2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE) | 978-1-6654-0337-5/21/$31.00 ©2021 IEEE | DOI: 10.1109/ASE51524.2021.9678927

What do pre-trained code models know about code?

Anjan Karmakar
Free University of Bozen-Bolzano
Bolzano, Italy
akarmakar@unibz.it

Romain Robbes
Free University of Bozen-Bolzano
Bolzano, Italy
rrobbes@unibz.it

***Abstract*—Pre-trained models of code built on the transformer architecture have performed well on software engineering (SE) tasks such as predictive code generation, code summarization, among others. However, whether the vector representations from these pre-trained models comprehensively encode characteristics of source code well enough to be applicable to a broad spectrum of downstream tasks remains an open question.**

One way to investigate this is with diagnostic tasks called *probes*. In this paper, we construct four probing tasks (probing for surface-level, syntactic, structural, and semantic information) for pre-trained code models. We show how probes can be used to identify whether models are deficient in (understanding) certain code properties, characterize different model layers, and get insight into the model sample-efficiency.

We probe four models that vary in their expected knowledge of code properties: `BERT` (pre-trained on English), `CodeBERT` and `CodeBERTa` (pre-trained on source code, and natural language documentation), and `GraphCodeBERT` (pre-trained on source code with dataflow). While `GraphCodeBERT` performs more consistently overall, we find that `BERT` performs surprisingly well on some code tasks, which calls for further investigation.

***Index Terms*—probing, source code models, transformers, software engineering tasks**

I. Introduction

The outstanding success of transformer-based [31] pre-trained models in NLP such as `BERT` [14], has inspired the creation of a number of similar pre-trained models for source code [2], [12], [21], [27], [30], [34]. These pre-trained models are first trained on a large corpus of code in a self-supervised manner and then fine-tuned on downstream tasks.

The progress made with pre-trained source code models is genuinely encouraging with applications in software security, software maintenance, software development and deployment. And although the pre-trained vector embeddings from the transformer models have worked well on many tasks, it remains unclear what exactly these models learn about code—specifically what aspects of code structure, syntax, and semantics are known to it. Thus, our work is motivated by the need to assess the properties of code that are learned by pre-trained embeddings, in order to build accurate, robust, and generalizable models for code, beyond single-task models.

An emerging field of research addresses this objective by means of *probes*. Probes are diagnostic tasks, in which a simple classifier is trained to predict specific properties of its input, based on the *frozen* vector embeddings of a pre-trained model. The degree of success in the probing tasks indicates whether the information probed for is present in the pre-trained embeddings. Probing has been extensively used for natural language models, and has already begun to pick up steam with numerous probing tasks [1], [4], [8], [13], [24], [26], [28], [29] investigating a diverse array of natural language properties.

In this work, we adapt the probing paradigm to pre-trained source code models. We assess the hidden state embeddings of multiple models, and determine their ability to capture elemental characteristics related to code, that may be suitable for use in several downstream SE tasks. We evaluate `CodeBERT` [15], `CodeBERTa` [34], and `GraphCodeBERT` [16], with `BERT` [14] as our transformer baseline. As an initial study, we have chosen to work with `BERT` and its code-trained descendants, as it provides a ground for comparison among natural language models, models trained jointly on natural language and code, models trained just on code, and models trained on code with additional structural information.

We construct four initial probing tasks for this purpose: AST Node Tagging, Cyclomatic Complexity Prediction, Code Length Prediction, and Invalid Type Detection. These four tasks are meant to assess whether pre-trained models are able to capture different aspects of code, specifically the syntactic, structural, surface-level and semantic information respectively. The tasks were chosen to cover the most commonly identifiable abstractions of code, although more tasks are needed to thoroughly probe for each type of code abstraction.

