The Shortcomings of Code LLMs in Modeling Code Properties

Srivishnu Pyda
Department of Computer Science
University of Maryland
College Park, USA
spyda@umd.edu

Daniel Nichols
Department of Computer Science
University of Maryland
College Park, USA
dnicho@umd.edu

Abhinav Bhatele
Department of Computer Science
University of Maryland
College Park, USA
bhatele@cs.umd.edu

***Abstract*—Large language models have rapidly taken over software development tools and are now being used to generate code, write documentation, and even fix GitHub issues. Despite their success, many studies across various fields of computer science have shown that these models often struggle to reason about code properties, such as performance, security, etc. In this paper, we demonstrate the limitations of text-based learning for code properties and how structured code representations are more effective for understanding some code properties. We evaluate over several code benchmarks and demonstrate the limitations of the internal code representation within large language models.**

***Index Terms*—Code Representations, LLMs, Code Properties**

I. Introduction

Software development tools have rapidly adopted large language models (LLMs) for coding tasks making full use of their impressive code generation capabilities. Code LLMs can generate code, write documentation, and even fix GitHub issues. This is all powered by the success of transformer models at understanding and generating text. These models learn to predict new text based on preceding text, and they have been shown to be highly effective at producing code. Due to their success, the majority of code modeling tasks in recent literature have focused on transformer-based models.

While incredibly powerful, code LLMs present many limitations when it comes to modeling code and, in particular, understanding code properties. Code properties are characteristics of code that are not directly related to the code's syntax or semantics, such as its performance, security, or maintainability. While code contains information about these properties, it is a noisy and indirect signal. For example, LLMs can easily be fooled about the performance of a snippet of code by adding particular keywords or comments, such as `fast` or `naive`. Furthermore, code LLMs, which are trained to predict missing text, may not learn the important underlying properties of the code as it is not necessary for the generation of text.

Even though code LLMs have been shown to be effective at generating code, their inability to distinguish between code properties limits their generative capabilities. If LLMs are to be used for tasks that require understanding code properties, i.e. writing efficient code, fixing security vulnerabilities, or improving code readability, they must be able to reason about these properties. That is, if the internal representation within a code LLM cannot capture code properties, then it will not be able to generate code that adheres to these properties. Creating models that can internally understand and model more complex code properties is crucial towards creating better LLMs that can reason about complex code behaviors.

In this paper, we present results and analysis across a series of benchmarks to demonstrate the limitations of the internal representations within current state-of-the-art code LLMs. We compare across several relevant tasks such as performance mapping and algorithm classification. We further motivate the need for more structured code representations to be combined with text-based models to ameliorate these limitations.

We address the following research questions:

RQ1 *How does code as text compare with structured code representations for predicting code properties?*

RQ2 *Can LLM's internal representations of code distinguish between different code properties?*

II. Code Representations

In this section we highlight three code representations that can be used for learning code properties: abstract syntax trees (ASTs), PrograML, and source code as text. While not exhaustive, we select these three as PrograML is a state-of-the-art structured code representation, source code text is now the state-of-the-art for many code generation tasks via transformers, and ASTs are a simple baseline for structured code representations.

A. Abstract Syntax Trees (ASTs)

Abstract Syntax Trees, a tree-based representation of the abstract syntactic structure of source code, are a popular code representation used internally within compilers. Each node in the AST corresponds to a programming construct, such as a variable, expression, or control structure. ASTs abstract away the source code's syntactic details such as semicolons and parentheses but still maintain its logical structure. ASTs are a core dataset within compilers for representing source code. After lexing the code into tokens a compiler will construct an AST to use for further tasks like static analysis, optimization, and code generation. However, another use case for this code representation is statistical modeling. By utilizing their inherent tree structure, ASTs can be transformed into graph representations, with node types encoded using one-hot encoding to serve as inputs for graph neural networks (GNN). These GNNs can be used for downstream tasks like code classification and vulnerability detection. Several variations of ASTs have been used for code modeling [1], [2] and shown success at predicting code properties using only transformations of ASTs. Despite their success ASTs still have several shortcomings. Natively, they do not incorporate direct information about control and data flow and do not represent semantic information about individual instructions. Furthermore, they are language dependent, so a model trained on ASTs of C/C++ code will not be useful for Fortran or Python code.

