Received: 30 October 2024, Revised: 14 December 2024, Accepted: 15 December 2024, Online: 19 January 2025

DOI: https://dx.doi.org/10.55708/js0401001

Enhancing Python Code Embeddings: Fusion of Code2vec with Large Language Models

Long H. Ngo*, **Jonathan Rivalan**

Smile France, Asnières-sur-Seine, 92600, France

*Corresponding author: Long H. Ngo, Paris, France, Email: long.ngo@smile.fr

ABSTRACT: Automated code comprehension has recently become integral to software development. Neural networks, widely employed in natural language processing tasks, can capture the semantic meanings of language by representing it in vector form. Although programming code differs from natural language, we hypothesize that neural models can learn both the syntactic and semantic attributes inherent in code. This study presents an innovative approach to improve code representation and understanding for Python, building upon a previous work (code2vec extended with ASTminer). The novel method integrates embeddings from Large Language Models (LLMs) with code2vec vectors, aiming to align both semantic and syntactic information in code representations. We explore various fusion techniques, including simple concatenation, weighted sum, or attention-based mechanism, to combine LLM embeddings with code2vec vectors. We explore how semantic information from LLMs complements the structural information from code2vec, and discuss the potential impact of this synergy on software development practices. These findings open new directions for more accurate and adaptable code understanding models, with implications for improving documentation, code search, and overall software development efficiency.

KEYWORDS: Machine learning, Neural network, Large Language Model, Distributed representations, Code search

1. Introduction

In recent years, the field of automated code understanding has become increasingly crucial in software development. Certain problems in software development, such as assigning meaningful method names, highlight the need for concise semantic representations of code snippets. The core challenge lies in encoding code snippets in a way that captures semantically relevant information, transferable across multiple programs, and enables the prediction of properties like code labeling. This involves two key components: first, representing the code snippet in a manner that supports learning across different programs, and second, determining which parts of the representation are critical for property prediction and how to prioritize them.

This paper builds upon a previous study in [1] at the International Conference on Software Engineering and Artificial Intelligence (SEAI), by addressing its limitations and introducing novel methodology to improve code representation and search capabilities. Code representation presents challenges due to the need to capture both semantic and syntactic information in a format conducive to machine learning. The previous study, in [1], used a path-based representation technique to address two key tasks in software development: assigning semantic labels to code snippets and performing code searches. The path-based method represents a snippet by extracting paths from its abstract syntax tree (AST), capturing both the structure and semantics of the code, as demonstrated in previous research [2]. The first task focused on automating the prediction of a semantic label for a given code snippet. This is a challenging task because it requires learning complex relationships between the content of a method and a semantic label. Specifically, it involves condensing numerous expressions and statements within the method into a single descriptive label, which demands sophisticated techniques for code representation and classification [3]. The second task aimed to develop an efficient and effective search mechanism for locating code snippets based on query requirements. This is a critical need for developers, who often need to find and reuse existing code. Successful code search must be able to match a query against a large codebase and retrieve relevant code snippets. Addressing both of these tasks effectively would significantly enhance software development productivity and efficiency.

However, the work in [1] also revealed limitations, particularly in aligning the docstrings with extracted paths from the input code snippets. It might miss out on high-level semantic relationships. This challenge highlighted the need for more sophisticated approaches to code representation.

In this extended version, we present several significant advancements:

- Integration of large language models (LLMs) with code2vec to create more comprehensive code embeddings.
- Development of fusion techniques to combine semantic embeddings from LLMs with syntactic vectors from code2vec.

- Extensive experimental settings with various fusion strategies, including concatenation, weighted sum, and attention-based mechanisms.

By addressing these areas, we aim to push the boundaries of code representation, enabling more accurate and efficient automated understanding of Python code. This work not only builds upon the previous findings but also opens new directions for research in the intersection of natural language processing and programming language analysis.

The remainder of this paper is structured as follows: Section 2 reviews related work in code representation and analysis. Section 3 describes the work where code2vec was extended with ASTminer. Section 4 describes the present work, including the adaptation of code2vec for Python and the proposed fusion techniques. Section 5 discusses the implications of our findings. Section 6 concludes the paper with a summary of contributions and directions for future work.

