Learning of Python Source Code From Structure and Context

Adam Stecklov
261113375
adamkutak@gmail.com

Ling Fei Zhang
260985358
lzhang133@gmail.com

Noah El Rimawi-Fine
260914339
noah.elrimawi-fine@mail.mcgill.ca

GitHub: https://github.com/Ling01234/550Final-project

Abstract

Code clone Detection aims to detect code fragments that score highly on some metric of similarity. Detecting code clones is crucial for the maintenance and evolution of software. Existing approaches typically focus on either the structure or the identifiers in the code, but not both simultaneously. In this paper, we introduce a transformer model designed to encapsulate the semantics of Python functions and to predict function names based on code snippets. The core idea revolves around representing a code snippet as a fixed length vector embedding. This is achieved through a process involving the decomposition of code into an abstract syntax tree, extraction of path contexts, and feeding this set of path contexts to our encoder transformer. Then, the decoder processes this vector embedding and predicts a method name. We demonstrated the effectiveness of our approach by achieving 24.3% accuracy on our testing set, which consists of 2500 samples. We also demonstrate that the performance of the model increases proportionally to the dataset size.

1 Introduction

Vector representations of language, whether applied to individual words, sentences, or paragraphs, have greatly contributed to the works of natural language processing (NLP). For example, the distributed representation of words with "word2vec" showed that large language models (LLM) can learn the relationship between words with the concept of "king - man + queen = woman". However, we believe that the reach of LLMs should not be restricted to spoken language alone, but rather extended to all forms of language that convey actions, including coding languages. Specifically, our goal is to capture the semantic meaning of the Python language as actions.

Building upon the success of "word2vec", we want to distill Python functions into fixed length vectors, which encapsulates the semantic nuances of actions. In other words, we want to obtain fixed vector embeddings which represent the unique semantics of Python. For instance, if we have the two functions in 1, we would expect their vector embeddings to be nearly identical:

Function 1	**Function 2**

`def f(x):`
`  x = x + 10`
`  x = x**2`
`  return x`
	`def g(y):`
	`  y = (y + 10)**2`
	`  return y`

Table 1: Two functions that perform the same *action*.

Note that although the functions have generic names in this example, our dataset has function specific names. We use those names as labels and our model attempts to predict the label values. We also replace the function and variable names with generic ones before inputting the function. The hypothesis being that the model will attend to the actual actions in the code, instead of gleaning any meaning from the variable/function names.

2 Background

In this section, we provide some important definitions to help us understand how we obtain our vector embeddings (Alon et al., 2019).

Definition 1 (*Abstract Syntax Tree*) An AST for a function is a tuple $< N, T, \Sigma, s, \delta, \phi >$, where N is the set of non-terminal nodes, T is the set of terminal nodes, Σ is the set of values, s is the starting node, $\delta : N \rightarrow (N \cup T)^*$ is a function that maps non terminal nodes to its children nodes and $\phi : T \rightarrow \Sigma$ is a function that maps a terminal node to a value.

Definition 2 (*AST path*) An AST path p is a specific path in the AST from a terminal node n_1 to another terminal node n_{k+1}, where k is the length of the path. Specifically, and AST path of length k is a sequence $\{n_1 d_1 ... n_k d_k n_{k+1}\}$, where $n_i \in N, i \in [2, k]$ (i.e. non terminal) and $d_i \in \{\uparrow, \downarrow\}, i \in [1, k]$ are movements up or down the tree.

Definition 3 (*Path Context*) A path context is a tuple $< s, p, t >$ such that p is an AST path, and $s = \phi(n_1), t = \phi(n_{k+1})$. In other words, a path context represents a specific path in the AST by moving through the tree where we go from a value to another value.

3 Related Work

The evolution of code clone detection has greatly risen over the years. Early research on these learned language models, used solely ASTs (Baxter et al., 1998), text representations, tokens, and syntax indicators (Zhang et al., 2023). In the past couple of years, there have been many deep learning approaches taken to detect clones. The most common approaches have been to use ASTs as input for convolutional neural networks (CNNs), graph neural networks (GNNs), recursive recurrent neural networks (RtNNs) to model sequence of terms, compositional learning algorithms like recursive neural networks (RvNN) that are capable of modeling arbitrary structures, to predict the sentiment of natural language sentences (Goller and Küchler, 1996), or long short-term memory models (LSTMs). However, the existing methods based on the above networks have failed to capture contextual information with a full receptive field, as they miss certain long-range dependencies between code tokens (Zhang et al., 2023).