B. Graph Representations of Compiler Intermediate Representations

To solve many of the shortcomings of AST representations, recent papers have looked into structured graph representations of compiler intermediate representations (IR). These representations have many desirable properties such as being language agnostic and insensitive to semantic-preserving code transformations. A seminal graph IR code representation is Program Graphs for Machine Learning (PrograML) [3], a graph-based, compiler-agnostic approach built upon LLVM IR. IR, such as LLVM IR and XLA IR, encodes low-level program behavior in a language-agnostic format. PrograML includes control, data, and call relations by serving as the union between a call graph, control-flow graph, and data-flow graph. A node represents an instruction, variable, or constant while an edge represents a control flow relation, data flow relation, or call relation. By integrating multiple dimensions of a program into a single representation, PrograML captures more intra-program relations compared to other representations such as ASTs. This unified and expressive graph representation provides an expressive, but flexible format for using code representations for code-related downstream tasks such as learning about program semantics. There are several other similar graph-based program representations that make use of IR and control/data flow graphs to represent code [4]–[6], which all exhibit comparable performance across tasks. We select PrograML as it is a seminal work in this area and still state-of-the-art. However, we include results from these papers, namely inst2vec [6] and perfograph [5], in our plots for reference.

C. Source Code and Transformers

Since the introduction of the transformer architecture [7], source code has become a predominant representation for code-related modeling tasks, in particular for code generation. This last code representation treats text as a sequence of tokens and uses LLMs to predict new tokens based on their internal representation of the code. To process an input sequence, transformers leverage a mechanism called attention, which enables them to dynamically focus on the most relevant parts of the inputs. This attention mechanism allows transformers to effectively model long-range dependencies, making them well-suited for code-related tasks. LLMs are primarily designed for causal language modeling, where the goal is to predict the next token in a sequence given its context. This capability has been adapted for code-focused LLMs such as CodeBERT [8], DeepSeek-Coder [9], and Starcoder [10]. They have been shown to be remarkably successful at various code generation tasks [11]. Internally, each layer of the transformer applies matrix transformations and attention mechanisms to a vector representation of the code. At the final layer, the output is a vector representation that encodes the model's understanding of the source code. This internal representation is passed through a language modeling head, which maps the high-dimensional representation back into the token space in order to predict the next token in the sequence, which enables tasks like code completion and generation. We can use the input into the language modeling head as a vector representation of the model's internal code understanding.

III. Benchmarks

We use three different benchmark tasks to evaluate various aspects of each code representation. In this section, we detail the two device mapping benchmarks and the algorithm classification benchmark that we use to evaluate each code representation.

A. Criteria for Benchmark Selection

We specifically opt for non-generation benchmarks because our experiments involve GNNs, which are not typically used for generating text, as they aren't inherently designed for sequential data generation. More importantly, our focus is to model more than just generative capabilities. Another consideration is that these benchmarks need to utilize LLVM IR as their native backend. For example, a C compiler like Clang can translate C code into IR. Taking these requirements into account, the two factors we want to model are performance of code and algorithm classification. To benchmark these effectively, we refer to commonly used datasets from the literature: Heterogeneous Device Mapping for performance and POJ-104 for algorithm classification.

B. Heterogeneous Device Mapping

The first downstream task is heterogeneous device mapping (devmap). We use the same benchmark as the one presented in [3]. This benchmark contains information about 256 OpenCL kernels that were collected from seven different sources, including Parboil, Rodinia, SHOC, NPB, NVIDIA SDK, and AMD SDK. Each of these OpenCL kernels has their corresponding runtimes from being run on Intel Core i7-3820 CPU, NVIDIA GTX 970 GPU, and AMD Tahiti 7970 GPU. Devmap is split into two sets: AMD (devmap-amd) and NVIDIA (devmap-nvidia). The NVIDIA set contains the results of running each kernel on the Intel CPU and NVIDIA GPU. The AMD set contains the results of running each kernel on the Intel CPU and AMD GPU. The benchmark also contains 680 LLVM IR instances that were collected from these kernels. We select this task because it is widely used as a benchmark in prior studies, allowing us to effectively compare the performance of LLMs with existing code representations. Furthermore, it allows us to evaluate how well code representations can distinguish performance characteristics of code.

