4-2024

Encoding version history context for better code representation

Huy NGUYEN

Christoph TREUDE
Singapore Management University, ctreude@smu.edu.sg

Patanamon THONGTANUNAM

Encoding Version History Context for Better Code Representation

Huy Nguyen
The University of Melbourne
Australia
huyxuan.nguyen@student.unimelb.edu.au

Christoph Treude
Singapore Management University
Singapore
ctreude@smu.edu.sg

Patanamon Thongtanunam
The University of Melbourne
Australia
patanamon.t@unimelb.edu.au

Abstract

With the exponential growth of AI tools that generate source code, understanding software has become crucial. When developers comprehend a program, they may refer to additional contexts to look for information, e.g. program documentation or historical code versions. Therefore, we argue that encoding this additional contextual information could also benefit code representation for deep learning. Recent papers incorporate contextual data (e.g. call hierarchy) into vector representation to address program comprehension problems. This motivates further studies to explore additional contexts, such as version history, to enhance models' understanding of programs. That is, insights from version history enable recognition of patterns in code evolution over time, recurring issues, and the effectiveness of past solutions. Our paper presents preliminary evidence of the potential benefit of encoding contextual information from the version history to predict code clones and perform code classification. We experiment with two representative deep learning models, ASTNN and CodeBERT, to investigate whether combining additional contexts with different aggregations may benefit downstream activities. The experimental result affirms the positive impact of combining version history into source code representation in all scenarios; however, to ensure the technique performs consistently, we need to conduct a holistic investigation on a larger code base using different combinations of contexts, aggregation, and models. Therefore, we propose a research agenda aimed at exploring various aspects of encoding additional context to improve code representation and its optimal utilisation in specific situations.

CCS Concepts

• **Computing methodologies** → **Neural networks**; • **Software and its engineering** → **Reusability**.

Keywords

Source code representation, additional context, version history

1 Introduction

Understanding software becomes increasingly crucial to the development and application of technology to satisfy user demands [7]. Understanding complex software systems is often challenging. These challenges are compounded by time constraints and constantly changing business requirements [25]. Artificial Intelligence (AI) can help developers generate code quickly and efficiently [1, 6, 22], contributing to the growth of new source code in various domains, e.g. education [3, 21]. We argue that understanding software has become more essential than ever. However, the effectiveness of these AI tools strongly depends on their ability to comprehend the given context and the generated source code [20]. Therefore, improving AI's ability to understand source code and contextual information is critical to ensure that the outputs of these tools are reliable [32]. Although recent studies have advanced source code representation, they also reveal significant research gaps. Recent articles have predominantly focused on harnessing deep learning techniques for software comprehension tasks. However, they have often neglected the full utilisation of additional contexts that can significantly improve performance [31]. Although these studies shed light on various aspects, they often rely on relatively old and simplistic datasets, e.g. OnlineJudge or BigCloneBench [23]. These datasets are reliable regarding the volume or annotation of data, but they are limited in terms of additional contexts. Since the popularity of code hosting platforms, e.g. GitHub, crawling other contextual information has become easier [29]. For example, Wang and Lo [30] claim that putting together version history, similar reports, and structure can help locate relevant buggy files. However, the proposed statistical method uses only the latest version of the code and does not incorporate it into the code representation for downstream tasks. We argue that deep learning models may also comprehend source code better if they can access information beyond source code (e.g. version history).

Figure 1 illustrates a motivating example of a code clone pair of two Java methods and their historical versions from two open-source projects on GitHub. Their historical versions can help recognise code clones since their history shows more commonalities than their current versions [1] [2] [3] [4]. The takeaway from the example is that using version history along with source code could be beneficial for certain SE tasks, e.g. code clone detection or bug localisation.

We present the initial results of encoding version history context for better code representation into two representative deep learning models, i.e. ASTNN [29, 34] and CodeBERT [4]. Experimental results on two software engineering tasks, i.e. Code Clone Detection and Code Classification, indicate that adding multiple historical versions of code to the final representation improves the models' performance compared to using only the original code snippet.

Our experimental results show that concatenating only version history information can boost the model performance (F1 score) by 15% (from 0.667 to 0.769) for CodeBERT and by 7% (from 0.824 to 0.880) for ASTNN. We can achieve even better results when using multiple additional contexts (version history, call hierarchy, and the number of existing days). Namely, CodeBERT's F1 score for Code Clone Detection increases by 27% to 0.846 when we concatenate the representation of the absolute difference between the two methods with both version history context and number of days information. However, we also observe poorer performance in some scenarios.

