Exploring code2vec and ASTminer for Python Code Embeddings

Conference Paper · June 2023

DOI: 10.1109/SEAI59139.2023.10217505

4 authors, including:

Long H. Ngo
Université Sorbonne Paris Nord
11 PUBLICATIONS **64** CITATIONS

Exploring code2vec and ASTminer for Python Code Embeddings

Long H. Ngo
SMILE France
Asnières, France
long.ngo@smile.fr

Veeraraghavan Sekar
SMILE France
Asnières, France
svraghavan7@gmail.com

Etienne Leclercq
SMILE France
Asnières, France
etienne.leclercq@smile.fr

Jonathan Rivalan
SMILE France
Asnières, France
jonathan.rivalan@smile.fr

***Abstract*—Automated understanding of code meaning and use has become an intrinsic part of software development recently. Neural models are being used in various natural language processing tasks, as they can represent natural language using vectors that carry semantic meanings. Although code is not natural language, we believe neural models to be capable of learning semantics and syntactic properties available in code snippets. To achieve such goal, we represent a code snippet using its abstract syntax tree (AST) syntactic paths to capture regularities that reflect common code patterns. This representation lowers significantly learning effort while being scalable to multiple problems and large code bases. In our work, we adopt ASTminer with code2vec, to represent code snippets as continuously distributed code vectors called "code embeddings", used to predict the semantic properties of the snippets. This approach decomposes code into a collection of AST paths and learns each path's atomic representation while learning how to aggregate them. Code2vec is then paired with other neural models, which represent query, to create a hybrid model for the task of code search. While code2vec was originally developed for Java only, we present in this article our efforts to extend the method to Python language.**

Keywords—machine learning, neural network, distributed representations, code search

I. Introduction

Specific problems that require a brief semantic descriptor for a code snippet, such as a method name assignment, illustrate the need for a compact semantic representation. The challenge falls into encoding code snippets to capture semantically relevant information that is transferable across multiple programs and enables property prediction, such as labeling a snippet. This challenge involves two key aspects: first, representing a code snippet in a conducive manner that facilitates learning across different programs; second, discerning which parts of the representation are pertinent to predict the intended property and in which order to prioritize.

Goal. The objective of this study is to apply the path-based representation method to address two important tasks in software development: semantic labeling of code snippets and code search. Note, the path-based approach represents a snippet of code as a group of paths from its syntax tree that captures the structure of the code carrying its semantics and other information [1].

The first task aims to automate the process of predicting the semantic label of a given code snippet. This task poses a significant challenge due to the need to learn the complex correspondence between the entire content of a method and a semantic label. Specifically, the task involves the aggregation of potentially hundreds of expressions and statements from the method body into a single descriptive label. Effective solution to this task requires the development of advanced techniques for code representation and classification [2].

The second task is to enable an efficient and effective search for code snippets that match the requirements of input queries. This task is critical for software developers who often need to locate and integrate existing code into their projects. Effective code search requires the ability to match the query against a large codebase and to retrieve relevant code snippets that satisfy the query requirements. The successful solution of both tasks will significantly improve the productivity and efficiency of software development.

Approach. In this study, we utilized the code2vec neural network architecture [2], which has been designed to learn code embeddings that are low-dimensional vector representations of source code. Within these vectors, an entity's "meaning" is dispersed among several vector components, resulting in the mapping of semantically related entities to closed vectors. These embeddings provide a means to model the relationship between code snippets and associated labels in a natural and efficient manner. By leveraging the inherent structure of source code, code2vec is capable of aggregating multiple syntactic paths into a single vector, resulting in a more comprehensive and accurate representation of the code snippet.

II. Related Works

In natural language processing (NLP), the conventional practice, which are based on deep neural networks [3], [4], is to consider texts as a linear sequence of tokens. This approach is also commonly applied to source code representation in many existing methods [5]-[9]. However, recent research has suggested that structured representations that exploit the syntax of programming languages can significantly enhance their performance [10]-[12].

In recent years, progress has been made in the problem of predicting program properties through learning from massive codebase [5], [7], [8], [11], [13]. The ability to predict program semantic properties without its execution, and with minimal or

no semantic analysis, holds significant potential for various applications, including but not limited to predicting program entity names [10], [12], [14], code completion [15], [16], code summarization [5], code generation [17]-[19], among others [20], [21].