2. Related works

The field of code representation and analysis has seen significant advancements in recent years, driven by the application of deep learning techniques traditionally used in natural language processing (NLP). While the conventional approach in NLP involves treating text as a linear sequence of tokens processed through neural networks [4, 5], this method has also been widely adopted for source code representation [6, 7, 8, 9, 10]. However, recent research has highlighted the potential benefits of leveraging the inherent structure of programming languages. In [11, 12, 13], the authors have demonstrated that structured representations can significantly enhance performance in various code analysis tasks.

The ability to predict program properties through learning from large-scale codebases has been a focus of numerous studies [6, 8, 9, 12, 14]. This capability opens up a wide range of applications in software engineering, including predicting names for program entities [11, 13, 15], code completion [16, 17], code summarization [6], code generation [18, 19, 20]. These applications, among others [21, 22], showcase the potential impact of advanced code representation techniques on various aspects of software development and maintenance, with minimal or no semantic analysis.

A significant breakthrough in this field came with the introduction of code2vec [3]. This innovative approach uses neural networks to learn distributed representations of code, addressing one of the most challenging problems in software engineering. Unlike traditional methods that rely primarily on static code features, code2vec uses a large corpus of code snippets to learn representations that capture the temporal dynamics of code changes. This results in more accurate and robust code representations.

The code2vec model consists of two main components: 1) An input encoder that converts code snippets into token sequences. These sequences are then used to train a neural network to predict the next tokens in the sequence, which is similar to a large language model. 2) A code vectorizer that generates continuous-valued vector representations from the output of the encoder.

These components work together to capture both syntactic and semantic information from the code, utilizing attention mechanisms to prioritize the most relevant aspects of each snippet. The resulting vector representations have proven effective in various software engineering tasks, including code completion, bug detection, and program synthesis. The architecture of the code2vec network is illustrated in Figure 1.

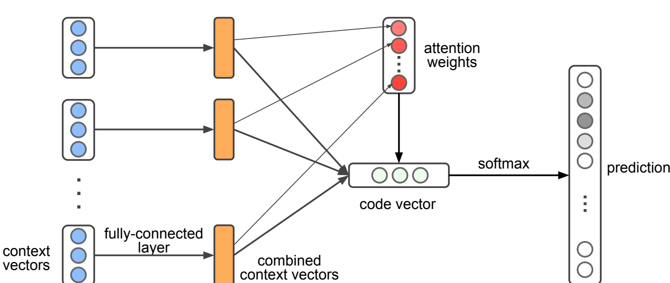


Figure 1: Original code2vec architecture. A fully-connected layer combines embeddings of each path-context. Attention weights are learned and used to compute a code vector, which predicts the label [3].

In [1], the authors adopted the code2vec neural network architecture [3], which was designed to learn low-dimensional vector representations, or embeddings, of the source code. In these vectors, the "meaning" of an entity is distributed across multiple components, allowing semantically similar entities to be mapped to nearby vectors. These embeddings enable efficient modeling of the relationship between code snippets and their semantic labels in a natural and efficient manner. Using the inherent structure of the source code, code2vec aggregated multiple syntactic paths into a single vector, leading to a more comprehensive and accurate representation of each code snippet.

The success of code2vec marks a significant departure from conventional code representation methods and has paved the way for further research in this area. Our work builds upon these foundations, extending the application of code2vec to Python and exploring novel ways to enhance its performance through integration with large language models.

3. Extended Code2vec with ASTminer for Python Code Embeddings

High-level view. In programming, a code snippet plays a pivotal role as it allows developers to write concise, reusable pieces of code applicable in different contexts. To better understand the construction of code snippets, it is essential to explore the concept of a bag of contexts. Each context within a code snippet is represented as a vector, referred to as the context vector. This vector is generated through a learning process that captures two important features of the context: (i) its semantic meaning and (ii) the attention it should receive. By aggregating these context vectors, a code vector is created, which can then be used for various downstream tasks.