[1] project: jdk11 | file: DoubleAdder.java | method: sumThenReset() | current version
[2] project: jdk11 | file: DoubleAdder.java | method: sumThenReset() | historical version
[3] project: guava | file: LongAdder.java | method: sumThenReset() | current version
[4] project: guava | file: LongAdder.java | method: sumThenReset() | historical version

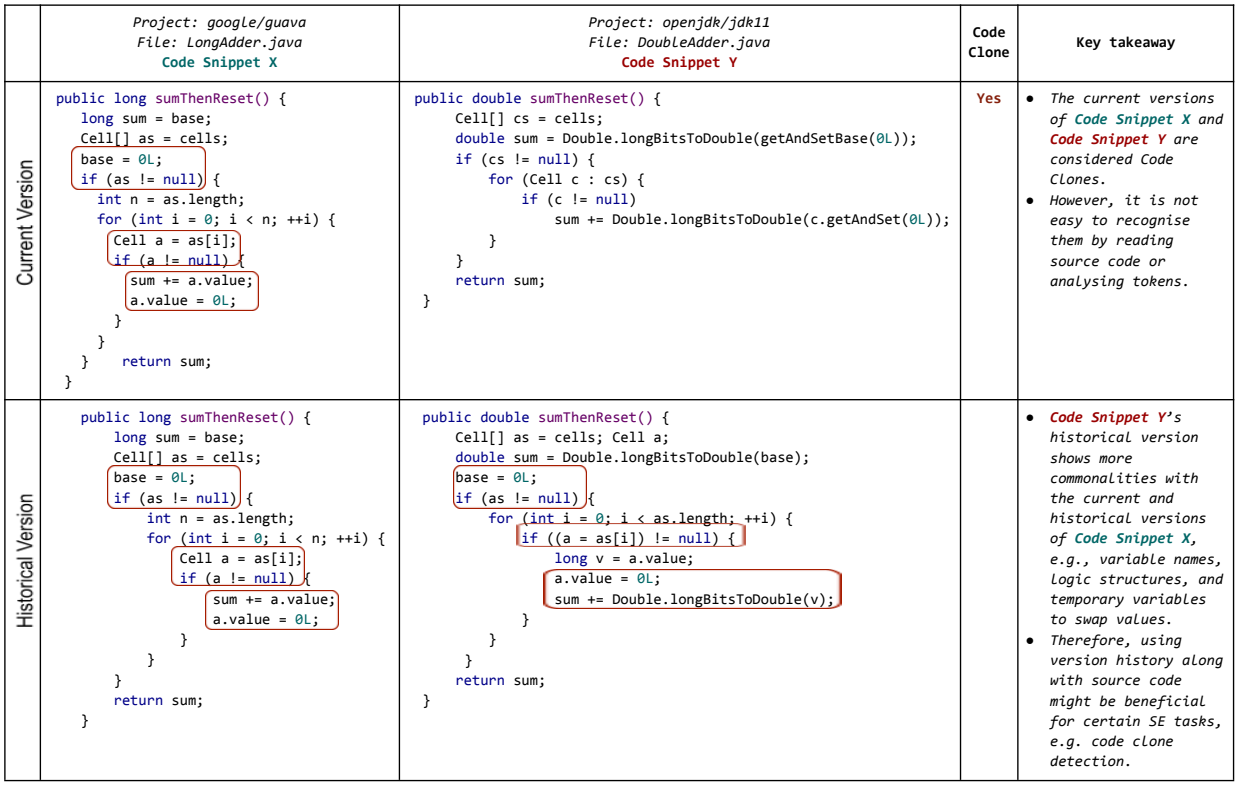

Figure 1: A motivating example of using Version History to detect code clones.

We conclude that the version history context can improve the code representation for deep learning, but how to best use it requires further investigation. From these findings, we propose our research agenda to explore various aspects of encoding additional contexts, especially version history, to improve code representation and its optimal utilisation in specific situations.

2 Related Work

Our research combines knowledge from two aspects, i.e. source code representation and program comprehension.

Source Code Representation. Source code, written by programmers or generated by tools, is initially a text-encoded representation of a program. Therefore, it can be converted into various forms of representation. An effective code representation could benefit program comprehension tasks, such as program repair or code clone detection [17].

Determining the appropriate representation of source code is thus a crucial aspect of many software engineering tasks. Recent papers have introduced popular techniques for addressing different downstream tasks, including graph-based, tree-based, or token-based techniques. In 2019, a well-known neural-based code representation for code, called ASTNN, used tree-based CNN to transform AST sub-trees into vector format [29, 34]. Many other approaches use tree- or graph-based representations for bug detection or program classification [9, 14, 35].