Recently, the authors of [2] presented a cutting-edge approach to learning distributed representations of code using a neural network, namely, code2vec. This paper addressed one of the most challenging problems in software engineering, namely, code representation. Unlike the conventional approaches, which focus solely on static code features, the proposed method utilizes a large corpus of code snippets to learn distributed representations that capture the temporal dynamics of code changes, resulting in more accurate and robust representations of code.

The code2vec model is a neural network-based approach for learning distributed representations of code. The model consists of two main components: an input encoder and a code vectorizer. The input encoder takes as input a code snippet and converts it into a sequence of tokens, which is then fed into a neural network. The network is trained to predict the next token in the sequence, similar to a language model. The code vectorizer takes the output of the input encoder and uses it to generate a continuous-valued vector representation of the code. The vector representation captures both the syntactic and semantic information of the code, allowing it to be used for various software engineering tasks. The code vectorizer uses attention mechanisms to weigh the importance of different parts of the code snippet, capturing the essential aspects of the code that are relevant to the task at hand. The resulting vector representation can be used for tasks such as code completion, bug detection, and program synthesis. Overall, the code2vec model represents a significant departure from traditional approaches to code representation and has shown considerable success in various software engineering tasks. The architecture of the code2vec network is illustrated in Fig. 1.

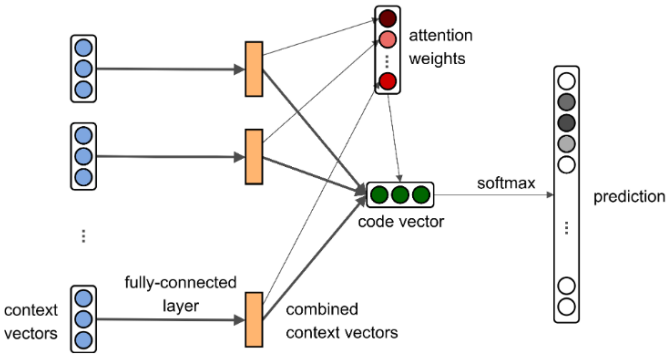

Fig. 1. Original code2vec architecture: A fully-connected layer learns to combine embeddings of each path-contexts with itself; attention weights are learned using the combined context vectors and used to compute a code vector. The code vector is used to predict the label [2].

III. Methodology

High-level view. The concept of a code snippet in programming is crucial as it allows developers to write concise and efficient pieces of code that can be reused in different contexts. To better understand how code snippets are constructed, it is essential to delve into the idea of a bag of contexts. Each context within a code snippet is represented by a vector known as the context vector. This vector is generated through a process of learning that captures two important aspects of the context: (i) its semantic meaning and (ii) the level of attention it should get. Through the aggregation of these context vectors, a singular code vector is derived, which can then be employed in subsequent tasks.

A. Semantic Labeling of Code Snippets

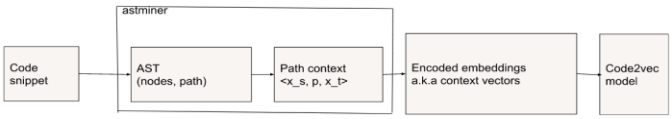

Fig. 2. code2vec architecture for Python programming language.

In order to extend code2vec to work with Python language, the ASTminer extractor [1] is employed to replace the JavaExtractor [2] used by the original code2vec model. ASTminer is an open-source library that enables the mining of code's path-based representations, as depicted in Fig. 2. By parsing the code and extracting the paths from its syntax tree, ASTminer provides a reusable toolkit that alleviates the burden of modeling source code for machine learning algorithms. The significance of an AST lies in its ability to depict the syntactical framework of a program, while selectively disregarding certain details, such as formatting, punctuation, or specific forms of syntactic constructs. Within the AST, each node corresponds to a distinct syntactic unit, including but not limited to variables, operations, and logical operators, while its respective children denote the lower-level units affiliated with the current node [1].

Using the path-based representation of code, the ASTminer extractor captures the coding style or structure and summarizes the coding logic. The conversion of code into embeddings occurs in two steps, where each code is first transformed into a vector, and these vectors are then combined, along with their corresponding weighted sum attention vectors, for further training of the vector model. The ASTminer architecture, shown in Fig. 3, is based on the code2vec embedding model.