This paper makes the following contributions:

- An introduction to probing for pre-trained code models. We introduce four probing tasks each probing a particular characteristic of code, and release the corresponding task datasets publicly.
- A preliminary empirical study, based on probing tasks and pre-trained code models, that highlights the potential of probes as a pseudo-benchmark for pre-trained models.
- A discussion on the efficacy of pre-trained models. We show to what extent different code properties are encoded in pre-trained models.

Overall, our probes suggest that the models do encode the syntactic and semantic properties, to varying degrees. While we find that models that have more knowledge of source code tend to perform better at the more code-specific probing tasks, yet, the difference in performance between the baseline and the source code models are smaller than expected. This calls for further study of the phenomenon, and for increased effort in designing pre-training procedures that better capture diverse source code characteristics.

978-1-6654-0337-5/21/$31.00 ©2021 IEEE
DOI 10.1109/ASE51524.2021.00163

II. Background

A probe fundamentally consists of a probing task and a probing classifier. A probing task is an auxiliary diagnostic task that is constructed to determine whether a specific property is encoded in the pre-trained model weights. Probing is useful when assessing the raw predictive power of pre-trained weights without any sort of fine-tuning with (downstream) task data. Probing tasks are often simple in nature compared to downstream tasks to minimize interpretability problems.

A probing classifier, on the other hand, is used to train on the probing task where the input vectors of the training samples are extracted from the *frozen* hidden layers of the pre-trained model. Importantly, the probing classifier, which is usually a linear classifier, is *simple* with no hidden layers of its own. If a simple probing classifier can predict a given attribute from the pre-trained embeddings, the original model most likely encodes it in its hidden layers. Usually, the raw accuracies from a probe are not the focus of the study; rather, the probe is used to assesses whether a model encodes a characteristic *better* than another, or compares several model layers.

Related work. Studies in NLP research have shown how several pre-trained natural language models encode different linguistic properties such as sentence length, and verb tense, among other properties [13]. Studies such as [19] show that `BERT` encodes phrase-level information in the lower layers, and a hierarchy of linguistic information in the intermediate layers, with surface features at the bottom, syntactic features in the middle and semantic features at the top of a vector space. Other studies focus on word morphology [8], or syntax [26], to name a few. Studies of the `BERT` models alone have spawned a subfield known as *BERTology* with over 150 studies surveyed [25]. While probing is well established in NLP, it is almost absent for source code models. The only example we are aware of uses a single coarse task (programming language identification)—and is not the focus of the paper [15].

III. Probing Source Code

Probing Tasks. In order to determine whether the pre-trained vector embeddings of source code transformer models reflect code understanding in terms of syntactic, semantic, structural, and surface-level characteristics, we have constructed four diverse probing tasks.

AST Node Tagging (AST) As Abstract Syntax Trees (ASTs) are the basis of many structured source code representations [5]–[7], [9], [17], [23], [32], [33], they emerge as a rational choice to evaluate pre-trained source code models on syntactic understanding. In order for a pre-trained code model to be good at code tasks such as code completion, it must necessarily learn and interpret the syntactic structure of a sequence of code tokens and predict a syntactically valid next token. Thus, identifying AST node tags often is a hidden prerequisite to solving a given code task—making it a suitable contender for probing any pre-trained source code model. We probe the pre-trained models with this task to determine to what extent **syntactic information** is encoded in the model layers.

Cyclomatic Complexity (CPX) To probe whether some sort of code **structure information** is encoded in the hidden layers of a pre-trained model, we construct the cyclomatic complexity task. Since the complexity is an inherent characteristic of any code snippet, the models should be able to predict it without explicit fine-tuning. Furthermore, since the complexity of a code snippet depends on the number of linearly independent paths through the code snippet, predicting it based simply on the sequence of tokens might be a challenge.

Code Length Prediction (LEN) We conjecture that the length of a code snippet, especially when it is fed into the model as a sequence of code tokens, should be easy to predict for the transformer models. To determine whether the code transformers encode such elementary **surface information**, we probe the models with the code length prediction task.