Furthermore, existing studies on transformer-based models for programming languages use a tokeniser to convert the text input to numerical representation that can be processed by the model [4]. Other studies performed deep learning tasks using code representation, incorporating high-level semantic and low-level syntactic information [11]. Hybrid representation techniques are becoming increasingly popular, where more than one code representation can be used [15, 23, 29].

However, there is an existing research gap on how to improve the input of representation techniques. We argue that many available programming artefacts, e.g. version history, that go beyond source code could benefit code representation and downstream tasks.

Context Considered by Humans During Program Comprehension. Understanding software is a term in software engineering research that encompasses both the human and deep learning perspective [25]. We argue that additional contexts from the software development process support developers in comprehending source code; therefore, it may work similarly for deep learning.

Maletic and Marcus [16] claim that multiple software artefacts, with semantic and structural context, provide valuable support for program comprehension. Furthermore, Kulkarni and Varma [13] indicated that the cues derived from different programming contexts help establish the relevance of information for software engineering tasks. In addition, developers may be interested in task-related software artefacts to understand program logic rather than using the entire source code [24].

Most recent studies on deep learning only use the source code itself as an input [15, 31]. However, few papers focus on exploring different types of data [10, 11, 22, 29], for instance, version history or execution traces. The main reason could be the challenges of mining data from artefacts produced as part of the software engineering process or the limited computational resources to train the models [23]. However, the fast evolution of foundation models, such as Large Language Models (LLMs) like GPT-4, could help to overcome current challenges in proposing new code representation techniques [1, 3, 21, 22].

3 Preliminary Study

This section outlines our preliminary study approach to investigate the feasibility of encoding version history to source code representation and how it improves deep learning's performance in software engineering tasks. To conduct the study, we 1) mine a version history in a code repository, 2) explore suitable aggregation techniques, and 3) evaluate the performance of two well-known models (ASTNN and CodeBERT) on downstream tasks. Lastly, we explore combinations of multiple contexts and their effectiveness.

To evaluate the benefits of adding version history to source code representation, we set out the following three research questions:

RQ1 What is the impact of encoding of version history on the performance of deep learning models?
RQ2 What is the impact of different aggregation techniques on the representation of source code and its version history context?
RQ3 How does combining multiple additional contexts into source code representation impact deep learning models?

Data Collection. Since the version history we need is only available on the source hosting system, we need a dataset that contains the repository information. Hence, we use SeSaMe [12], a dataset of semantically similar Java methods from 11 software projects, all available on code hosting platforms (e.g. GitHub). The version history context refers to all changes to a method (or a code fragment) during its lifetime, and each version is a particular snapshot. Thus, every method has a version history.

To extract version history data, we use *PyDriller* [26] to walk through all commits based on the provided *commit hash*. We also use *Lizard* to analyse the source code and extract only the methods from the SeSaMe dataset [28]. Lastly, we keep only versions in which the method's source code was changed. Along with version history, we extract call hierarchy (caller and callee) [29] and number of days as additional contexts for the experiment. The number of days contains a numerical value that describes how long a method existed in the repository. We argue that this numerical information might indicate some relationship between the two methods and help to detect code clones.

Table 1 above introduces descriptive statistics of our dataset. We extracted 10,531 code versions of 1,679 unique Java methods from 11 open-source projects. The number of methods per project and the number of versions per method are diverse, ranging from an average of 1 version/method (*trove*) to an average of 26.71 versions/method (*checkstyle*). A method's lifetime varies from 17 to 6,334 days, and the average number of changed lines/version is 3.94 lines/version.

Table 1: Statistical Analysis of the Dataset

GitHub Project	# of methods	Avg # of version /method	Avg # of changed lines /version	Min|Max|Avg # of days
caffeine	63	2.25	1.14	196 | 1,328 | 1,174
checkstyle	52	26.71	3.57	125 | 1,665 | 989
commons-collections	81	1.51	0.56	273 | 1,994 | 1,883
commons-lang	57	9.51	1.76	535 | 3,198 | 2,815
commons-math	93	1.51	3.51	474 | 1,290 | 1,234
deeplearning4j	212	1.39	1.44	17 | 140 | 136
eclipse.jdt.core	178	22.80	4.41	200 | 6,334 | 4,317
freemind	87	2.29	4.71	299 | 2,742 | 2,335
guava	156	3.96	2.53	378 | 2,720 | 2,184
openjdk11	688	4.38	4.65	58 | 413 | 335
trove	12	1.00	-	1,593 | 1,593 | 1,593

Encoding and Aggregation. We selected two popular model architectures, ASTNN and CodeBERT, to evaluate the impact of encoding version history context into source code representation in Code Clone detection. ASTNN uses tree-based architecture, allowing the model to capture hierarchical structural information based on its understanding of source code patterns [29, 34]. CodeBERT is constructed from a bimodal pre-trained model using six programming languages [4, 33]. Unlike ASTNN, taking input as ASTs, CodeBERT accepts code snippets.