C. Algorithm Classification

As the second benchmark, we test each code representation on the task of algorithm classification. We utilize the POJ-104 benchmark [1], which contains 104 algorithm classes and approximately 500 C++ code samples per algorithm. Similar to the previous task, we choose this task because of its popularity among previous approaches in the literature. Additionally, this task allows us to evaluate the code representations against their ability to discriminate code semantic and algorithm properties. This is a separate problem from the device placement benchmark and requires a different set of code properties to be captured. Furthermore, it allows us to test for invariance to code symmetry, i.e. is the representation sensitive to semantic preserving code transformations. This is an important property for a useful code representation to possess.

IV. Experimental Setup

Using the device mapping and algorithm classification benchmarks, we evaluate the three code representations across two tasks: classification and embedding. In this section we highlight how we train each model for classification and obtain embeddings for each data point.

A. Experimental Justification

Inst2Vec [6] and perfograph [5] are state-of-the-art structured code representation models that have demonstrated strong performance in tasks relating to understanding code properties. Inst2vec defines an embedding space for individual IR statements while perfograph builds upon PrograML by adding numerical awareness. While these models are relevant to our study, we encountered challenges in reproducing their results and were unable to fully integrate them into our experimental pipeline. Despite these challenges, we still reported their results in our evaluation to to contextualize the performance of our models relative to existing state-of-the-art approaches.

B. Training the Models for Classification

When testing PrograML, we utilize different model architectures for different benchmarks. For the algorithm classification benchmark, we use a Relational Graph Convolutional Network (R-GCN) designed for heterogeneous graphs with two graph convolutional layers. This heterogeneous architecture is necessary as PrograML defines multiple edge types. We use a hidden layer size of 64 and a final linear layer of size 64, and we train this model for 100 epochs. With the device mapping benchmark, we opt for an R-GCN with one graph convolutional layer. We use a hidden layer size of 32 and an embedding size of 64, and we train this model for 100 epochs. While this architecture deviates from the original one used in the PrograML paper [3], we found it to give comparable or even better results.

When testing ASTs, we utilize the same base model architecture but change the hyperparameters across different benchmarks. We use a Graph Convolutional Network (GCN) with two convolutional layers. For the device mapping benchmark, we use a hidden size of 32 and an embedding size of 36, which corresponds to the number of unique node types. We train the model for 50 epochs for the NVIDIA set and 60 epochs for the AMD set. For the algorithm classification benchmark, we use a hidden size of 64 and an embedding size of 256. We train the model for 200 epochs.

Across both benchmarks for PrograML and AST, the parameters that we tune are learning rate, hidden layer dimension, and embedding size. We perform a grid search between these parameters to see which combination yields the best results. First, for the learning rate, we experiment with learning rates ranging from 0.1 to 0.0001, initially decreasing by an order of magnitude 0.1 at each step. Once we identify a promising range through these larger adjustments, we perform finer tuning by testing intermediate values, such as 0.0005, to further optimize performance. For both the hidden layer and embedding size, we experiment with values between 16 through 128, increasing the size by a factor of 2 at each step. Additionally, all models are trained or fine-tuned on the same amount of data.

When testing source code, we utilize DeepSeek Coder [9] and Starcoder [10], [12], state-of-the-art code LLMs, across all benchmarks. We use the following models: TinyStarCoderPy (164M parameters), StarCoderBase-1B, StarCoder-3B, StarCoder2-7B, and DeepSeek-Coder-6.7B. We fine-tune these language models for text classification. This is done by replacing the language modeling head with a classification head before doing fine-tuning. We initialize the rest of the model weights using the pre-existing models. We utilize FP16 precision and an H100 GPU to accelerate fine-tuning. We train the models for 10 epochs for the algorithm classification and device mapping benchmarks.

