Better Modeling the Programming World with Code Concept Graphs-augmented Multi-modal Learning

Martin Weyssow
DIRO, Université de Montréal
Montreal, Canada
martin.weyssow@umontreal.ca

Houari Sahraoui
DIRO, Université de Montréal
Montreal, Canada
sahraouh@iro.umontreal.ca

Bang Liu
DIRO & Mila, Université de Montréal
Montreal, Canada
bang.liu@umontreal.ca

ABSTRACT

The progress made in code modeling has been tremendous in recent years thanks to the design of natural language processing learning approaches based on state-of-the-art model architectures. Nevertheless, we believe that the current state-of-the-art does not focus enough on the full potential that data may bring to a learning process in software engineering. Our vision articulates on the idea of leveraging multi-modal learning approaches to modeling the programming world. In this paper, we investigate one of the underlying idea of our vision whose objective based on concept graphs of identifiers aim at leveraging high-level relationships between domain concepts manipulated through particular language constructs. In particular, we propose to enhance an existing pretrained language model of code by joint-learning it with a graph neural network based on our concept graphs. We conducted a preliminary evaluation that shows gain of effectiveness of the models for code search using a simple joint-learning method and prompts us to further investigate our research vision.

1 INTRODUCTION

One of the main objective of language models and graph neural networks (GNN) in natural language processing (NLP) is to implicitly encode knowledge about the world by learning semantically meaningful representations of words and concepts. In software engineering, language models have been extensively used to modeling the programming world [3, 13, 19] and reason about it through a variety of tasks [1, 7]. Alternatively, graph neural networks have been used to learn from structured information such as abstract syntax trees and have shown great success to modeling source code [2, 9, 18]. Most of the aforementioned related work focus on learning either from a unique abstraction of a data source or from
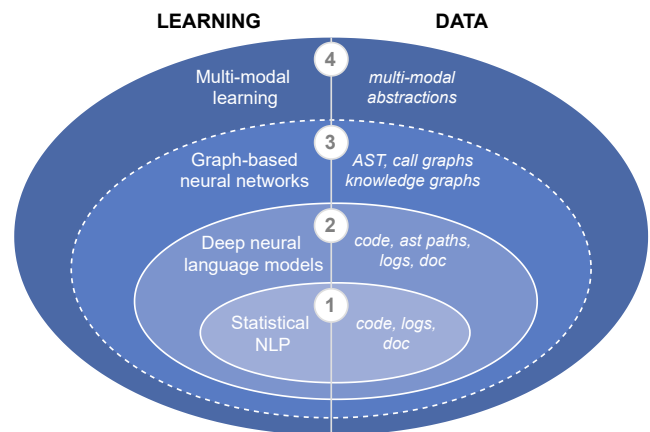
Figure 1: NLP-based learning approaches in software engineering research with their corresponding usual data sources. The outermost layer relates to our vision about combining multiple data modalities and learning models.

multiple data sources, *e.g.,* a code and its corresponding documentation. We claim that neither of these approaches are sufficient to fully modeling the programming world and that we need to design multi-modal approaches tailored to learn from different levels of abstractions of the same software artifact, *e.g.,* a source code and its AST. Furthermore, we conjecture that high-level data abstractions such as concept graphs (CG) could guide learning models towards a better understanding of software.