To explore different combinations of contexts to observe their interaction within code representation, we design our model experiment into three steps, including (1) Encoding, (2) Aggregation, and (3) Model Training.

First, we use the corresponding technique from ASTNN and CodeBERT to convert the method's source code and its additional contexts into vector representation [4, 34]. The output derived from a method's source code is a single vector, and the output derived from version history contains a long vector that consists of multiple vectors parsed from historical versions [29, 33]. We follow a recent study [29] to select the longest caller and callee from the call hierarchy to produce two separate vectors for caller and callee, respectively. Also, the number of days is also transformed into a vector. After the Encoding step, we have five vectors (or numerical representations) in total representing information of the method's current source code, multiple historical versions, caller, callee, and number of days.

Secondly, to combine selections of vector representations from the previous step, we select three aggregation methods: 1) concatenation, 2) max-pooling, and 3) concatenation of absolute difference. These are the aggregation methods that are suitable for both vector representations (for ASTNN) and text representations (for CodeBERT) [5]. In our preliminary study, we select these three methods since they are well-established approaches within our problem domain [29]. Finally, we pass the output from the Aggregation step into a linear layer and a sigmoid layer (as the Model training step) to determine whether the two methods are code clones.

Concatenation refers to merging the representation of source code with representations of its additional contexts [5]. We compute the absolute value of the difference between concatenated vectors and pass it into a linear layer to predict cloned code.

Max-pooling refers to the pooling technique in deep neural networks [5], where we select a vector with the highest values in each

dimension among all input vectors from two methods, composed of the method source code and the version history context. Then, we pass it to a linear layer.

Concatenation of absolute difference computes the difference between vectors of two methods' source code first, then merges the output vector with all remaining additional context vectors before passing it to the linear layer function [29].

Please refer to our online appendix [18] for all three aggregation scenarios that take the above representation as input.

In Code Classification, we only have concatenation and max-pooling scenarios because the input contains only one method. After aggregating the vectors for the respective scenarios, we pass them to a softmax layer for a classifier, where the output is an array of probabilities for each label in the dataset.

Experimental Setup. The SeSaMe dataset allows us to experiment with two software engineering tasks [34]:

Code Clone Detection: We compute the label from human annotation data on code clone pairs in the SeSaMe dataset [29]. Each pair of codes contains a binary label (0/1), which was constructed based on weights to reflect high, medium, and low confidence. The evaluation metric for this task is the F1 score.

Code Classification: The original dataset contains the 11 GitHub project names associated with code snippets. We use this information as labels for the classification task. For the classification task, we use Accuracy as the evaluation metric.

Training settings. To ensure fair comparison of our experiment with baseline performance, we also adopt training, validation, and testing sets with an 80:10:10 ratio. We also use the same hyperparameters settings, workflows and loss functions for all models. We selected the models with the best results on the validation set.

4 Experimental Results

Table 2 and 3 present the performance of ASTNN and CodeBERT when combining version history with source code representation for Code Clone Detection and Code Classification tasks, respectively. We now present results to answer each research question.

RQ1: Impact of Adding Version History to Code Representation. We compare the ASTNN's and CodeBERT's performance between (i) without additional context (baseline) and (ii) with version history using the concatenation aggregation.

Table 2 shows that for the Code Clone Detection task, the F1 score of the ASTNN and CodeBERT models with version history (using concatenation aggregation) increases by 7% and 15% compared to the models without additional context. Similarly, Table 3 shows that for the Code Classification task, the accuracy of the ASTNN and CodeBERT models with version history increases by 6% and 4%, respectively. These results suggest that encoding version history context, which contains multiple source code versions, helps deep learning perform better than without context.

RQ2: Impact of Different Aggregation Techniques. We select the experiment of encoding version history context to source code representation in both software engineering tasks. Among the proposed scenarios, no technique is always better than others in all models and tasks.