Following the work in [22], the embeddings are constructed using the code and code path tokens. Then these embeddings are concatenated and converted to numerical representation by applying the activation function FC, as depicted in Fig. 3. The resultant vectors are further merged to create a batch where the weights of individual paths are added to identify the importance of path vectors. Finally, the code author or developer is predicted by applying analysis over the batch information. Similarly, the code title is predicted in our use case which follows a similar pipeline.

Analogous to [2], cross-entropy loss between the predicted distribution q and the "true" distribution p is used in the training phase. p assigns a value of 1 to the actual tag in the training sample and 0 otherwise. Hence, the cross-entropy loss for a

single sample is equivalent to the negative log-likelihood of the true label, which can be expressed as follows:

$$\mathcal{H}(p||q) = -\sum_{y \in Y} p(y) \cdot logq(y) = -logq(y_{true}) \quad (1)$$

where y_{true} is the actual tag that was seen in the sample. The loss is the negative logarithm of $q(y_{true})$, the probability that the model assigns to y_{true}. As $q(y_{true})$ tends to 1, the loss approaches zero. The further $q(y_{true})$ goes below 1, the greater the loss becomes. Thus, minimizing this loss is equivalent to maximizing the log-likelihood that the model assigns to the true labels y_{true}. In the process of network training, a gradient descent algorithm is employed to update the learned parameters. The prevailing method involves the standard approach of backpropagating the training error through each of the learned parameters. Specifically, this entails computing the gradient of the loss function with respect to each of the learned parameters and iteratively updating their values by a small step in the direction of minimizing the loss.

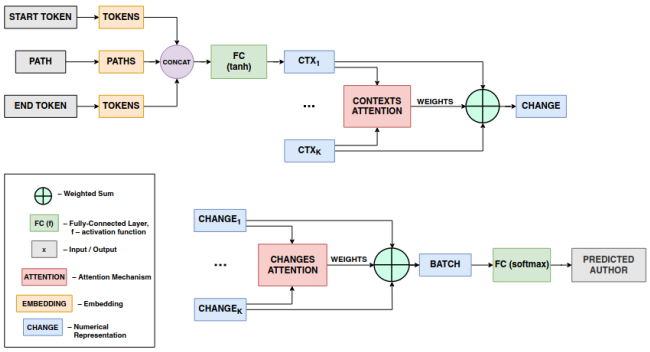

Fig. 3. Overview of authorship attribution pipeline, which generates authorship-based embeddings of method changes. Furthermore, it highlights the significance of individual method changes in the authorship attribution. The method nodes and the corresponding attention weights are subsequently utilized to create author representation [22].

Building on the study conducted in [2], the network can be effectively applied to predict categories for unseen code. This approach involves computing the code vector based on the weights and parameters that were acquired through the training phase. Subsequently, the prediction process entails identifying the closest target label. Such a predictive model has the potential to facilitate accurate and efficient categorization of software code snippets, thereby enhancing various tasks such as software maintenance, bug detection, and code optimization.

B. *Code search*

The present study draws on previous research in the works of [23]-[25] to construct a neural search system that uses joint embeddings of codes and queries. This system is built using one encoder per input language, be it natural or programming, and these encoders are trained to map their respective inputs to a single, joint vector space. The primary goal of the training is to ensure that the vector representations corresponding to code and its associated language are located in close proximity to each other's, as this allows for the implementation of a search algorithm that embeds queries and retrieves the relevant code snippets based on their proximity in the embedding space. Despite the existence of more sophisticated models that take into account multiple interactions between queries and code, the use of a single vector per query/snippet in this architecture facilitates efficient indexing and search [25]. A brief overview of the model's architecture we designed for the Python language of code search task is shown in Fig. 4.