Invalid Type Detection (TYP) To understand to what extent pre-trained code models are aware of code semantics and are able to distinguish between semantically valid snippets of code from invalid ones—keeping both syntactically legitimate, we construct an invalid type detection task. For this probing task, the negative samples consist of code snippets where some *primitive data types* are misspelled intentionally. The code transformers are probed to determine if they are able to classify code snippets based on invalid **semantic information**.

Probing data & labels. We gather our data from a subset of the 50K-C dataset of compilable Java projects [22]. We construct ASTs of method-level code snippets from several of the largest projects and collect a diverse range of AST node tags as labels: totaling 20 classes of node type labels. We use the metrix++ tool to obtain the complexity labels (with complexities ranging from 0 to 9). We tokenize our training samples with ANTLR to gather the length labels in five class-bins (0-50, 50-100, etc.). The labels for invalid types are obtained by interchanging consecutive characters at random indexes to resemble misspelled types. Our probing datasets have 10k samples for each task and are class-balanced.

Pre-trained Models. We probe four state-of-the-art models: `BERT`, `CodeBERT`, `CodeBERTa`, and `GraphCodeBERT`. All the models are built upon the multi-layer bidirectional transformer introduced by Vaswani et al. [31]. Other than `BERT`, which is our baseline, pre-trained on a large corpus of English data, all the other transformer models are pre-trained on the CodeSearchNet dataset extracted from GitHub [18], which includes 6.4 million methods across six programming languages. While `CodeBERT` and `CodeBERTa` gain knowledge of code by training on source code and natural language documentation, `GraphCodeBERT` goes further, encoding even data-flow information extracted from the ASTs.

Probing Classifier. We train a simple linear classifier that takes the input feature vectors from the hidden layers of a pre-trained code transformer. This is done to determine which of the code properties are linearly correlated with pre-trained model embeddings. Since a linear classifier has a basic model architecture with no hidden units, therefore it must heavily rely on the pre-trained embeddings to do well in the tasks.

IV. Early Results and Discussion

The results and observations from the probe analyses are discussed below. It is important to note that our interest is more on the difference between the models rather than the general accuracy of the probes.

TABLE I
Probing Task Accuracies

Model	LEN *surface*	AST *syntactic*	CPX *structural*	TYP *semantic*
Naive	20.00	05.00	10.00	50.00
BERT	**76.05**	89.90	42.65	86.85
CodeBERT	68.15	89.45	41.40	93.85
CodeBERTa	70.35	**92.55**	40.80	90.10
GraphCodeBERT	71.10	85.50	**46.70**	**97.20**

A. Model Analysis

Surface-level information. BERT performs the best with 76.05% accuracy for code length prediction. Considering the amount of training data BERT has seen and given the task of predicting the number of tokens from the input sequence, it is not unexpected that BERT does well on this task.

Syntactic information. For AST node tagging, we find that with enough training samples, all of the pre-trained models are able to achieve classification accuracies beyond 85%, reaching up to 92.55% for CodeBERTa [34]. This shows that syntax-related information requisite for the node tagging task is encoded in the model hidden layers.

Structural information. GraphCodeBERT, having been trained with the most structured information, performs the best with 46.70% accuracy for cyclomatic complexity prediction. The standard BERT model also performs well with 42.65% accuracy. This raises the question whether BERT encodes structural information of code, or alternatively, whether the pre-trained code models fail to encode it effectively.

Semantic information. GraphCodeBERT has the highest accuracy (97.20%) for invalid type prediction. Although all of the pre-trained code models exhibit prediction accuracies beyond 90%, BERT is not far behind in terms of accuracy. Note that since this task is a simple binary classification task, a higher naive baseline accuracy is expected.

Overall observations. In all cases, the hidden-layer vector embeddings of the pre-trained models seem to encode information that correlates with code characteristics to varying degrees. The structure-based code task (CPX) is the most challenging: implying that the pre-trained embeddings may not have direct linear correlations with the probed code properties, as linear classifiers are not able to extract it effectively.