Table 2: Code Clone Detection using Version History or Multiple Contexts

		Context(s)	Aggregation	P	R	F1	%F1↑
ASTNN		Without Context		0.913	0.750	**0.824**	
	*	Version History	Concatenation	1.000	0.786	**0.880**	**7%**
			Max-pooling	0.821	0.821	0.821	0%
			Diff & Concat	0.833	0.714	0.769	-6%
	*	Call Hierarchy	Concatenation	0.913	0.750	0.824	0%
			Max-pooling	0.885	0.821	0.852	4%
			Diff & Concat	0.955	0.750	0.840	2%
	**	Version History	Concatenation	0.885	0.821	0.852	4%
		+ Call Hierarchy	Max-pooling	0.852	0.821	0.836	2%
			Diff & Concat	0.875	0.750	0.808	-2%
	**	Version History	Concatenation	1.000	0.786	**0.880**	**7%**
		+ No. of Days	Max-pooling	0.852	0.821	0.836	2%
			Diff & Concat	0.913	0.750	0.824	0%
	**	Version History	Concatenation	0.885	0.821	0.852	4%
		+ Call Hierarchy	Max-pooling	0.920	0.821	0.868	6%
		+ No. of Days	Diff & Concat	0.864	0.679	0.760	-7%
CodeBERT		Without Context		0.655	0.679	**0.667**	
	*	Version History	Concatenation	0.778	0.750	0.764	15%
			Max-pooling	0.714	0.714	0.714	7%
			Diff & Concat	0.833	0.714	**0.769**	**15%**
	*	Call Hierarchy	Concatenation	0.821	0.821	0.821	23%
			Max-pooling	0.864	0.679	0.760	14%
			Diff & Concat	0.840	0.750	0.792	19%
	**	Version History	Concatenation	0.840	0.750	0.792	19%
		+ Call Hierarchy	Max-pooling	0.800	0.714	0.755	13%
			Diff & Concat	0.815	0.786	0.800	20%
	**	Version History	Concatenation	0.679	0.679	0.679	2%
		+ No. of Days	Max-pooling	0.643	0.643	0.643	-4%
			Diff & Concat	0.917	0.786	**0.846**	**27%**
	**	Version History	Concatenation	0.875	0.750	0.808	21%
		+ Call Hierarchy	Max-pooling	0.857	0.643	0.735	10%
		+ No. of Days	Diff & Concat	0.846	0.786	0.815	22%

*: Single Context | **: Multiple Contexts

In Code Clone Detection, ASTNN with the concatenation of version history to code representation achieves the highest F1 score of 0.880 (7% increase compared to the baseline); nevertheless, both concatenation scenarios in CodeBERT achieve 15% improvement. The max-pooling scenario gains the lowest improvement in both models, only 7% with CodeBERT and even 0% with ASTNN. On the contrary, the experiment with the Code Classification task displays another tendency. CodeBERT with max-pooling scenarios achieved a 7% improvement (0.852 in accuracy). In ASTNN, concatenating techniques increase accuracy by 6%, from 0.583 (baseline) to 0.617.

The reason why no aggregation technique outperforms others in all experiments can be explained by how we handle multiple historical code versions to create the final representation. Each method may have one or hundreds of versions during its lifetime. Concatenation and Concatenation of Absolute Difference merge vector representations and then rely on the model's learning capability. ASTNN and CodeBERT have limitations on the input length; therefore, a method with too many versions may be truncated. On the other hand, max-pooling relies on selecting the maximum value within the pool of vector representations. Therefore, some critical information in the unselected representation may be dismissed.

RQ3: Impact of Adding Multiple Artefacts to Code Representation. In this section, we aim to answer RQ3 on the impact of combining multiple programming artefacts, i.e. version history with other contextual information (refer to ** in Tables 2 and 3).

Table 3: Code Classification with Version History or Multiple Contexts

		Context(s)	Aggregation	Acc	P	R	%Acc↑
ASTNN		Without Context		**0.583**	0.515	0.414	
	*	Version History	Concatenation	**0.617**	0.557	0.471	**6%**
			Max-pooling	0.591	0.563	0.481	1%
	*	Call Hierarchy	Concatenation	0.713	0.640	0.558	22%
			Max-pooling	0.652	0.623	0.547	12%
	**	Version History	Concatenation	**0.739**	0.705	0.588	**27%**
		+ Call Hierarchy	Max-pooling	0.704	0.739	0.554	21%
	**	Version History	Concatenation	0.600	0.568	0.459	3%
		+ No. of Days	Max-pooling	0.678	0.710	0.545	16%
	**	Version History	Concatenation	0.722	0.688	0.608	24%
		+ Call Hierarchy	Max-pooling	0.687	0.663	0.528	18%
		+ No. of Days					
CodeBERT		Without Context		**0.800**	0.753	0.645	
	*	Version History	Concatenation	0.835	0.851	0.693	4%
			Max-pooling	**0.852**	0.777	0.772	**7%**
	*	Call Hierarchy	Concatenation	0.843	0.845	0.700	5%
			Max-pooling	0.835	0.830	0.706	4%
	**	Version History	Concatenation	0.817	0.767	0.674	2%
		+ Call Hierarchy	Max-pooling	**0.896**	0.847	0.812	**12%**
	**	Version History	Concatenation	0.826	0.806	0.706	3%
		+ No. of Days	Max-pooling	0.835	0.787	0.700	4%
	**	Version History	Concatenation	0.870	0.862	0.773	9%
		+ Call Hierarchy	Max-pooling	0.817	0.829	0.638	2%
		+ No. of Days					

