Learning of Python Source Code From Structure and Context

Adam Stecklov
ID
adamkutak@gmail.com

Ling Fei Zhang
260985358
lzhang133@gmail.com

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

Abstract

In this paper, we introduce a neural 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 x% accuracy on our testing set, which consists of 2500 samples.

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 Table 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, $\delta : N \to (N \cup T)^*$ is a function that maps non terminal nodes to its children nodes and $\phi : T \to \Sigma$ 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), or long short-term memory models (LSTMs). Harun et al, has shown that CNNs have been successful, where they used a method pair to be compared and converted to a sequence of tokens, similar to an image data in order to obtain sensible input for CNN. 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. They 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.

4 Dataset

Dataset: For this project, our primary data source is Hugging Face's "the-stack-smol" dataset. This dataset is a small subset (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 (60%), validation (15%), and testing (25%) 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.

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. Additionally, these values are one-hot encoded to serve as input for our transformer model.

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 model parameters that yielded the highest accuracy based on the validation set:

input dim	10 000
embed dim	128
num heads	4
num layers	2
output dim	100
epochs	10

Table 2: Transformer Model Params

6 Results

results

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 using by creating sequences and passing those into our encoder transformer to extract feature representations, we successfully predicted similarity between two code fragments. However, our work could be improved in several regards. Firstly, our model is only trained on the Python language, and is thus, not language agnostic. It has been shown that multilingual models do outperform mono-lingual variants on all programming languages (Zügner et al., 2021). Secondly, our training dataset is limited in size since we do not have access to high-end computing power and hardware. We believe that leveraging a larger dataset could potentially improve our model accuracy. Lastly, we believe that it is worthwhile to explore the incorporation of additional layers into our neural network. Specifically, the inclusion of convolution layers, known for raw data processing, could potentially contribute to the model's performance.

8 Statement of Contributions

Work was equally distributed among the three team members.

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.

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).*