3.1. Semantic labeling of code snippets

To extend code2vec for Python programming language, the ASTminer extractor [2] was employed instead of the JavaExtractor in the original code2vec [3]. ASTminer is an open source library that enables the extraction of path-based representations of codes, as shown in Figure 2. ASTminer parses the syntax tree of a code snippet to extract paths that summarize the structural and semantic essence of the code. Hence, it offers a reusable toolkit designed to simplify the task of modeling source code for machine learning algorithms, reducing the associated complexity and effort.

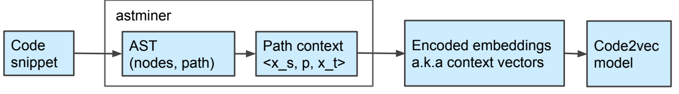


Figure 2: code2vec architecture for Python programming language.

The AST is essential in representing the syntactic structure of a program, selectively ignoring details such as punctuation, formatting or specific syntactic forms, while maintaining each node as a distinct syntactic unit. In the AST, each node represents a specific syntactic construct, such as variables, operations, or logical operators. The associated child nodes correspond to the subordinate elements related to the parent node. [2]

The ASTminer extractor uses this path-based representation to capture coding styles and structures, thereby encapsulating the logic of the code. The conversion of code into embeddings occurs in two steps: first, transforming the code into a vector, followed by combining these vectors with corresponding attention vectors. This prepares the model for further training. The architecture, depicted in Figure 3, mirrors the original code2vec embedding model.

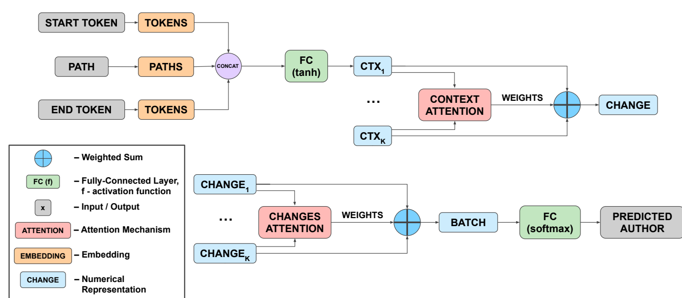


Figure 3: Overview of authorship attribution pipeline, which generates authorship-based embeddings of method changes. Furthermore, it highlights the significance of individual method changes in the authorship attribution. The method nodes and the corresponding attention weights are subsequently utilized to create author representation [23].

Following the approach in [23], the embeddings are constructed using the code and path tokens, then concatenated, and converted to a numerical representation using a fully connected (FC) activation function, as shown in Figure 3. These vectors are merged into batches where the weights of individual paths are combined, revealing the importance of each path. This pipeline is employed for predicting code titles, following the same principles as authorship attribution.

Analogous to [3], cross-entropy loss, which calculates the difference of the predicted distribution q and the "true" distribution p, is used during training phase. p assigns a value of 1 to the actual label 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:

$$\mathcal{H}p\|q \;-\; _{y\in Y} py \cdot logqy \;\; -logqy_{true} \tag{1}$$

where y_{true} is the actual label of the sample. The loss is the negative logarithm of qy_{true}, the probability that the model assigns to y_{true}. As qy_{true} tends to 1, the loss approaches zero. The further qy_{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}. During training, gradient descent is utilized to iteratively update the parameters by minimizing the loss function. The learned parameters are refined using backpropagation. In other words, the parameters are iteratively adjusted by taking small steps in the direction that reduces the loss, thereby optimizing the model.

Based on the study conducted in [3], the network model can be applied effectively to predict categories for unseen code. This method involves generating a code vector by leveraging the weights and parameters learned during the training phase. The prediction is then made by determining the closest target label. This predictive model offers significant potential for improving the accuracy and efficiency of categorizing software code snippets, supporting tasks such as software maintenance, bug detection, and code optimization.

3.2. Code search

This study also builds upon previous works [24, 25, 26] to design a neural search system that uses joint embeddings of code snippets and queries. Each type of input (either code or natural language) is processed using separate encoders, trained to map inputs into a shared vector space. The system is trained to ensure that related code and queries are embedded closely in this space, enabling efficient search and retrieval of relevant code snippets. Although there are more sophisticated models that account for multiple interactions between queries and code, this architecture utilizes a single vector per query/snippet, simplifying indexing and search tasks [26]. Figure 4 provides an overview of the model architecture used for the Python language in the code search task.