For all the experiments above, we utilize an 80-10-10 train-test-validation split during training. Furthermore, cross-entropy loss and the AdamW optimizer [13] are used across all training runs. The number of epochs was chosen by monitoring the loss and validation curves to stop training when the model performance plateaued.

C. Retrieving Program Embeddings

In addition to evaluating the classification performance we want to retrieve vector embeddings from each program representation to further analyze. For both the ASTs and PrograML representations this is fairly straightforward given the GCN architecture used. We train over a number of AST and PrograML IR graphs, respectively, and remove the final classification layer from the GNN. Now we can use the raw output logits from the GNN as the learned embeddings for each code. Through experimentation, we found an ideal embedding size of 256 for ASTs and 64 for ProgaML IR graphs.

For LLMs, embeddings are created for code samples using a state-of-the-art embedding language model. Similar to the graph representations we are able to retrieve the internal representation the LLM uses to generate the code. We use the final hidden state of the model as the embedding for each code sample. For all three representations, we also reduce the dimensionality of the embeddings to 2 dimensions using t-SNE [14] for visualization purposes.

V. Evaluation Metrics

In this section we present the metrics we use to evaluate each representation across the benchmarks and tasks.

A. Classification Metrics

We first look at the classification accuracy of each trained model on the devmap and algorithm classification benchmarks. The accuracy is computed over the test dataset, which is a 10% split of the original dataset. The accuracy is reported as the percentage of correctly classified samples over the total number of samples in the test dataset. For the devmap classification task, this is the percentage of times that the model is correctly able to predict if an OpenCL kernel would be faster on the CPU or GPU. For the algorithm classification task, this is simply choosing the correct algorithm out of the 104 possible algorithms in the benchmark.

B. Embedding Metrics

For the embeddings we use the *silhouette score* to evaluate the quality of embeddings. The silhouette score is a measure of how similar the vectors within a label class are to each other, compared to the vectors in other classes. For a given sample $\boldsymbol{x}$, the silhouette score is computed as:

$$s(\boldsymbol{x}) = \frac{b(\boldsymbol{x}) - a(\boldsymbol{x})}{\max(a(\boldsymbol{x}), b(\boldsymbol{x}))}$$

where $a(\boldsymbol{x})$ is the average distance between $\boldsymbol{x}$ and all other samples in the same class, and $b(\boldsymbol{x})$ is the average distance between $\boldsymbol{x}$ and all samples in the nearest class that is not the same as $\boldsymbol{x}$'s class. For classes we use the existing labels, *"faster on CPU or GPU"* and *"algorithm"*, from the devmap and algorithm classification tasks, respectively. The final silhouette score of a representation is reported as the average of the silhouette scores of all samples in the benchmark. This value ranges between -1 and 1 with higher values indicating better embeddings. This metric is used to evaluate the quality of the learned embeddings by how well their internal representations can be separated along code properties. Furthermore, for the algorithm classification samples, this will show how invariant the internal representations are to code symmetry.

We finally use visual embeddings to analyze the quality and meaningfulness of the learned representations. While embedding vectors themselves do not have inherent accuracy, their effectiveness lies in how well they capture and preserve semantic information. Visual presentations of the embeddings can shed light into the quality of the learned representations. This is exemplified when the data points belonging to the same class to be close together in the embedding space. We use t-SNE [14] to visualize two dimensional embeddings of the samples with their color corresponding to their label. For the algorithm classification task, we display a random subset of five algorithms to reduce visual clutter.

VI. Results

In this section we present our results across the classification tasks and the embeddings. We further discuss the implications of the LLMs performance on these tasks.