In fact, as we human communicate by the mean of natural languages, we manipulate words and concepts at different level of abstraction. That is, our understanding of the language relies on a principle of compositionality where high-level concepts are defined as compositions of lower level concepts, *e.g.,* a tree is made of leaves and a trunk. Furthermore, at the grammar-level of the language, a sentence can be decomposed as a dependency graph between words or group of words making the language very hierarchical and structured [23, 29, 37]. In a similar way, this notion of compositionality and hierarchies is central in programming languages. Even though we write code as sequences of textual elements, these sequences have an equivalent deterministic abstraction in form of an abstract syntax tree that encodes structural, semantic and syntactic information of the code. In other terms, AST-like abstractions of the code have the benefit of incorporating higher level information such as the logical structure of code. In addition and as part of this work, analogously to natural languages we conjecture that programs are concept-based. That is to say, a program can be
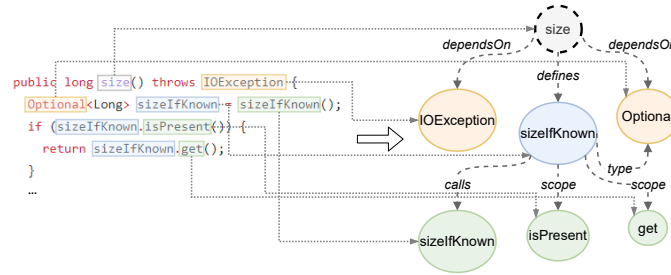
Figure 2: Excerpt of the extraction of a concept graph from a code snippet in Java.

decomposed in term of high-level concerns, *e.g.,* a method or a class, that manipulate low-level concepts which target a more specific and local purpose such as a variable. In addition, programming concepts are not always defined as such and can designate a direct manipulation of domain concepts, *e.g.,* specified in the requirements of the software. In other words, we underline the fact that concepts are deeply rooted in source code artifacts and manipulated using particular language-specific constructs each of them associated with an identifier bearing the meaning of the concept.

In order to support our first claim, lets go through a discussion about the first three layers depicted in Fig. 1 which summarizes NLP-based learning approaches widely used to learn from text with concrete examples of data sources specific to software engineering. The first two layers relate to token-based language modeling, *i.e.,* learning from sequences of texts. On the one hand, statistical NLP approaches are based upon learning recurrent patterns from software artifacts such as code, commit logs or natural language documentation [3, 17, 19]. On the other hand, recent deep-learning-based approaches leveraging powerful language model architectures [8, 12, 20, 32, 38] to learn representations of source code have shown impressive results in tasks such as code completion or code search [13, 24, 28, 33, 36]. Within these two kind of approaches, the learning model relies heavily on linguistics features specific to the input language and the data are required to be textual, albeit some works designed approaches to integrate structural information such as AST paths or data flow in the input data [4, 5, 16]. Next, at a higher level, we identified graph-based learning approaches which usually leverage language-specific code abstractions such as ASTs or call graphs. Similarly to natural language processing, here GNNs [14, 34] can be used to focus on learning structural information embedded in the input graph data. In addition, graph-based learning approaches benefit from graph representations and their ability to encode higher abstraction and logical structures in a more robust way. Therefore, that makes this latter approach complementary to the two previous one and combining them could result in learning even more potent source code representations by learning from different level of abstractions.

To summarize, we make the following observations:

— *The data that can be extracted from a same software artifact can be highly heterogeneous and lie at different levels of abstraction (c.f., Fig. 1), each of them with the potential to offer various structural, semantic and syntactic information about the artifact.*

— *Few amount of previous works propose to combine learning models in a multi-modal fashion by considering several data modalities of software artifacts.*

As a matter of fact, to our knowledge multi-modal related work mostly involve bi-modal learning using paired data samples of *<code tokens, code documentation>* in the context of code search or documentation generation [13, 16, 21] or multi-modal learning with textual data only [15] and thus without leveraging graph-based modalities that embed more structural information.

These observations lead us to the two following challenges for the future. The first one related to the data and the second one to the learning:

> Challenge 1: **what are meaningful data abstractions to be defined and to extract from source code artifacts in order to enable learning programming knowledge from various data modalities?**
>
> Challenge 2: **how to efficiently modeling source code artifacts using multi-modal learning approaches?**

As part of this work and as we believe the notion of concept to be central in software programming, we make a first attempt to tackle both these challenges by (1) learning representations of concept graphs extracted from code snippets and (2) proposing a learning framework to jointly learn graph representations of the concept graphs and their corresponding code representations. The objective of our approach is to improve the effectiveness of an existing pretrained language model of code in the context of a code search task. In Sec. 2 we provide details about our implementation then present preliminary results in Sec. 3. Finally, we discuss future work direction in Sec. 4 and draw a conclusion.