While GraphCodeBERT has the most consistent performance overall, it is surprising that BERT, pre-trained just on English text, exhibits a similar competitive performance against the other pre-trained code models— which *should* have more knowledge of code. BERT does considerably well on most tasks, even for the tasks which are more code-dependent, which calls for an extended investigation on this topic.

B. Layer analysis

Alain & Bengio [4] compare *probes* to thermometers used to measure the temperature (accuracies) simultaneously at several different locations (layers). Figure 1 shows the accuracies of the pre-trained source code models across all its layers.

All of the models have 12 layers, except CodeBERTa which has only 6. The accuracies are based on the pre-trained vector embeddings extracted from each of these hidden layers ranging from 1-12. The encoding layer of each model is represented by layer 0 which displays naive baseline accuracy.

The pre-trained code models appear to have heterogeneous performance across the layers. We observe no single layer consistently performs optimally for all tasks, which is expected, indicating that the abstraction of the different learned code properties are spread across multiple layers.

BERT shows similar behavior for English [19], but it does not show this localization of performance for code—exhibiting fairly homogeneous performance across the layers.

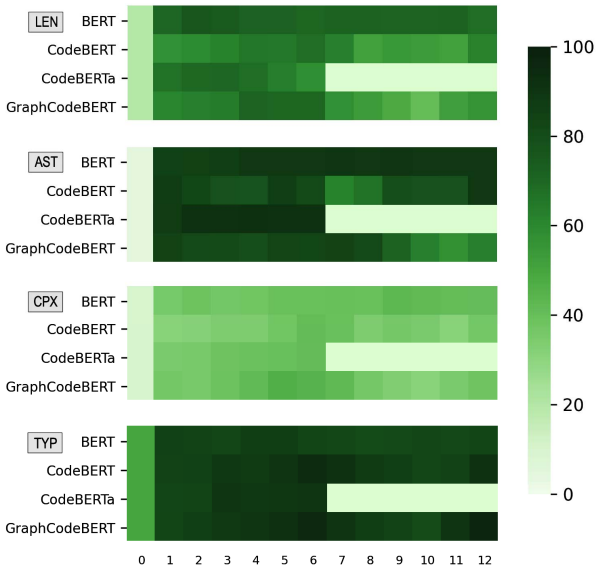

Fig. 1. Pre-trained model accuracies by layers.

C. Sample Size Analysis

Pre-trained models when fine-tuned on a downstream task, need only a fraction of the data a model trained from scratch needs to do well. Hence, when evaluating pre-trained models with probes it is also essential to study the effects of training data volume. Thus we evaluate how the probes perform when data is scarce - with 10% and 1% of the dataset size.

We limit our training to a maximum of 10,000 samples with the intuition that the general underlying syntactic, semantic, structural rules can be learned in a sample-efficient way and does not require memorization.

As expected, the overall model accuracies increase with the increase in the number of samples from 100 to 1000 to 10,000 samples. However, it is interesting to note that the irregularity

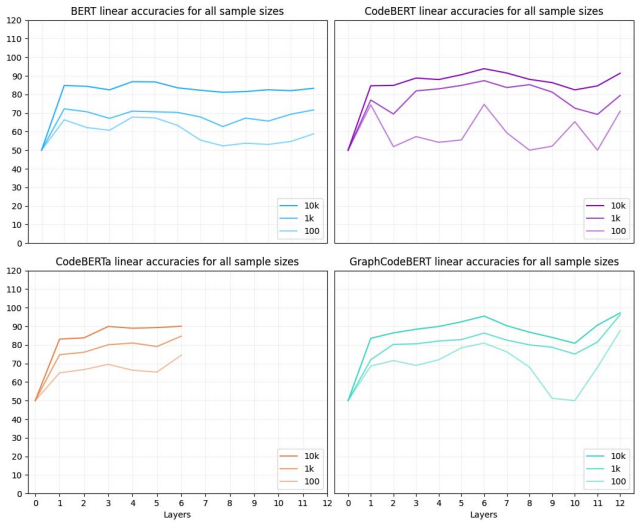

Fig. 2. Model accuracies by sample sizes for *Invalid Type Prediction*

(uneven trend) in accuracies from layer to layer is flattened as more samples are provided (Fig. 2). This applies to all tasks.