A. Classification

> **RQ1** *How does code as text compare with structured code representations for predicting code properties?*

Figure 1 shows the classification accuracy for the different representations across the benchmarks. We also include results from inst2vec and perfograph on these tasks for reference. We observe that the structured representations, namely ProgaML, inst2vec, and perfograph, have better performance than the language models on all the prediction tasks. These representations are better able to train a model to predict code properties. The difference is largest for the performance prediction task, where LLMs perform significantly worse.

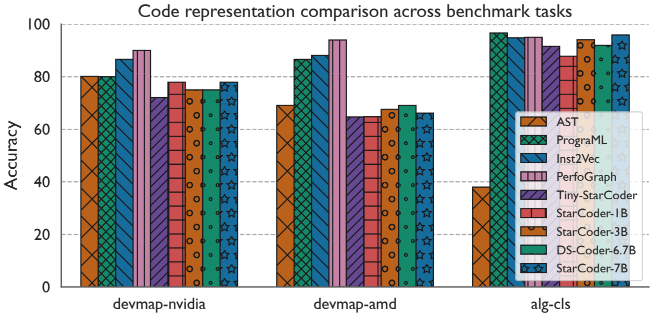

Fig. 1. Classification accuracy for the different code representations across the three benchmarks. The structured representations outperform the LLMs on all tasks with the largest advantage on the performance prediction task.

The picture is even bleaker for the LLMs when we consider the number of parameters required to achieve these results. Figure 2 shows the parameter efficiency of each representation as accuracy per parameter. While not a readily interpretable metric, it gives us insight into the scale of percentage points accuracy improvement per parameter. We see that the structured representations are significantly more parameter efficient than the LLMs. This is a huge upside to these models as they can be used to model code properties and run in inference with very low overhead.

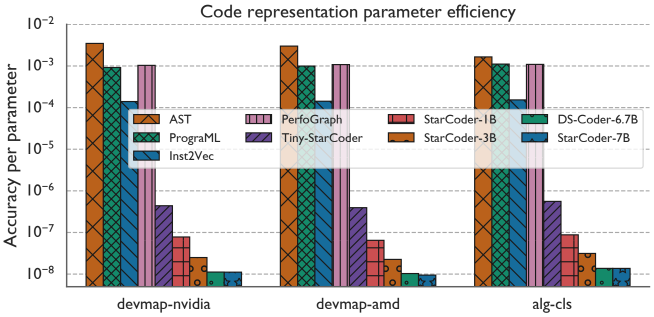

Fig. 2. Classification accuracy for the different code representations across the three benchmarks. The structured representations are much more parameter efficient than the LLMs. They significantly outperform the LLMs while having several orders of magnitude less parameters.

B. Embeddings

> **RQ2** *Can LLM's internal representations of code distinguish between different code properties?*

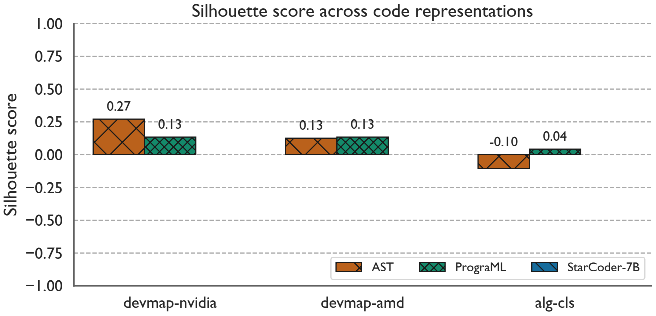

Fig. 3. Silhouette scores for the different code representations. Higher is better. The structured representations, namely PrograML, are better at encoding program properties within their internal representations.

Figure 3 shows the silhouette scores for the different code representations. For the OpenCL kernels, the AST and PrograML representations provide embeddings that are better separated than the LLMs. This follows along with the trends in the classification tasks. The LLM embeddings are near zero, indicating that many of the embeddings for different labels overlap. This further indicates that the LLM embeddings do not encode information about the code performance well, while the structured representations do. The results for algorithm classification are all similar. ASTs perform quite poorly here, while PrograML and LLMs achieve similar near-zero silhouette scores.