This study uses our extended code2vec model for Python programming language to encode code snippets. The docstring accompanying each code snippet is tokenized, and the resulting tokens are encoded via a query encoder, such as the Neural Bag of Words (NboW) [26], Bidirectional RNN model [27], 1D Convolutional Neural Network [28], or Self-Attention [29]. Subsequently, the resulting token embeddings are combined using a pooling function to obtain a sequence embedding. In this context, we have implemented mean/max-pooling and an attention-based weighted sum mechanism. In the training process, a collection of N code-docstring pairs denoted by (c_i, d_i) is presented, alongside the instantiation of a code encoder E_c and a query encoder E_q. Our objective is to minimize the loss function:

$$-\frac{1}{N}\sum_{i} log\left(\frac{exp(E_c(c_i))^T E_q(d_i)}{\sum_j exp(E_c(c_j))^T E_q(d_i)}\right) \quad (2)$$

The above equation aims to maximize the inner product of the code and query encodings of the pair while minimizing the inner product between each c_i and the distractor snippets $c_j (i = j)$. During model training, Mean Reciprocal Rank (MRR) serves as the evaluation metric to assess the model's performance against the validation dataset. At the testing stage, Annoy[1] is employed to index all functions in the CodeSearchNet Corpus. Annoy is a high-speed, approximate nearest neighbor indexing and search technique offered by Spotify. The index encompasses all functions in the corpus, including those without an accompanying documentation comment. The construction of this index is pivotal in achieving favorable results [25].

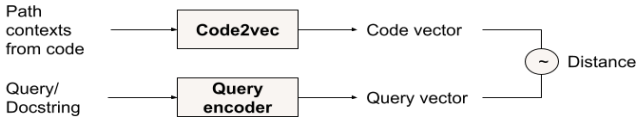

Fig. 4. Text to code architecture based on code2vec model.

C. *Result - Challenge*

As previously mentioned, the training of the network needs the incorporation of both code snippets and their corresponding queries or docstrings. Nevertheless, we have encountered a challenge pertaining to the management of output path contexts in ASTminer. Consequently, the difficulty of training the network arises when it is impossible to align the given docstring with extracted path contexts from the input code snippet. There is the possibility of conducting research to build a new tool for extracting information from Python language or adapting the

1 https://github.com/spotify/annoy

currently existing ASTminer to extract path-based representations of code, according to specific requirements.

IV. Conclusion and Future Works

This article discusses the use of the neural model for code representation and search, with a focus on the path-based representation of code snippets. We presented our efforts to extend the code2vec, a path-based neural model, to Python language for the task of code labeling and search. We utilized paths in the abstract syntax tree to represent code snippets and learned their atomic representation as well as aggregating a set of them. To achieve this, we used ASTminer with code2vec to represent code snippets as continuously distributed code vectors, also known as "code embeddings," which can be used to predict the semantic properties of the snippet. We also presented a hybrid model for the code search task by pairing code2vec, which represents code, with another neural model, which represents query/docstring. The model was trained using cross-entropy loss to predict the name of the code. Overall, our work demonstrates the promise of using neural models for code labeling and search in Python language or any other languages and highlights the importance of incorporating the syntactic and semantic properties of code snippets in these models.

However, the problem of mapping the code's path contexts and its corresponding query, as previously mentioned, needs overcoming in future works. One may study and develop a novel extractor for Python language or modify the existing ASTminer that mines the path-based representations of code as desired. Hence, we can control the output path contexts of this extractor. Recently, large language models (LLMs) show their advantages over conventional methods in NLP tasks. Therefore, it is worth studying this approach following three baseline systems, including the BERT-style bidirectional encoder representation, GPT-style decoder representation, and Encoder-Decoder representation from transformer architecture, with or without the AST representation for the code search task. Leveraging LLMs, like ChatGPT in an open source perspective, in the context of Python programming can be valuable for automatically providing textual hints and suggestions. By fine-tuning ChatGPT on the code search dataset that pairs code snippets with queries/docstrings, it can generate relevant hints based on a given code snippet or query. This approach has the potential to contribute to the open-source knowledge transfer from proprietary models to the wider developers community.

References

[1] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, “Pathminer: a library for mining of path-based representations of code,” in Proceedings of the 16th International Conference on Mining Software Repositories. IEEE Press, 2019, pp. 13–17.

[2] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “code2vec: Learning distributed representations of code,” Proceedings of the ACM on Programming Languages, vol. 3, no. POPL, pp.1–29, 2019.

[3] Shuo Hu, Yaqing Zuo, Lizhe Wang, and Peng Liu, "A Review about Building Hidden Layer Methods of Deep Learning," Vol. 7, No. 1, pp. 13-22, February, 2016. doi: 10.12720/jait.7.1.13-22

[4] Yasufumi Sakai, Yu Eto, and Yuta Teranishi, "Structured Pruning for Deep Neural Networks with Adaptive Pruning Rate Derivation Based on Connection Sensitivity and Loss Function," Journal of Advances in Information Technology, Vol. 13, No. 3, pp. 295-300, June 2022.