*: Single Context | **: Multiple Contexts

In Clone Detection, combining version history, number of days, and method's source code to new code representation achieves the highest F1 score. Namely, ASTNN with concatenation scenario achieves 0.88 in the F1 score and 7% improvement. In addition, CodeBERT, with the concatenation of absolute differences, improves the F1 score by 27% compared to the baseline, equivalent to 0.846. The Code Classification result shows that the combination of version history and call hierarchy context achieves the highest accuracy, increasing by 27% to 0.739 in ASTNN with concatenation scenario. CodeBERT with max-pooling improves accuracy by 12%, from 0.645 to 0.896 in Code Classification problems.

Our technique for encoding the version history to source code representation is in its infancy, and the dataset is modest in terms of size and diversity of code base. This may explain why the experiment results do not show a stable tendency. However, if we select the suitable model and aggregation methods, the technique may improve by up to 27% compared to baseline performance without context. Accordingly, we may obtain the best result if we combine suitable additional artefacts with appropriate techniques.

5 Threats to Validity and Limitations

We now discuss possible threats to the validity of the results and limitations. First, as our final vector presentation is concatenated from all historical versions, the vector may be truncated due to CodeBERT's maximum length of input sequences (512 tokens). In our dataset, the total number of tokens of a method varies from 38 to 369,824 tokens. We observed only 38% (637/1,679) of the methods might be impacted by the truncation issue. We arranged all versions from the most recent to the oldest version and exhaustively concatenated tokens until they reached the model's limit.

Second, our experiment is based on a single dataset. Thus, a statistical test for performance improvement is not applicable. Nevertheless, we quantify the improvement by measuring the percentage difference in F1-score (for Code Clone Detection) and Accuracy (for Code Classification) between adding version history (and other contexts) against the 'without-context' scenario. Lastly, our work confirms that additional context from version history can play a role in improving code representation for code clone detection and code classification. Future work will investigate the potential improvement for other software engineering tasks.

6 Research Agenda

Our preliminary research produces promising outcomes. This section outlines our research agenda to explore different approaches to incorporate version history (and other additional contexts) for better source code representation.

Software Engineering Artefacts. Recent papers claim that additional artefacts are essential to support software developers and deep learning models in comprehending source code [29]. While mining source code repositories to extract version history data, we can collect different types of artefacts and experiment to encode them into source code representation for downstream tasks. These additional contexts may vary in forms, e.g. natural language (commit messages), graphs (call hierarchy), timestamp (commit datetime), or numerical data (number of days, number of versions) [27].

While mining additional contexts, we encounter challenges like inconsistent availability and imbalances in artefacts. For instance, in the SeSaMe dataset, 80% of methods have only one or two versions, yet some have over 200, often with minor differences. This disparity can introduce noise and computational inefficiencies in deep learning, underscoring the need for further analysis on how to best encode version history context to code representation.

Aggregation and Underlying Models. Our preliminary research suggests a need for varied aggregation methods for different contexts in code representation, particularly as our current method of concatenating code versions faces limitations with long-sequence data in ASTNN and CodeBERT. ASTNN is constructed on RNN and GRU, which has a long-term dependency problem [8, 36]. The algorithm allows the model to learn and connect the previous information to the present task; however, when the information is too long (too many versions), it may lose the connection between the present task and the previous nodes. Besides, CodeBERT was pretrained to handle input with a maximum length of 512 tokens [19]; the version history with longer text length will be truncated.

To address the challenge of multiple code versions, potential solutions could be 1) new algorithms that can encode versions selectively or 2) other neural networks, such as Graph Transformer [2], which can handle long-term dependencies and hierarchical information more effectively. We also plan to evaluate the impact of various aggregation techniques, including domain-specific aggregation and general-purpose pooling, for deep learning tasks [5]. With this research agenda, we aim to recommend how to best use source code and additional context.

Data Availability. All the materials produced from this study are available on GitHub [18]

Acknowledgment. Patanamon Thongtanunam was supported by the Australian Research Council's Discovery Early Career Researcher Award (DECRA) funding scheme (DE210101091).

References

[1] Muneera Bano, Rashina Hoda, Didar Zowghi, and Christoph Treude. 2024. Large language models for qualitative research in software engineering: exploring opportunities and challenges. *Automated Software Engineering* 31, 1 (2024), 8.