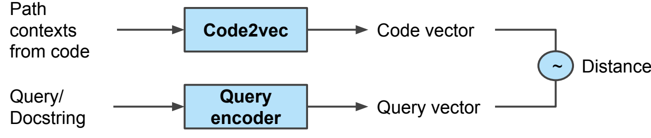


Figure 4: Text to code architecture based on code2vec model.

For this task, the extended code2vec model is used to encode Python code snippets. Queries in the form of docstrings are tokenized and passed through a query encoder, such as the Neural Bag of Words (NBoW) [27], Bidirectional RNN model [28], 1D Convolutional Neural Network [29], or Self-Attention [30]. Subsequently, the resulting token

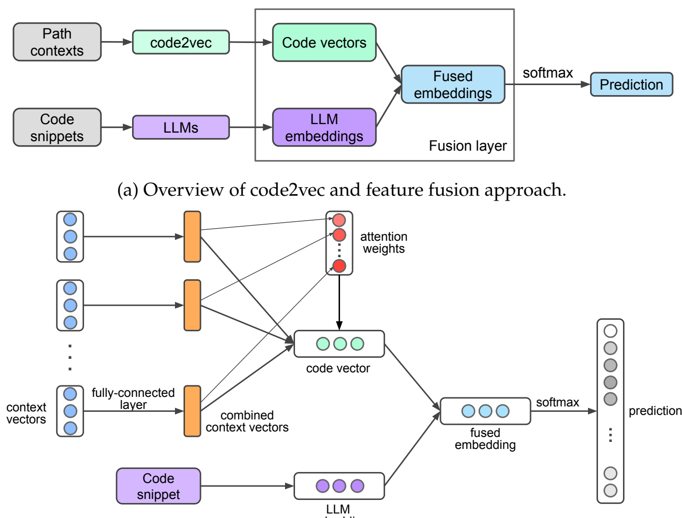
embeddings are combined into a sequence embedding using a pooling function, either mean or max pooling, and an attention-based weighted sum mechanism. The training process used a collection of N code and docstring pairs, denoted by c_i, d_i, alongside the instantiation of a code encoder E_c and a query encoder E_q. The objective is to minimize the following loss function:

$$-\frac{1}{N}{}_{i} log \frac{expE_c c_i{}^T E_q d_i}{{}_{j}\, expE_c c_j{}^T E_q d_i} \quad (2)$$

This loss encourages the inner product between matching code and docstring encodings pairs to be maximized, while minimizing the inner product between the query d_i and irrelevant code snippets c_j. During model evaluation, Mean Reciprocal Rank (MRR) is used to assess performance on the validation set. During the testing stage, the Annoy library[1], a high-speed, approximate nearest-neighbor indexing and search technique offered by Spotify, is used to index the entire CodeSearchNet Corpus. The index encompasses all functions in the corpus, including those without accompanying documentation. Annoy helps to achieve efficient and accurate retrieval [26].

4. Extended work: Code2vec and Embeddings Fusion

In this study, we propose a novel approach for enhancing the performance of code representation models by leveraging large language models (LLMs), such as Code Llama [31], Qwen2.5-coder [32], and code2vec for Python language. The integration of LLMs with traditional code representation techniques presents a promising way to enhance code understanding. Our approach combines the semantic richness of LLM embeddings with the structural insights of syntactic code2vec vectors. This fusion aims to create a more comprehensive and nuanced representation of code snippets. Figure 5 depicts the architecture of the embeddings fusion approach adapted with the code2vec model.



(a) Overview of code2vec and feature fusion approach.

(b) Updated code2vec architecture.

Figure 5: Proposed architecture of code2vec with fused embeddings using LLMs.

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

We utilize Code Llama - 7B [31], a variant of the Llama model fine-tuned on code, to generate semantic embeddings. Code Llama represents the cutting edge of publicly available large language models (LLMs) for code-related tasks. It offers the potential to enhance developer workflows by improving efficiency and speed, while also reducing the barriers for individuals new to programming. Additionally, Code Llama could serve as a valuable tool for productivity and education, assisting programmers in producing more reliable and well-documented software.