Finally, we can visualize the embeddings in Figures 4 and 5. The first, Figure 4, shows the embeddings for the device mapping benchmark. We first notice that all of the representations produce smaller clusters within each label class. However, the LLM has these smaller clusters intermingled and less separated, which PrograML has better separation between the classes. This trend is similar but less extreme with the AST embeddings. The second, Figure 5, shows the embeddings for the algorithm classification benchmark. Only a subset of five random algorithms are displayed for clarity. Each of the representations produces well-defined clusters with the AST embeddings having slightly more overlap between the classes. This lines up with the classification results, where the structured representations are only slightly better than the LLMs.

VII. Threats To Validity

A key threat to validity arises from the inconsistent experimental setup. The LLMs are fine-tuned specifically for the downstream tasks, but the GNNs are trained from scratch. This discrepancy in how the models are trained introduces an unfair comparison. To address this bias and ensure a more rigorous experimental design, one strategy is to pre-train GNNs using a masked training objective similar to the masked language modeling (MLM) used for LLMs. In this setup, the GNN would be pre-trained on LLVM IR data by masking parts of the intermediate representation and training the model to predict the masked components. Alternatively, the LLMs could be trained from scratch on the specific code property tasks, just like the GNNs. This approach means initializing the LLMs with random weights.

VIII. Conclusion

In this paper, we have evaluated three different program representations on their efficacy for modeling code properties. We found that structured representations tend to outperform language models on tasks like performance modeling and algorithm classification. Furthermore, they are able to do so with significantly fewer parameters. Finally, we demonstrated that the internal representations of LLMs are incapable of distinguishing between different code properties, while structured representations are able to do so. This suggests that code as text might not be enough to model code properties effectively.

Furthermore, if LLMs' internal representations are incapable of distinguishing code properties, then they will be unable to effectively generate code with those properties. This is a significant limitation of LLMs and will continue to limit their capabilities for more complex coding tasks. With this work, we aim to motivate future research into developing code representations that are better able to model code properties and generate code with those properties.

Acknowledgment

This research used supercomputing resources at the University of Maryland.

References

[1] L. Mou, G. Li, L. Zhang, T. Wang, and Z. Jin, "Convolutional neural networks over tree structures for programming language processing," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, ser. AAAI'16. AAAI Press, 2016, p. 1287–1293.

[2] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," in *International Conference on Learning Representations*, 2018. [Online]. Available: https://openreview.net/forum?id=BJOFETxR-