[2] Vijay Prakash Dwivedi and Xavier Bresson. 2021. A Generalization of Transformer Networks to Graphs. arXiv:2012.09699 [cs.LG]

[3] Damian Okaibedi Eke. 2023. ChatGPT and the rise of generative AI: Threat to academic integrity? *Journal of Responsible Technology* 13 (2023), 100060.

[4] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. arXiv:2002.08155 [cs.CL]

[5] Hossein Gholamalinezhad and Hossein Khosravi. 2020. Pooling Methods in Deep Neural Networks, a Review. arXiv:2009.07485 [cs.CV]

[6] GitClear. 2024. Coding on Copilot: 2023 Data Suggests Downward Pressure on Code Quality (incl 2024 projections) - GitClear — gitclear.com. https://www.gitclear.com/coding_on_copilot_data_shows_ais_downward_pressure_on_code_quality. [Accessed 02-02-2024].

[7] Nicolas Gold, Andrew Mohan, Claire Knight, and Malcolm Munro. 2004. Understanding service-oriented software. *IEEE software* 21, 2 (2004), 71–77.

[8] Alexander Greaves-Tunnell and Zaid Harchaoui. 2019. A Statistical Investigation of Long Memory in Language and Music. In *Proceedings of the 36th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 97)*, Kamalika Chaudhuri and Ruslan Salakhutdinov (Eds.). PMLR, Long Beach, California, USA, 2394–2403. https://proceedings.mlr.press/v97/greaves-tunnell19a.html

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. arXiv:2009.08366 [cs.SE]

[10] Thong Hoang, Hong Jin Kang, David Lo, and Julia Lawall. 2020. CC2Vec: distributed representations of code changes. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering* (Seoul, South Korea) *(ICSE '20)*. Association for Computing Machinery, New York, NY, USA, 518–529. https://doi.org/10.1145/3377811.3380361

[11] Yuan Jiang, Xiaohong Su, Christoph Treude, and Tiantian Wang. 2022. Hierarchical semantic-aware neural code representation. *Journal of Systems and Software* 191 (2022), 111355.

[12] Marius Kamp, Patrick Kreutzer, and Michael Philippsen. 2019. SeSaMe: a data set of semantically similar Java methods. In *Proceedings of the 16th International Conference on Mining Software Repositories (MSR '19)*. IEEE Press, Montreal, Quebec, Canada, 529–533. https://doi.org/10.1109/MSR.2019.00079

[13] Naveen Kulkarni and Vasudeva Varma. 2014. Supporting comprehension of unfamiliar programs by modeling an expert's perception. In *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering* (Hyderabad, India) *(RAISE 2014)*. Association for Computing Machinery, New York, NY, USA, 19–24. https://doi.org/10.1145/2593801.2593805

[14] Yi Li, Shaohua Wang, Tien N Nguyen, and Son Van Nguyen. 2019. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–30.

[15] Ting Long, Yutong Xie, Xianyu Chen, Weinan Zhang, Qinxiang Cao, and Yong Yu. 2022. Multi-View Graph Representation for Programming Language Processing: An Investigation into Algorithm Detection. *Proceedings of the AAAI Conference on Artificial Intelligence* 36, 5 (Jun. 2022), 5792–5799. https://doi.org/10.1609/aaai.v36i5.20522

[16] Jonathan I. Maletic and Andrian Marcus. 2001. Supporting program comprehension using semantic and structural information. In *Proceedings of the 23rd International Conference on Software Engineering* (Toronto, Ontario, Canada) *(ICSE '01)*. IEEE Computer Society, USA, 103–112.

[17] Marjane Namavar, Noor Nashid, and Ali Mesbah. 2022. A controlled experiment of different code representations for learning-based program repair. *Empirical Software Engineering* 27, 7 (2022), 190.

[18] Huy Nguyen, Christoph Treude, and Patanamon Thongtanunam. 2024. Replication Package for "Encoding Version History Context for Better Code Representation". https://github.com/huynxvn/EncodingVersionHistory4CodeRepresentation.