2 CODE CONCEPT GRAPHS-AUGMENTED MULTI-MODAL LEARNING

Driven by our claims and motivations, we propose to extract concept graphs from code snippets in view of enhancing an existing pretrained LM of code. By joint-learning a LM based on code tokens and a GNN based on concept graphs, we expect both models to learn better vector representations that incorporate more semantics by covering different level of abstractions thanks to the diversity of the data modalities. Furthermore, as part of our framework, we propose two methods for joint-learning both models. We depict the overall architecture of our framework in Fig. 3 and illustrate its applicability to code search.

2.1 Code concept graphs

We envision our code concept graphs to represent high-level relationships and dependencies between domain concepts materialized by particular code constructs, *e.g.,* variables, calls, methods. That is, our proposed CG includes only code identifiers and does not directly depend on the syntax of the input language. This is different from other abstractions that have been used previously to improve code comprehension. Among others, some works proposed to create knowledge graphs based on code and documentation [11, 35] or code ontologies incorporating many linguistic features of the input programming language [6, 22]. On the one hand, the aforementioned works do not intend to integrate their knowledge-based data as part of a deep NLP learning process such as ours. And, conversely to our proposed code CG, the data abstractions in these works are mostly language-specific and do not aim at extracting relational CG of identifiers.

Concept graph design. A code CG is a relational graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ where $\mathcal{V}$ denotes the set of identifiers nodes and $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{R} \times \mathcal{V}$ the set of edges with a set of typed relations $\mathcal{R}$. Given a code snippet, we retrieve all the identifiers (id_{name}, id_{type}) where $type \in \{method\ name, parameters, imports, variables, calls\}$ and map it to a unique node $\mathbf{z} \in \mathcal{V}$ with id_{name} and id_{type} its features. Then, we retrieve in total 9 types of relations $\in \mathcal{R}$, each of them describing a semantic dependency between two identifiers that results from their utilization in the code. We showcase an example for Java in Fig. 2 that includes 4 types of relations. For example, "*dependsOn*" denotes a relation between the method and an imported identifier that is manipulated in the method. Another example is "*defines*" that describes a variable being defined in the body of the method. Or else, "*calls*" denotes that a variable depends on the result of a call, *e.g.,* during an assignment. We end-up with a CG of identifiers and their semantic relationships which is also much less dependent on linguistic features of the language than code tokens.

Task and data extraction. Code search is a task that consists of retrieving the most relevant code snippets from a corpus given a natural language query which reflects the user's information need. Therefore, the starting point of our approach requires a dataset of pairs of <*code, doc*>, *i.e.,* a syntactically valid code method and its corresponding natural language description. Even though the natural language description is not a user query per se, it is commonly used as a proxy query in the literature to train or fine-tune a model without the need of any user relevance's annotations [13, 15, 21]. From the code snippet, we extract its corresponding CG resulting in triplets of <*CG, code, query*>. We choose to explore our approach on Java but our designed CG could be extracted from other programming languages. Additionally, it could be generalized to other tasks with few adjustments.

2.2 Joint-learning and models combinations

Multi-modal or multi-view deep learning refers to learning systems that learn representations from multiple modalities [27, 31]. In this section, we present two approaches for joint-learning two learning models. We refer to [27] for in-depth details on multi-modal representation learning.

Combining models outputs. Lets assume two data modalities whose vector representations say $\mathbf{h}_x$ and $\mathbf{h}_y$ can be obtained by two learning models. The goal of this approach is to obtain a final representation $\mathbf{H} = \phi(\mathbf{h}_x, \mathbf{h}_y)$. The function ϕ denotes the combination of both vectors which can be defined as a sum $\phi(\mathbf{h}_x, \mathbf{h}_y) = \mathbf{h}_x + \mathbf{h}_y$, a concatenation $\phi(\mathbf{h}_x, \mathbf{h}_y) = [\mathbf{h}_x, \mathbf{h}_y]$, or a more sophisticated fusion method. Coming back to our data, one can use a LM and GNN with pooling layers to encode respectively the code tokens and the CG into single-vector representations, *i.e.,* code and graph embeddings. The embeddings are then combined using a function ϕ to obtain the final representation that embeds knowledge from both modalities.