[5] M. Allamanis, H. Peng, and C. Sutton, “A convolutional attention network for extreme summarization of source code,” in International conference on machine learning. PMLR, 2016, pp. 2091–2100.

[6] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, “Toward deep learning software repositories,” in 2015 IEEE/ACM 12th Working Conference on Mining Software

[7] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Learning natural coding conventions,” in Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2014, pp. 281–293.

[8] M. Allamanis and C. Sutton, “Mining source code repositories at massive scale using language modeling,” in 2013 10th working conference on mining software repositories (MSR). IEEE, 2013, pp. 207–216.

[9] D. Movshovitz-Attias and W. Cohen, “Natural language models for predicting programming comments,” in Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), 2013, pp. 35–40.

[10] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, “A general path-based representation for predicting program properties,” ACM SIGPLAN Notices, vol. 53, no. 4, pp. 404–419, 2018.

[11] P. Bielik, V. Raychev, and M. Vechev, “Phog: probabilistic model for code,” in International conference on machine learning. PMLR, 2016, pp. 2933–2942.

[12] V. Raychev, M. Vechev, and A. Krause, “Predicting program properties from” big code”,” ACM SIGPLAN Notices, vol. 50, no. 1, pp. 111–124, 2015.

[13] V. Raychev, P. Bielik, and M. Vechev, “Probabilistic model for code with decision trees,” ACM SIGPLAN Notices, vol. 51, no. 10, pp. 731–747, 2016.

[14] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, “Suggesting accurate method and class names,” in Proceedings of the 2015 10th joint meeting on foundations of software engineering, 2015, pp. 38–49.

[15] V. Raychev, M. Vechev, and E. Yahav, “Code completion with statistical language models,” in Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation, 2014, pp. 419–428.

[16] A. Mishne, S. Shoham, and E. Yahav, “Typestate-based semantic code search over partial programs,” in Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 2012, pp. 997–1016.

[17] M. Amodio, S. Chaudhuri, and T. W. Reps, “Neural attribute machines for program generation,” arXiv preprint arXiv:1705.09231, 2017.

[18] Y. Lu, S. Chaudhuri, C. Jermaine, and D. Melski, “Data-driven program completion,” arXiv preprint arXiv:1705.09042, 2017.

[19] C. Maddison and D. Tarlow, “Structured generative models of natural source code,” in International Conference on Machine Learning. PMLR, 2014, pp. 649–657.

[20] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, “A survey of machine learning for big code and naturalness,” ACM Computing Surveys (CSUR), vol. 51, no. 4, pp. 1–37, 2018.

[21] M. Vechev, E. Yahav et al., “Programming with “big code”,” Foundations and Trends® in Programming Languages, vol. 3, no. 4, pp. 231–284, 2016.

[22] V. Kovalenko, E. Bogomolov, T. Bryksin, and A. Bacchelli, “Building implicit vector representations of individual coding style,” in Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops, 2020, pp. 117–124.

[23] X. Gu, H. Zhang, and S. Kim, “Deep code search,” in Proceedings of the 40th International Conference on Software Engineering, 2018, pp. 933–944.

[24] B. Mitra, N. Craswell et al., “An introduction to neural information retrieval,” Foundations and Trends® in Information Retrieval, vol. 13, no. 1, pp. 1–126, 2018.

[25] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, “CodeSearchNet challenge: Evaluating the state of semantic code search,” arXiv preprint arXiv:1909.09436, 2019.

[26] I. Sheikh, I. Illina, D. Fohr, and G. Linares, “Learning word importance with the neural bag-of-words model,” in Proceedings of the 1st Workshop on Representation Learning for NLP, 2016, pp. 222–229.

[27] K. Cho, B. Van Merriënboer, D. Bahdanau, and Y. Bengio, “On the properties of neural machine translation: Encoder-decoder approaches,” arXiv preprint arXiv:1409.1259, 2014.

[28] K. Yoon, “Convolutional neural networks for sentence classification [ol],” arXiv Preprint, 2014.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, “Attention is all you need,” Advances in neural information processing systems, vol. 30, 2017.