Prediction. In edge prediction objective, the model learns representation from data flow by training to predict masked variables' edges in data flows. When fine-tuned and evaluated on several downstream tasks, GraphCodeBERT shows slightly better performance than CodeBERT on natural language code search, and is able to achieve top results in clone detection tasks. In fine-tuning stage, the task is a binary classification where each input consists of two source codes and their corresponding data flows. The output is a probability of whether the two inputs are clones or not.



```
# recursive MASK0
def binarySearch(arr, left, right, x):
    mid = (left + MASK1
    if arr MASK2 == x:
        return mid
```

```
MASK0 binary search MASK1 right ) // 2
MASK2 [ mid ]
```

```
# recursive binary search
def MASK0(MASK1, MASK2, MASK3, MASK4):
    MASK5 = (MASK2 + MASK3) // 2
    if MASK1[MASK5] == MASK4:
        return MASK5
```

```
MASK0 binarySearch MASK1 arr MASK2
left MASK3 right MASK4 x MASK5 mid
```

```
if arr [ mid ] == x : return mid
```

```
# recursive binary search
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
    ...
```

```
# recursive binary search
```

```
def binarySearch(arr, left, right, x):
    mid = (left + right) // 2
    if arr[mid] == x:
        return mid
```

Figure 2: CodeT5 learning objectives[11]

CodeT5 Different from models above and many other models, CodeT5[11] does not rely on either an encoder-only model similar to BERT or a decoder-only model like GPT. When applied for code summarization tasks, encoder only models like CodeBERT needs an additional decoder which cannot benefit from the pre-training[11]. CodeT5 is a a pre-trained encoder-decoder model building on T5 architecture. Compared to GraphCodeBERT and some other models which leverages the structural aspect of the programming language, CodeT5 focus on leveraging identifiers whose names are assigned by developers as they contains rich semantic information of the code. Its objectives include **Identifier-aware Denoising Pre-training** which requires the decoder to recover the original source code from a randomly masked version, **Identifier Tagging** which requires the model to identify identifiers in the input, **Masked Identifier Prediction** which masks identifiers specifically and ask the model to recover them from the context, and **Bimodal Dual Generation** which aims to improve the alignment between the NL and PL counterparts. As a result, when evaluated on code clone detection task with the Java data provided in [10], the F1 score of CodeT5 is 97.2 which is slightly better than GraphCodeBERT's 97.1 and CodeBERT's 96.5. Moreover, it also performs well in other downstream tasks such as code summarization, code generation, etc.

4 Conclusion

In this survey, we have introduced various state-of-the-art approaches in the field of code clone detection. As the list of code understanding models growing, it is becoming more and more clear that transformer architecture is gaining its popularity in the field due to its training efficiency and great performance. Besides, transformer architecture can also be seen in the recent popular application `ChatGPT` driven by `GPT-3` model.

Therefore, in this project I will use pre-trained transformers as they perform better on larger dataset and the complexity of fine-tuning pre-trained models is much lower than training one from scratch.

References

[1] Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. Electra: Pre-training text encoders as discriminators rather than generators, 2020.

[2] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2018.

[3] Chunrong Fang, Zixi Liu, Yangyang Shi, Jeff Huang, and Qingkai Shi. Functional code clone detection with syntax and semantics fusion learning. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2020, page 516–527, New York, NY, USA, 2020. Association for Computing Machinery.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[5] 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. Graphcodebert: Pre-training code representations with data flow, 2020.

[6] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach, 2019.

[7] Daniel Perez and Shigeru Chiba. Cross-language clone detection by learning over abstract syntax trees. In *2019 IEEE/ACM 16th International Conference on Mining Software Repositories (MSR)*, pages 518–528, 2019.

[8] Chanchal Roy and James Cordy. A survey on software clone detection research. *School of Computing TR 2007-541*, 01 2007.

[9] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[10] Wenhan Wang, Ge Li, Bo Ma, Xin Xia, and Zhi Jin. 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)*, pages 261–271, 2020.

[11] Yue Wang, Weishi Wang, Shafiq Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, 2021.

[12] Wikipedia contributors. Abstract syntax tree — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Abstract_syntax_tree&oldid=1103626323, 2022. [Online; accessed 26-October-2022].

[13] Wikipedia contributors. Bert (language model) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=BERT_(language_model)&oldid=1107671243, 2022. [Online; accessed 31-October-2022].

[14] Wikipedia contributors. Control-flow graph — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Control-flow_graph&oldid=1115609094, 2022. [Online; accessed 25-October-2022].

[15] Wikipedia contributors. Recurrent neural network — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Recurrent_neural_network&oldid=1117752117, 2022. [Online; accessed 28-October-2022].

[16] Wikipedia contributors. Transformer (machine learning model) — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Transformer_(machine_learning_model)&oldid=1118251522, 2022. [Online; accessed 28-October-2022].

[17] Yifan Xie, Wenan Zhou, Huamiao Hu, Zhicheng Lu, and Mengyuan Wu. Code similarity detection technique based on ast unsupervised clustering method. In *2020 IEEE 6th International Conference on Computer and Communications (ICCC)*, pages 1590–1595, 2020.

[18] Shouguo Yang, Long Cheng, Yicheng Zeng, Zhe Lang, Hongsong Zhu, and Zhiqiang Shi. Asteria: Deep learning-based AST-encoding for cross-platform binary code similarity detection. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, jun 2021.

[19] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. A novel neural source code representation based on abstract syntax tree. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*, pages 783–794, 2019.