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, for the two functions in Table 1, we would expect their vector embeddings to be similar.

It's worth noting that in our examples, we've given different functions and variable names. This is because, ideally, the neural model should be able to recognize actions, which is independent of variable or function names.

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*.

2 Background

In this section, we provide some important definitions to help us understand how we obtain our vector embeddings.

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 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.

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 generalize variable and method names. The actual method name is stored and served as the true label for each sample. Finally, we reverted the AST back into code, resulting in our cleaned code.

During preprocessing, our initial intent was to preserve comments in the functions. However, we observed that around 90% of the functions lacked comments. Thus, we decided to exclude all comments from our dataset to avoid potential outlier effects. Finally, The dataset will be divided into training (60%), validation (15%), and testing (25%) sets.

4 Related Work

The main contributions of this paper are (Alon et al., 2019):

5 Method

Row 1, Column 1	Row 1, Column 2
Row 2, Column 1	Row 2, Column 2
Row 3, Column 1	Row 3, Column 2
Row 4, Column 1	Row 4, Column 2
Row 5, Column 1	Row 5, Column 2
Row 6, Column 1	Row 6, Column 2

Table 2: Transformer Model Params

6 Results

results

7 Discussion and Conclusion

discussion

8 Statement of Contributions

contrib

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.