Combining models. This second more sophisticated method consists of combining modules or sub-parts of the learning models such that their architectures are no longer independent. For instance, existing approaches such as [25] allow to condition the word representations learned by a LM on external modalities such as images. In the context of our work, a LM and a GNN can be combined in several ways. Following [25], the representation of each identifier code token in the LM could be conditioned on the representation of its corresponding node in the CG obtained by the GNN. Or else, the embedding layer of the LM or the GNN could be initialized with a pretrained embedding layer obtained from the other modality. Conversely to the former one, this method allows more fine-grained combinations of both learning models by integrating the multi-modal learning perspective as part of their architecture.

For sake of clarity lets call $\mathbf{h}_q$ the representation of the query, $\mathbf{h}_{code}$ the representation of the code and $\mathbf{h}_{CG}$ the representation of the concept graph. As part of this work, we get the output representation $\mathbf{h}_c$ by combining the models outputs with $\phi = \mathbf{h}_{code} + \mathbf{h}_{CG}$. As the target end-task is code search, the learning objective is to learn representations $\mathbf{h}_q$ and $\mathbf{h}_c$ that are semantically close by minimizing their distance. Therefore, we use cosine similarity to compare $\mathbf{h}_c$ with $\mathbf{h}_q$ and maximize the similarity during learning.

3 PRELIMINARY EVALUATION

To get insights on whether our work direction on multi-modal learning with concept graphs is worth investigating, we conducted a preliminary evaluation of our approach with code search as end-task in Java.

Data and models. We use a modified version of CodeSearchNet dataset [21] including extra preprocessing and cleaning [16]. It consists of 181,061 pairs of <*code, doc*> with 164,923 for training, 5,183 for validation and 10,955 for testing. From the codes, we extract the data following the process described in Sec. 2.1 resulting in triplets of <*CG, code, query*>. We use state-of-the-art CodeBERT [13, 30] as pretrained LM to encode both codes and queries. To encode the CGs, we implemented a gated attention network (GAT) [10, 39] with a global attention pooling layer [26] on top of it.

Models training and test. We fine-tune CodeBERT with the <*code tokens, query*> pairs as baseline and joint-learn our CCGNN model, *i.e.,* code concepts GNN, with the pretrained version of CodeBERT, using the triplets <*CG, code, query*>. We fine-tune/train both CodeBERT and CCGNN for 10 epochs. To test our models, we use the same evaluation setup as in Husain *et al.*'s [21] work where the
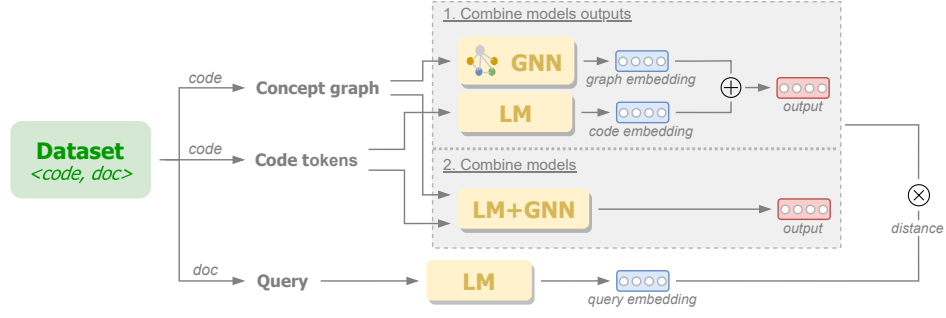
Figure 3: Overview of our framework. Given a data sample <*code, doc*>, we extract a query from the *doc* and code tokens and a concept graph (CG) from the *code* (Sec. 2.1). A language model (LM) is used to get a representation of the query and the LM is joint-learned with a concept graph neural network (GNN) with as learning objective to minimize the distance between the representations of the code+CG and the query (Sec. 2.2).
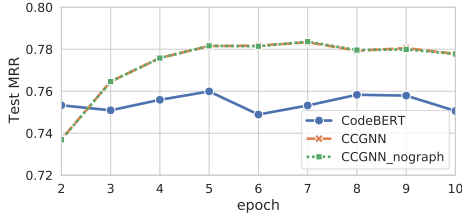
Figure 4: Comparison of the test MRR of CodeBERT and CCGNN (code concept GNN) fine-tuned *w.r.t* the number of training epochs.