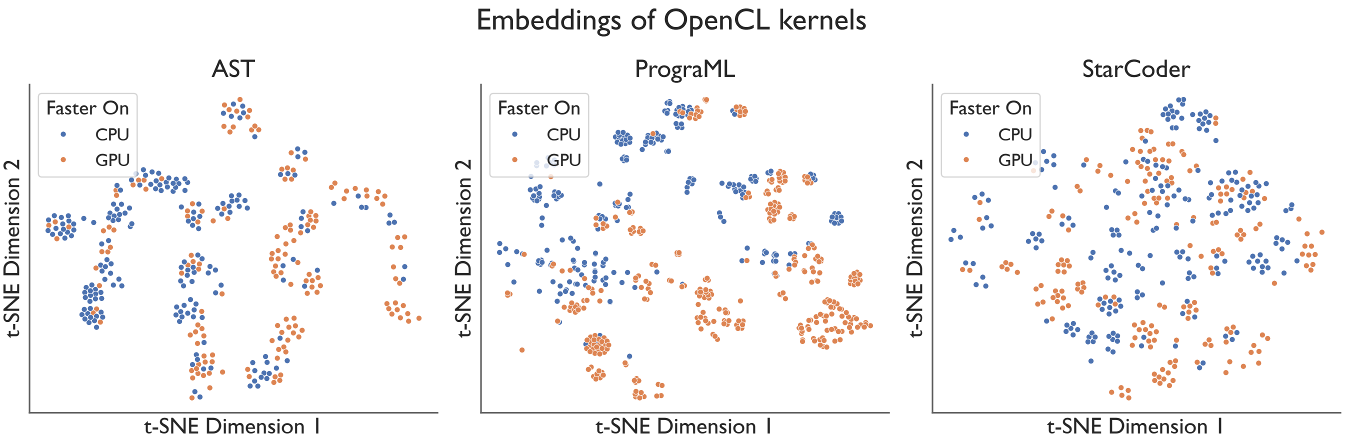

Fig. 4. Embeddings from the different program representations of OpenCL kernels. The embeddings are projected to 2D using t-SNE. The points are colored by whether the kernel is faster on a CPU or a GPU. We see that the more structured representations provide better separation between the two classes.

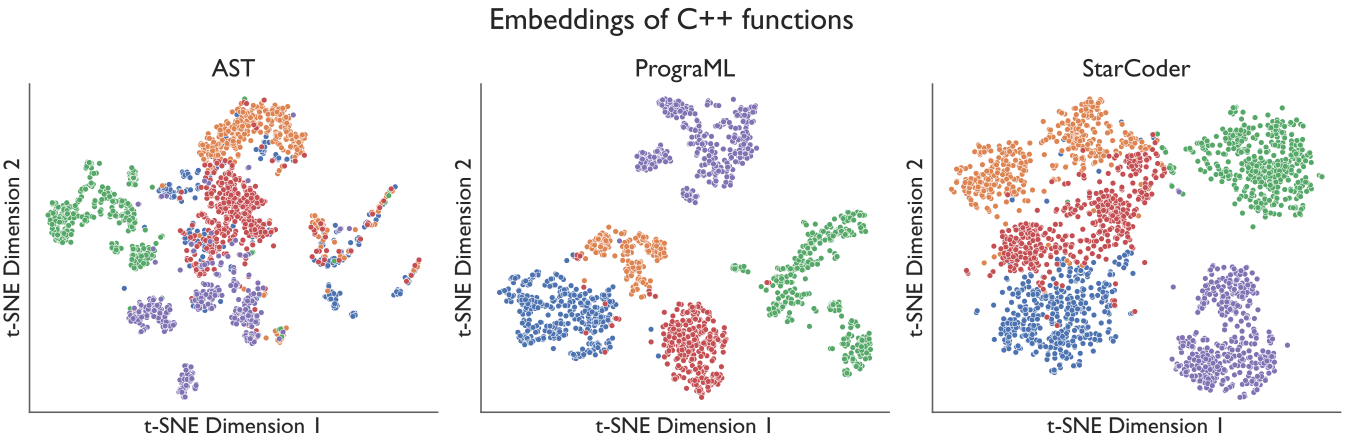

Fig. 5. Embeddings from the different program representations of POJ-104 algorithms. The embeddings are projected to 2D using t-SNE. The points are colored by the algorithm class.

[3] C. Cummins, Z. V. Fisches, T. Ben-Nun, T. Hoefler, M. F. P. O'Boyle, and H. Leather, "Programl: A graph-based program representation for data flow analysis and compiler optimizations," in *Proceedings of the 38th International Conference on Machine Learning*, ser. Proceedings of Machine Learning Research, M. Meila and T. Zhang, Eds., vol. 139. PMLR, 18–24 Jul 2021, pp. 2244–2253. [Online]. Available: https://proceedings.mlr.press/v139/cummins21a.html

[4] B. Steenhoek, H. Gao, and W. Le, "Dataflow analysis-inspired deep learning for efficient vulnerability detection," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ser. ICSE '24. New York, NY, USA: Association for Computing Machinery, 2024. [Online]. Available: https://doi.org/10.1145/3597503.3623345

[5] A. T. Jamsaz, Q. I. Mahmud, L. Chen, N. K. Ahmed, and A. Jannesari, "Perfograph: a numerical aware program graph representation for performance optimization and program analysis," in *Proceedings of the 37th International Conference on Neural Information Processing Systems*, ser. NIPS '23. Red Hook, NY, USA: Curran Associates Inc., 2024.