We used the CodeSearchNet corpus for training and evaluation. This dataset contains about 1.1 million Python functions, spanning diverse domains and programming paradigms. Each code snippet is processed through Code Llama to produce a high-dimensional vector capturing contextual and semantic information. Concurrently, we process the same code snippets through our adapted code2vec model for Python, which generates vectors representing the structural and syntactic features of the code.

To achieve this fusion, we explore various techniques to integrate LLM embeddings and code2vec code vectors into a single, unified representation. These techniques include:

1. Simple concatenation: A straightforward approach where Code Llama embeddings and code2vec vectors are simply combined end-to-end. This method preserves all information but increases dimensionality.

$$fused_vector \;\; llm_embedding; code2vec_vector \quad (3)$$

2. Weighted sum: We apply learnable weights to each embedding type before summation, allowing the model to adjust the importance of semantic versus syntactic information.

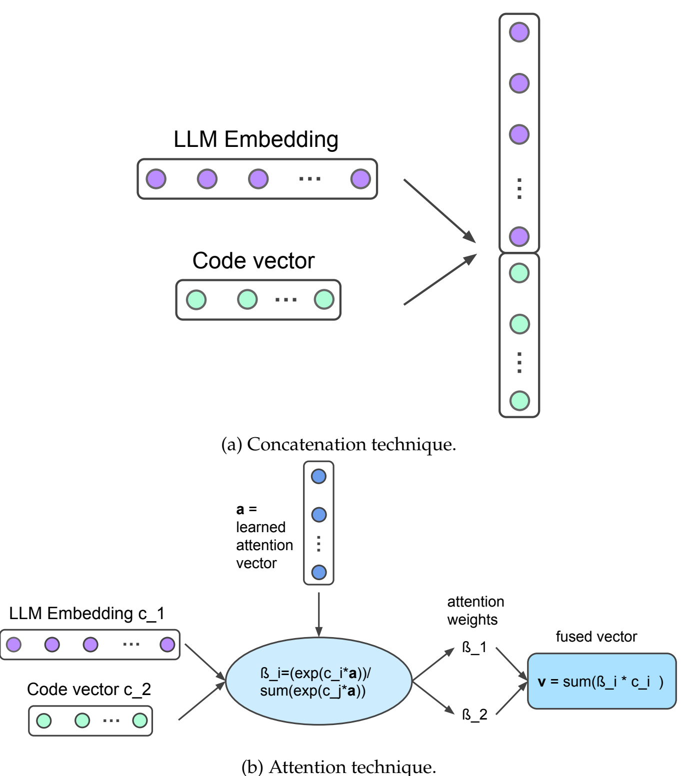
$$fused_vector \;\; w1 * llm_embedding \;\; w2 * code2vec_vector \quad (4)$$

3. Attention-based mechanisms: Inspired by transformer architectures, we implement a multi-head attention mechanism. This allows dynamic focus on different aspects of each embedding based on the specific code context. On the other hand, it allows the model to dynamically weight semantic and syntactic information based on task relevance, resolving the issue of fixed representation alignment by enabling the model to prioritize critical information.

$$attention_output \;\; MultiHeadAttentionllm_embedding, code2vec_vector$$
$$fused_vector \;\; FeedForwardattention_output \quad (5)$$

Each method offers different advantages in terms of balancing the contribution of each type of embedding to the final representation. For instance, concatenation allows for a straightforward integration of both embeddings, while weighted sum can modulate the influence of each embedding type, and attention-based fusion offers the potential to dynamically adjust the focus on relevant features during training and inference. The fused representation is then employed in both the training and testing phases of code2vec, with the goal of improving performance on tasks such as code labeling and code search.

Figure 6 illustrates examples of two fusion techniques: concatenation and attention-based fusion. These visualizations provide insight into how each method integrates the embeddings and the potential impact on the resultant code representation.



(a) Concatenation technique.

(b) Attention technique.