model has to find the most-relevant code snippet among 1000 candidates given a natural language query.

Results. In Fig. 4, we report the evolution of the mean reciprocal rank (MRR) on the test set for CodeBERT and CCGNN. The third alternative CCGNN_nograph is equivalent to CCGNN but the CG representations are not used at test time. We can observe that our approach slightly increases the baseline reaching more than **0.78** of MRR. However, we performed *t*-tests between CodeBERT and CCGNN for each epoch which showed no significant difference between both models. We did not perform *t*-test between CodeBERT and CCGNN_nograph as the MRR curve of the latter is identical to CCGNN. This means that in our multi-modal learning approach, the concept graphs representations obtained with the GNN allows to guide the LM of code towards learning better representations. In addition, it does not require to load the CGs and the GNN to achieve better results at test which can be useful for low-computational setups. Finally, even though we show little improvement, we used a very simple combination method that did not require a lot of engineering which might suggest that more sophisticated combination methods could bring impressive gains.

4 FUTURE PLANS

In this preliminary work, we have shown that it is possible to augment an existing powerful pretrained LM of code by incorporating code concept graphs as part of the training process and by joint-learning a LM with a GNN. Besides, our experiments show that our proposal is worth investigating as we achieve better results than a state-of-the-art LM using a simple and straightforward joint-learning method. Consequently, we intend to explore more sophisticated combining methods such as one based on combining the architectures of both learning models instead of combining their outputs. Additionally, even though we illustrate our framework for code search, our code concept graphs-augmented multi-modal learning approach applies to lots of other end-tasks with few adjustments and by modifying the learning objective. Therefore, we envision to design adaptations of our framework to tackle more tasks and evaluate the potential gains that it could bring. And, as our concept graphs intend to be as language agnostic as possible, we envision to extract equivalent CG for more programming languages and leverage transfer learning to apply our framework in several languages. Finally, our proposed framework relies on multi-modal learning and could apply to a lot of other software data modalities. Therefore, we plan on comparing different combination of data modalities in the future.

5 CONCLUSION

In this paper, we present an original idea to abstract knowledge from software using multi-modal learning approaches. The novelty of our vision is twofold. Firstly, it lies on the idea of combining multiple data modalities at different level of abstraction extracted from the same software artifact to modeling the programming world. And secondly, we propose to abstract semantic relationships between identifiers into concept graphs to learn from a high-level structured modality. We conducted a preliminary evaluation of our framework by joint-learning a LM that focuses on learning from code syntax and a GNN leveraging our code concept graphs. The results show gains in the effectiveness of the models for code search using a straightforward combination method of the learning models. Consequently, this preliminary work prompts us to carry on our work in a research line focusing on multi-modal learning approaches for software engineering while keeping a truly data-driven perspective in mind.

REFERENCES

[1] Miltiadis Allamanis, Earl T Barr, Premkumar Devanbu, and Charles Sutton. 2018. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)* 51, 4 (2018), 1–37.

[2] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs.

[3] Miltiadis Allamanis and Charles Sutton. 2013. Mining source code repositories at massive scale using language modeling. (2013), 207–216.

[4] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2018. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400* (2018).

[5] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[6] Mattia Atzeni and Maurizio Atzori. 2017. CodeOntology: RDF-ization of source code. In *International Semantic Web Conference.* Springer, 20–28.