To make up for these failures, transformer based methods have been proposed. Transformers are deep learning models that rely on self-attention mechanisms to weigh the significance of different parts of the input data. Unlike traditional models that process data sequentially, transformers can handle various parts of the data simultaneously, making them exceptionally efficient for tasks involving large datasets or complex dependencies. These models have demonstrated their superior ability to model long-term dependencies (Devlin et al.,2017). Zhang et al., demonstrate a feasible way to apply the transformer for efficient code clone detection, by using an efficient position embedding strategy, combined with a code token learner built with Graph Convolutional Networks and an attention mechanism. They make an additional similarity token to capture long-range dependencies.

Other models, such as Code2Seq and Code2vec have been used for code cloning and have achieved SOTA performance. Code2seq is a neural model designed for code identification using distributed vectors. Specifically, the model aims to predict a method's name given a vector representation of its body. To achieve this, the authors employ a tree-based abstract syntax tree (AST) to capture code structures. The model learns to map variable names to their corresponding context paths in the AST, and learns the atomic representation of each path simultaneously with learning how to aggregate a set of them. The model's training also involves a path-based attention mechanism, which enables it to focus on more relevant parts of the AST. Code2vec is a neural network for predicting program properties, centered around creating code embeddings, or vector representations of code. These embeddings link code snippets to labels, such as tags or names. The architecture merges multiple syntactic paths from a code snippet into a single vector, using a similar technique to word embeddings, as described by Mikolov et al. (Mikolov et al., 2013). The model processes a code snippet and an associated label, with the label representing a semantic property, like a tag or the name of a method or class. Given a code snippet (C) and its label (L), the model aims to infer the distribution of labels based on the syntactic paths in C, learning to predict P(L|C).

4 Dataset

For this project, our primary data source is Hugging Face's "the-stack-smol" dataset. This dataset is a small subset (approx. 0.1%) of "the-stack" dataset, where each programming language has 10 000 random samples from the original dataset. The dataset will be divided into training (70%) and testing (30%) sets. The Stack contains 100s of GB of Python code. Our model and data pipeline is fully compatible with the stack, but due to computational limits we use the small subset instead.

Each sample consists of a Python module. However, since our focus was only on understanding function semantics, we employed a few preprocessing techniques to tailor the dataset for our objective. Firstly, for each module, we extracted all functions within the file, treating each function as an individual row in our dataset. Then, we transformed each function into abstract syntax trees (AST) in order to replace variable and method names with generic ones (func1, var1, var2, etc…). The actual method name is stored and served as the true label for each sample. Finally, we reverted the processed AST back into code as the X input, and the Y input being the function name label.

Our code also extracts the comments from functions, however we ran into two issues with this and thus had to abandon the use of comments in our model (either as an input feature, or as a label). The first issue was most of the files lacked comments, and secondly those comments were not evenly distributed between the different functions in the files, or we could not find a programmatic way to decide which comments belonged to which functions.

Other preprocessing steps we took include converting all code to Python3, removing any examples that don't parse in a Python3 tree and removing non-ascii characters or other irregularities with regular expressions.

When formatting the data for inputting into the transformer model, we create a vocabulary where the tokens are the elements in the path contexts. We also create a vocabulary for the output labels, making it a classification task.

5 Method

One of the most challenging parts of this paper was making our dataset compatible with the open source astminer, as it has not been supported for a couple years. Initially designed by Along et al., the feature extraction script in our project has been adapted from JetBrain's open-source AST Miner. We configured the miner via a YAML file specifying our dataset's location, enabling it to sequentially process each file. We paired the dataset with ANother Tool for Language Recognition (ANTLR), a parser generator with an extensive grammar for the Python language used to parse and build trees, generating the Abstract Syntax Tree (AST).

The AST path consists of terminal nodes at both ends, representing token values. The middle layer comprises the path ID and the corresponding node type. Using this output, the AST miner traverses the tree to identify the path between terminal nodes. It outputs three .csv files, and a pathcontexts.c2s file. Using the pathcontexts file, we split the nodes into three segments: a start token, the path (a dictionary pairing of the path ID and the corresponding node type), and an end token.

Our neural network is composed of an encoder and a decoder. The encoder processes a set of path contexts, and outputs a fixed vector embedding that captures the function's semantics. The decoder takes this vector embedding as input and predicts a method name. To measure our model's performance, we employ the cross-entropy loss function. This error is propagated from the decoder back to the encoder, allowing us to train both the encoder and the decoder simultaneously. We can throw away the decoder at the end, and are left with the encoder which embeds the input as a vector in 128 dimensional space. Below are the default model parameters, chosen in the absence of a validation set for further fine tuning. More details on the computational limits are discussed in section 7.

input dim	Size of vocabulary
embed dim	128
num heads	4
num layers	2
output dim	Size of label vocabulary
epochs	20