[19] Truong Giang Nguyen, Thanh Le-Cong, Hong Jin Kang, Ratnadira Widyasari, Chengran Yang, Zhipeng Zhao, Bowen Xu, Jiayuan Zhou, Xin Xia, Ahmed E. Hassan, Xuan-Bach D. Le, and David Lo. 2023. Multi-Granularity Detector for Vulnerability Fixes. *IEEE Transactions on Software Engineering* 49, 8 (2023), 4035–4057. https://doi.org/10.1109/TSE.2023.3281275

[20] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. CodeGen: An Open Large Language Model for Code with Multi-Turn Program Synthesis. arXiv:2203.13474 [cs.LG]

[21] Wei Hung Pan, Ming Jie Chok, Jonathan Leong Shan Wong, Yung Xin Shin, Yeong Shian Poon, Zhou Yang, Chun Yong Chong, David Lo, and Mei Kuan Lim. 2024. Assessing AI Detectors in Identifying AI-Generated Code: Implications for Education. In *2024 IEEE/ACM 46th International Conference on Software Engineering: Software Engineering Education and Training (ICSE-SEET)*. IEEE, IEEE / ACM, Lisbon, Portugal, 11–22.

[22] Chanathip Pornprasit, Chakkrit Tantithamthavorn, Patanamon Thongtanunam, and Chunyang Chen. 2023. D-ACT: Towards Diff-Aware Code Transformation for Code Review Under a Time-Wise Evaluation. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE / ACM, Taipa, Macao, 296–307. https://doi.org/10.1109/SANER56733.2023.00036

[23] Hazem Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. 2022. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software* 16, 4 (2022), 351–385.

[24] Bonita Sharif, Timothy Shaffer, Jenna Wise, and Jonathan I Maletic. 2016. Tracking Developers' eyes in the IDE. *IEEE Software* 33, 3 (2016), 105–108.

[25] Richard Sites. 2021. *Understanding Software Dynamics*. Addison Wesley, Boston, USA.

[26] Davide Spadini, Maurício Aniche, and Alberto Bacchelli. 2018. PyDriller: Python framework for mining software repositories. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 908–911. https://doi.org/10.1145/3236024.3264598

[27] Patanamon Thongtanunam, Raula G Kula, Ana EC Cruz, Norihiro Yoshida, Kohei Ichikawa, and Hajimu Iida. 2013. Mining history of gamification towards finding expertise in question and answering communities: experience and practice with Stack Exchange. *The Review of Socionetwork Strategies* 7 (2013), 115–130.

[28] Patanamon Thongtanunam, Weiyi Shang, and Ahmed E Hassan. 2019. Will this clone be short-lived? Towards a better understanding of the characteristics of short-lived clones. *Empirical Software Engineering* 24 (2019), 937–972.

[29] Fuwei Tian and Christoph Treude. 2022. Adding Context to Source Code Representations for Deep Learning. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE / ACM, Limassol, Cyprus, 374–378. https://doi.org/10.1109/ICSME55016.2022.00042

[30] Shaowei Wang and David Lo. 2014. Version history, similar report, and structure: putting them together for improved bug localization. In *Proceedings of the 22nd International Conference on Program Comprehension* (Hyderabad, India) *(ICPC 2014)*. Association for Computing Machinery, New York, NY, USA, 53–63. https://doi.org/10.1145/2597008.2597148

[31] Yuekun Wang, Yuhang Ye, Yueming Wu, Weiwei Zhang, Yinxing Xue, and Yang Liu. 2023. Comparison and Evaluation of Clone Detection Techniques with Different Code Representations. In *Proceedings of the 45th International Conference on Software Engineering (ICSE '23)*. IEEE Press, Melbourne, Victoria, Australia, 332–344. https://doi.org/10.1109/ICSE48619.2023.00039

[32] Kaiyuan Yang, Junfeng Wang, and Zihua Song. 2023. Learning a holistic and comprehensive code representation for code summarization. *Journal of Systems and Software* 203 (2023), 111746. https://doi.org/10.1016/j.jss.2023.111746

[33] Zhengran Zeng, Hanzhuo Tan, Haotian Zhang, Jing Li, Yuqun Zhang, and Lingming Zhang. 2022. An extensive study on pre-trained models for program understanding and generation. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 39–51. https://doi.org/10.1145/3533767.3534390

[34] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Kaixuan Wang, and Xudong Liu. 2019. A novel neural source code representation based on abstract syntax tree. In *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (Eds.). IEEE / ACM, Montreal, QC, Canada, 783–794. https://doi.org/10.1109/ICSE.2019.00086

[35] Kechi Zhang, Zhuo Li, Zhi Jin, and Ge Li. 2023. Implant Global and Local Hierarchy Information to Sequence based Code Representation Models. arXiv:2303.07826 [cs.SE]

[36] Jingyu Zhao, Feiqing Huang, Jia Lv, Yanjie Duan, Zhen Qin, Guodong Li, and Guangjian Tian. 2020. Do RNN and LSTM have Long Memory?. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*, Hal Daumé III and Aarti Singh (Eds.). PMLR, Vienna, Austria, 11365–11375. https://proceedings.mlr.press/v119/zhao20c.html