Figure 6: Examples of 2 feature fusion techniques used in our feature fusion approach.

5. Discussion

The integration of LLM embeddings with code2vec vectors shows a significant theoretical advancement in code representation techniques, which offers promising directions for combining semantic and syntactic information in code understanding tasks. Here, we discuss the implications of these findings and their potential impact on the field of automated code understanding.

Our findings highlight the complementary nature of LLM embeddings and code2vec vectors, demonstrating how these two approaches can be combined to enhance code representation. LLM embeddings capture rich semantic information and contextual nuances of code, while code2vec vectors excel at representing syntactic structure. The synergy between these two types of information allows our model to develop a more comprehensive understanding of code snippets. The relevance of semantic versus syntactic information may vary depending on the specific code snippet or task at hand. The multi-head attention mechanism in our fusion approach allows the model to dynamically focus on different aspects of the code representation. This adaptability is particularly valuable for handling various programming paradigms, coding styles, or specific tasks.

The integration of LLM embeddings, which are pre-trained on vast amounts of code data, with task-specific code2vec vectors opens up new possibilities for transfer learning in code analysis. This approach allows models to benefit from the broad knowledge captured by LLMs while still maintaining the ability to fine-tune on specific tasks or codebases. This transfer learning capability could be particularly beneficial for organizations with limited labeled data or for tackling niche programming languages or domain-specific coding patterns. It suggests a path towards more generalizable code understanding models that can quickly adapt to new contexts.

The fusion of LLM embeddings and code2vec vectors also increases the computational complexity of the model. The attention-based fusion, in particular, adds a non-trivial amount of computation to the process. As we scale this approach to larger codebases or real-time applications, careful consideration must be given to balancing performance gains with computational efficiency. Future work could explore techniques for model compression or distillation to make this approach more feasible for resource-constrained environments or large-scale deployments.

The enhanced capabilities in code labeling and search have the potential to significantly impact software development practices. More accurate code labeling could lead to improved auto-documentation tools, helping maintain up-to-date and accurate code documentation. Enhanced code search capabilities could boost developer productivity by making it easier to find and reuse existing code snippets, potentially reducing duplication and improving code quality. Furthermore, these advancements could contribute to the development of more sophisticated code recommendation systems, assisting developers in writing more efficient, readable, and maintainable code.

6. Conclusion

This study has presented a significant extension to the previous work on code representation using code2vec and ASTminer for Python. By integrating large language model (LLM) embeddings with code2vec vectors, we have demonstrated a novel approach to capturing both semantic and syntactic information in code representations. While the current work focuses on architectural design and theoretical analysis, it lays important groundwork for future research in automated code understanding.

Our key theoretical contributions include: the development of fusion techniques, particularly an attention-based mechanism, to combine LLM embeddings with code2vec vectors; insights into the synergistic relationship between semantic information from LLMs and structural information from code2vec, and how this synergy can be leveraged in a novel architecture for better code understanding; theoretical foundation for improved code understanding models in future empirical studies.

These advancements address some of the limitations identified in the previous code2vec work, particularly in aligning semantic information (such as docstrings) with structural code representations. The fusion approach we developed offers a more robust and flexible way to represent code, adapting to the specific needs of different tasks and code structures.

Despite the promising aspects, several limitations and areas for further research remain:

- Language specificity: Our current work focuses on

Python. Further research is needed to assess the generalizability of this approach to other programming languages.

- Lack of empirical validation: The current study focuses on theoretical foundations and architectural design, without experimental results to validate performance claims. Comparative analysis with existing code representation techniques remains to be conducted.

- Model interpretability: While the attention mechanism provides some insight into the model's decision-making process, further work is needed to improve the interpretability of the fused representations.

- Fine-grained code understanding: Future research could explore how this approach performs on more fine-grained tasks, such as variable naming or type inference.

- Temporal aspects of code: Investigating how to incorporate information about code evolution and version history into our fused representations could provide additional valuable insights.