[7] Jacob Austin, Augustus Odena, Maxwell Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie Cai, Michael Terry, Quoc Le, et al. 2021. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732* (2021).

[8] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. 2014. Neural machine translation by jointly learning to align and translate. *arXiv preprint arXiv:1409.0473* (2014).

[9] Marc Brockschmidt, Miltiadis Allamanis, Alexander L Gaunt, and Oleksandr Polozov. 2018. Generative code modeling with graphs. *arXiv preprint arXiv:1805.08490* (2018).

[10] Shaked Brody, Uri Alon, and Eran Yahav. 2021. How Attentive are Graph Attention Networks? arXiv:2105.14491 [cs.LG]

[11] Kun Cao and James Fairbanks. 2019. Unsupervised Construction of Knowledge Graphs From Text and Code. *arXiv preprint arXiv:1908.09354* (2019).

[12] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[13] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. 2020. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155* (2020).

[14] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. 2017. Neural message passing for quantum chemistry. In *International conference on machine learning.* PMLR, 1263–1272.

[15] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE).* IEEE, 933–944.

[16] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366* (2020).

[17] Vincent J Hellendoorn and Premkumar Devanbu. 2017. Are deep neural networks the best choice for modeling source code?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering.* 763–773.

[18] Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations.*

[19] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[20] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[21] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. Codesearchnet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019).

[22] Azanzi Jiomekong, Gaoussou Camara, and Maurice Tchuente. 2019. Extracting ontological knowledge from Java source code using Hidden Markov Models. *Open Computer Science* 9, 1 (2019), 181–199.

[23] Dan Jurafsky. 2000. *Speech & language processing.* Pearson Education India.

[24] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE, 1073–1085.

[25] Ryan Kiros, Ruslan Salakhutdinov, and Rich Zemel. 2014. Multimodal neural language models. In *International conference on machine learning.* PMLR, 595–603.

[26] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. 2017. Gated Graph Sequence Neural Networks. arXiv:1511.05493 [cs.LG]

[27] Yingming Li, Ming Yang, and Zhongfei Zhang. 2018. A survey of multi-view representation learning. *IEEE transactions on knowledge and data engineering* 31, 10 (2018), 1863–1883.

[28] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, et al. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *arXiv preprint arXiv:2102.04664* (2021).

[29] Mitchell Marcus, Beatrice Santorini, and Mary Ann Marcinkiewicz. 1993. Building a large annotated corpus of English: The Penn Treebank. (1993).

[30] Microsoft. [n. d.]. Huggingface Implementation of CodeBERT. https://huggingface.co/microsoft/codebert-base.

[31] Jiquan Ngiam, Aditya Khosla, Mingyu Kim, Juhan Nam, Honglak Lee, and Andrew Y Ng. 2011. Multimodal deep learning. In *ICML.*

[32] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[33] Veselin Raychev, Martin Vechev, and Eran Yahav. 2014. Code completion with statistical language models. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 419–428.

[34] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. 2008. The graph neural network model. *IEEE transactions on neural networks* 20, 1 (2008), 61–80.

[35] Kavitha Srinivas, Ibrahim Abdelaziz, Julian Dolby, and James P. McCusker. 2020. Graph4Code: A Machine Interpretable Knowledge Graph for Code. *CoRR* abs/2002.09440 (2020). https://arxiv.org/abs/2002.09440

[36] Alexey Svyatkovskiy, Sebastian Lee, Anna Hadjitofi, Maik Riechert, Juliana Vicente Franco, and Miltiadis Allamanis. 2021. Fast and memory-efficient neural code completion. In *2021 IEEE/ACM 18th International Conference on Mining Software Repositories (MSR).* IEEE, 329–340.

[37] Kai Sheng Tai, Richard Socher, and Christopher D Manning. 2015. Improved semantic representations from tree-structured long short-term memory networks. *arXiv preprint arXiv:1503.00075* (2015).

[38] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems.* 5998–6008.

[39] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. 2018. Graph Attention Networks. arXiv:1710.10903 [stat.ML]