Table 2: Transformer Model Params

6 Results

In table 3, we present code clone results on the small python dataset. As the dataset grows in size, our model shows improvements. This trend suggests that our model is effectively learning from the data it processes and it indicates that our model benefits from more extensive and diverse examples, enhancing its ability to recognize patterns and similarities between code fragments. This adaptability and learning capacity are crucial for the model's effectiveness in identifying code clones, particularly in complex and varied codebases. The positive correlation between dataset size and model performance is a strong indication of the model's scalability and robustness. As it encounters more data, it fine-tunes its understanding and detection capabilities, making it increasingly proficient in identifying code clones with higher accuracy. Overall, the results from the small Python dataset serve as a promising foundation, and the model's improved performance with larger datasets demonstrates its potential for further growth. However, the low accuracy emphasizes the fact that our model still performs quite poorly, and while the leading cause is likely a very small dataset due to computational limitations. Other factors that come into play are things such as the way code snippets are represented and processed in the model. If the feature representation doesn't capture the essential aspects of the code's structure or semantics effectively, the model may struggle to differentiate between clones and non-clones accurately. Our feature representation does delve into both the context and the structure of the code, but we focused on a single language and our examples may have been too similar in scope and functionality to properly differentiate our examples.

Data Size	Accuracy	Precision	F1-Score
500	0.200	0.154	0.171
2 700	0.243	0.185	0.200

Table 3: Model Performance

Figure 1 shows the transformer encoder-decoder model being trained over 20 epochs as its loss decreases. The model learns quite effectively. By the 15th epoch, the loss had essentially flatlined at approximately 0.015.

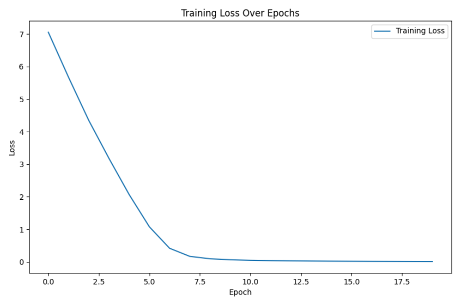

Figure 1: Training Loss over Epochs

7 Discussion and Conclusion

We presented a novel code-to-sequence model which considers the joint contribution of the source-code (the context) and the parsed abstract syntax tree (the structure). By leveraging the path contexts from the AST into a more verbose format by creating sequences and passing those into our encoder transformer to extract feature representations. However, the performance of the model is low, and probably would not be considered useful.

However, our methodology could be improved in several regards. Firstly, the main issue with this model is the lack of data and computing power. If we had access to cloud compute time, we could probably use the entire python section of the stack (100s of GB of python files). Leveraging the full The Stack dataset would likely greatly improve the models performance. Additionally, the presence of a validation set for parameter fine-tuning is another area for potential improvement. Unfortunately, we did not have enough computing power to utilize a validation set. Secondly, we think we could leverage the tree structure of an AST tree to make use of a graph neural network (GNN). Instead of generating paths between nodes in the tree to give to the transformer model in a string shape, the shape of the graph would be accounted for in a GNN. Python code leverages structure in its design (such as nesting for if statements or for loops). This project could also be expanded upon in a few ways. The transformer model we used could be enlarged or we could add additional layers to it. Finally, we could do a multi-language model that works across more than just Python. It has been shown that multilingual models outperform mono-lingual variants on all programming languages (Zügner et al., 2021).

Another area of future work that would assist in elucidating our model's efficacy would be to create a custom generated dataset with two similar functions and one different function, run them through our transformer model, and then get the cosine similarity between the three functions, such that we can properly test the accuracy of our model.

A major source of failure is the Astminer library we relied on to generate the path contexts. It had errors and dependency problems we had to fix manually, then recompile it as a jar. This ate up significant development time and ended up slowing our project down a lot. We also tried generating the path contexts manually, but it is a very involved process.

8 Statement of Contributions

Noah worked on astminer; Adam worked on data preprocessing; Ling worked on the model. All three team members equally contributed to the redaction of the report.

References

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. Code2vec: Learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL):40:1–40:29.

I. D. Baxter, A. Yahin, L. Moura, M. Sant'Anna, and L. Bier. 1998. Clone detection using abstract syntax trees. In *Proceedings. International Conference on Software Maintenance (Cat. No. 98CB36272)*, pages 368–377, Bethesda, MD, USA.

Christoph Goller and Andreas Küchler. 1996. Learning task-dependent distributed representations by backpropagation through structure. In *ICNN'96*.

Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*.

Aiping Zhang, Liming Fang, Chunpeng Ge, Piji Li, and Zhe Liu. 2023. Efficient transformer with code token learner for code clone detection. *Journal of Systems and Software*, 197:111557.

Daniel Zügner, Tobias Kirschstein, Michele Catasta, Jure Leskovec, and Stephan Günnemann. 2021. Language-agnostic representation learning of source code from structure and context. In *International Conference on Learning Representations (ICLR)*.