In conclusion, our work on fusing LLM embeddings with code2vec vectors represents a significant step forward in the field of automated code understanding. By leveraging both semantic and syntactic information, this approach opens up new possibilities for more accurate and versatile code understanding models, with potential far-reaching implications for software development practices and tools. As we continue to refine these techniques and explore their applications, we potentially anticipate a profound impact on the landscape of software development. Through rigorous experimental validation and continued refinement, these theoretical contributions can evolve into practical tools for improving code comprehension and development efficiency.

Conflict of Interest The authors declare no conflict of interest.

References

[1] L. H. Ngo, V. Sekar, E. Leclercq, J. Rivalan, "Exploring code2vec and astminer for python code embeddings", "2023 IEEE 3rd International Conference on Software Engineering and Artificial Intelligence (SEAI)", pp. 53–57, IEEE, 2023, doi:10.1109/SEAI59139.2023.10217505.

[2] V. Kovalenko, E. Bogomolov, T. Bryksin, A. Bacchelli, "Pathminer: a library for mining of path-based representations of code", "Proceedings of the 16th International Conference on Mining Software Repositories", pp. 13–17, IEEE Press, 2019, doi:10.1109/MSR.2019.00015.

[3] U. Alon, M. Zilberstein, O. Levy, E. Yahav, "code2vec: Learning distributed representations of code", *Proceedings of the ACM on Programming Languages*, vol. 3, no. POPL, pp. 1–29, 2019, doi:10.1145/3290353.

[4] S. Hu, Y. Zuo, L. Wang, P. Liu, "A review about building hidden layer methods of deep learning", *Journal of Advances in Information Technology*, vol. 7, no. 1, 2016, doi:10.12720/jait.7.1.58-63.

[5] Y. Sakai, Y. Eto, Y. 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*, 2022, doi:10.12720/jait.13.1.1-7.

[6] M. Allamanis, H. Peng, C. Sutton, "A convolutional attention network for extreme summarization of source code", "International conference on machine learning", pp. 2091–2100, PMLR, 2016, doi:10.48550/arXiv.1602.03001.

[7] M. White, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, "Toward deep learning software repositories", "2015 IEEE/ACM 12th Working Conference on Mining Software Repositories", pp. 334–345, IEEE, 2015, doi:10.1109/MSR.2015.33.

[8] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, "Learning natural coding conventions", "Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering", pp. 281–293, 2014, doi:10.1145/2635868.2635883.

[9] M. Allamanis, C. Sutton, "Mining source code repositories at massive scale using language modeling", "2013 10th working conference on mining software repositories (MSR)", pp. 207–216, IEEE, 2013, doi:10.1109/MSR.2013.6624029.

[10] D. Movshovitz-Attias, W. Cohen, "Natural language models for predicting programming comments", "Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers)", pp. 35–40, 2013, doi:10.18653/v1/P13-2007.

[11] U. Alon, M. Zilberstein, O. Levy, E. Yahav, "A general path-based representation for predicting program properties", *ACM SIGPLAN Notices*, vol. 53, no. 4, pp. 404–419, 2018, doi:10.1145/3192366.3192412.

[12] P. Bielik, V. Raychev, M. Vechev, "Phog: probabilistic model for code", "International conference on machine learning", pp. 2933–2942, PMLR, 2016, doi:10.48550/arXiv.1602.05259.

[13] V. Raychev, M. Vechev, A. Krause, "Predicting program properties from" big code"", *ACM SIGPLAN Notices*, vol. 50, no. 1, pp. 111–124, 2015, doi:10.1145/2676726.2677009.

[14] V. Raychev, P. Bielik, M. Vechev, "Probabilistic model for code with decision trees", *ACM SIGPLAN Notices*, vol. 51, no. 10, pp. 731–747, 2016, doi:10.1145/2983990.2984041.

[15] M. Allamanis, E. T. Barr, C. Bird, C. Sutton, "Suggesting accurate method and class names", "Proceedings of the 2015 10th joint meeting on foundations of software engineering", pp. 38–49, 2015, doi:10.1145/2786805.2786849.

[16] V. Raychev, M. Vechev, E. Yahav, "Code completion with statistical language models", "Proceedings of the 35th ACM SIGPLAN conference on programming language design and implementation", pp. 419–428, 2014, doi:10.1145/2594291.2594321.