[6] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems 31*, S. Bengio, H. Wallach, H. Larochelle, K. Grauman, N. Cesa-Bianchi, and R. Garnett, Eds. Curran Associates, Inc., 2018, pp. 3588–3600. [Online]. Available: http://papers.nips.cc/paper/7617-neural-code-comprehension-a-learnable-representation-of-code-semantics.pdf

[7] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," *CoRR*, vol. abs/1706.03762, 2017. [Online]. Available: http://arxiv.org/abs/1706.03762

[8] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," in *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Minneapolis, Minnesota: Association for Computational Linguistics, Jun. 2019, pp. 4171–4186. [Online]. Available: https://www.aclweb.org/anthology/N19-1423

[9] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. K. Li, F. Luo, Y. Xiong, and W. Liang, "Deepseek-coder: When the large language model meets programming – the rise of code intelligence," 2024.

[10] R. Li, L. B. Allal, Y. Zi, N. Muennighoff, D. Kocetkov, C. Mou, M. Marone, C. Akiki, J. Li, J. Chim, Q. Liu, E. Zheltonozhskii, T. Y. Zhuo, T. Wang, O. Dehaene, M. Davaadorj, J. Lamy-Poirier, J. Monteiro, O. Shliazhko, N. Gontier, N. Meade, A. Zebaze, M.-H. Yee, L. K. Umapathi, J. Zhu, B. Lipkin, M. Oblokulov, Z. Wang, R. Murthy, J. Stillerman, S. S. Patel, D. Abulkhanov, M. Zocca, M. Dey, Z. Zhang, N. Fahmy, U. Bhattacharyya, W. Yu, S. Singh, S. Luccioni, P. Villegas, M. Kunakov, F. Zhdanov, M. Romero, T. Lee, N. Timor, J. Ding, C. Schlesinger, H. Schoelkopf, J. Ebert, T. Dao, M. Mishra, A. Gu,

J. Robinson, C. J. Anderson, B. Dolan-Gavitt, D. Contractor, S. Reddy, D. Fried, D. Bahdanau, Y. Jernite, C. M. Ferrandis, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder: may the source be with you!" 2023.

[11] M. Chen and et al, "Evaluating large language models trained on code," 2021.

[12] A. Lozhkov, R. Li, L. B. Allal, F. Cassano, J. Lamy-Poirier, N. Tazi, A. Tang, D. Pykhtar, J. Liu, Y. Wei, T. Liu, M. Tian, D. Kocetkov, A. Zucker, Y. Belkada, Z. Wang, Q. Liu, D. Abulkhanov, I. Paul, Z. Li, W.-D. Li, M. Risdal, J. Li, J. Zhu, T. Y. Zhuo, E. Zheltonozhskii, N. O. O. Dade, W. Yu, L. Krauß, N. Jain, Y. Su, X. He, M. Dey, E. Abati, Y. Chai, N. Muennighoff, X. Tang, M. Oblokulov, C. Akiki, M. Marone, C. Mou, M. Mishra, A. Gu, B. Hui, T. Dao, A. Zebaze, O. Dehaene, N. Patry, C. Xu, J. McAuley, H. Hu, T. Scholak, S. Paquet, J. Robinson, C. J. Anderson, N. Chapados, M. Patwary, N. Tajbakhsh, Y. Jernite, C. M. Ferrandis, L. Zhang, S. Hughes, T. Wolf, A. Guha, L. von Werra, and H. de Vries, "Starcoder 2 and the stack v2: The next generation," 2024.

[13] I. Loshchilov and F. Hutter, "Fixing weight decay regularization in adam," *CoRR*, vol. abs/1711.05101, 2017. [Online]. Available: http://arxiv.org/abs/1711.05101

[14] L. van der Maaten and G. Hinton, "Visualizing data using t-sne," *Journal of Machine Learning Research*, vol. 9, no. 86, pp. 2579–2605, 2008. [Online]. Available: http://jmlr.org/papers/v9/vandermaaten08a.html