[17] A. Mishne, S. Shoham, E. Yahav, "Typestate-based semantic code search over partial programs", "Proceedings of the ACM international conference on Object oriented programming systems languages and applications", pp. 997–1016, 2012, doi:10.1145/2384616.2384698.

[18] M. Amodio, S. Chaudhuri, T. W. Reps, "Neural attribute machines for program generation", *arXiv preprint arXiv:1705.09231*, 2017, doi:10.48550/arXiv.1705.09231.

[19] Y. Lu, S. Chaudhuri, C. Jermaine, D. Melski, "Data-driven program completion", *arXiv preprint arXiv:1705.09042*, 2017, doi:10.48550/arXiv.1705.09042.

[20] C. Maddison, D. Tarlow, "Structured generative models of natural source code", "International Conference on Machine Learning", pp. 649–657, PMLR, 2014, doi:10.48550/arXiv.1401.0514.

[21] M. Allamanis, E. T. Barr, P. Devanbu, C. Sutton, "A survey of machine learning for big code and naturalness", *ACM Computing Surveys (CSUR)*, vol. 51, no. 4, pp. 1–37, 2018, doi:10.1145/3212695.

[22] M. Vechev, E. Yahav, *et al.*, "Programming with "big code"", *Foundations and Trends® in Programming Languages*, vol. 3, no. 4, pp. 231–284, 2016, doi:10.1561/2500000028.

[23] V. Kovalenko, E. Bogomolov, T. Bryksin, A. Bacchelli, "Building implicit vector representations of individual coding style", "Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops", pp. 117–124, 2020, doi:10.1145/3387940.3391461.

[24] X. Gu, H. Zhang, S. Kim, "Deep code search", "Proceedings of the 40th International Conference on Software Engineering", pp. 933–944, 2018, doi:10.1145/3180155.3180167.

[25] 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, doi:10.1561/1500000061.

[26] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, "CodeSearchNet challenge: Evaluating the state of semantic code search", *arXiv preprint arXiv:1909.09436*, 2019, doi:10.48550/arXiv.1909.09436.

[27] I. Sheikh, I. Illina, D. Fohr, G. Linares, "Learning word importance with the neural bag-of-words model", "Proceedings of the 1st Workshop on Representation Learning for NLP", pp. 222–229, 2016, doi:10.18653/v1/W16-1626.

[28] K. Cho, B. Van Merriënboer, D. Bahdanau, Y. Bengio, "On the properties of neural machine translation: Encoder-decoder approaches", "Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation", 2014, doi:10.3115/v1/W14-4012.

[29] K. Yoon, "Convolutional neural networks for sentence classification [ol]", *arXiv Preprint*, 2014, doi:10.48550/arXiv.1408.5882.

[30] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, "Attention is all you need", *Advances in neural information processing systems*, vol. 30, 2017, doi:10.48550/arXiv.1706.03762.

[31] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez, *et al.*, "Code llama: Open foundation models for code", *arXiv preprint arXiv:2308.12950*, 2023, doi:10.48550/arXiv.2308.12950.

[32] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Dang, *et al.*, "Qwen2. 5-coder technical report", *arXiv preprint arXiv:2409.12186*, 2024, doi:10.48550/arXiv.2409.12186.

Copyright: This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY-SA) license (`https://creativecommons.org/licenses/by-sa/4.0/`).

LONG H. NGO has done his B.E. degree in electrical and electronics engineering from Ho Chi Minh City University of Technology in 2016. He has received his master's degree in imaging and networks from Université Sorbonne Paris Nord, France, in 2017. He has completed his PhD degree in machine learning and image processing from Université Sorbonne Paris Nord, France, in 2021.

His research interests include machine learning, image processing, computer vision, and NLP.

JONATHAN RIVALAN is R&D manager at Smile, French open source services integrator based in various locations in Europe. His main line of work targets workflow automation, digital infrastructures optimization and advanced HCI developments. He is currently participating and leading national and European R&D programs, which results are available as open source solutions, such as Rating Operator (metrics transformation towards KPIs) and Palindrome.js (multidimensional monitoring for distributed systems).