An intriguing observation is that `GraphCodeBERT`'s performance with just 100 samples exceeds `BERT`'s performance with 10,000 samples for the TYP task. We attribute this to `GraphCodeBERT` being much more sample-efficient for this task, as it can extract a lot of the signal from fewer samples. For the other tasks, such as AST and CPX where pre-trained code models improved upon the baseline, no such patterns implying sample-efficiency were present.

D. Discussion

Code properties. Our findings suggest that while certain code characteristics can be extracted from pre-trained code transformers with linear classifiers, implying that they are firmly encoded in the hidden layers, others, such as cyclomatic complexity, cannot be extracted as effectively. We plan to investigate whether these characteristics can be extracted with non-linear classifers such as an MLP with its own hidden layers [13]. In general, the results also point to the idea that more effective pre-training procedures that can make these characteristics more accessible should be explored.

Model performance. For all of the tasks `BERT` has shown competitive performance against the other `BERT`-style code transformers. It shows understanding of surface information better than the others, while being the next best model for syntax- and structure-based evaluations. `CodeBERTa` and `CodeBERT` show promising results for syntax and semantic understanding respectively, with an improvement on the baseline `BERT` model. `GraphCodeBERT` struggles to reach accuracies beyond 85% on AST Node Tagging task—we hypothesize it is missing out on the data-flow context while making predictions based on a single code token. Besides that, it is the most consistent model with up to 10-12% improvements in accuracies from the `BERT` baseline.

At first glance, the baseline results are surprising, since `BERT` is supposed to have *no specific knowledge of source code*. These concerns are somewhat alleviated by the lack of localization of code properties in `BERT` layers. In addition, tasks such as TYP show a clear advantage for all pre-trained source code models, where they exhibit competitive accuracies with fewer samples, particularly `GraphCodeBERT`.

We hypothesize that, rather than `BERT` possessing competitive knowledge of source code, the most likely cause is that the source code models were not substantially as effective in encoding the probed code characteristics as was expected. Further investigation on more tasks could confirm this; if confirmed this would also suggest that more effective pre-training procedures should be explored.

V. Conclusion & Future Work

As more pre-trained code models are introduced to the SE community at a rapid pace, through IDE extensions, plugins, and web engines, e.g. Tabnine, IntelliCode [27], TransCoder [21], and more recently Github Copilot [11]—it becomes imperative for us to be aware of their capabilities and drawbacks. In order to do so, we probe into four publicly-available pre-trained source code models, surveying a diverse set of code characteristics with relevant and representative probing tasks.

We show how probes help us to gauge the strengths and weaknesses of a model, to understand the role played by the individual hidden layers in model performance, to verify the linear extractability of properties, to get insight on a model's sample efficiency for a given task, and overall to peek into the "black boxes" that are large-scale pre-trained models.

With our initial probes, we were surprised to notice the slim margin of difference in performance between models that should have no knowledge of code and models that do. This clearly needs to be investigated further, and, if confirmed, would suggest that there is room for research in more advanced pre-training techniques for source code models, so that they can effectively leverage their knowledge of source code.

As future work, we plan to construct further probing tasks evaluating additional source code characteristics, while adding more tasks based on structure, syntax and semantics. Furthermore, we intend to report a benchmark-style comparison study of additional pre-trained models such as `CuBERT` [20], `C-BERT` [10], `PLBART` [3], and `Codex` [11].

We plan to further extend our suite of probes to more comprehensive source code properties, including context-based probes for applications such as code search and summarization, and extend it to several languages, in order to support a broader array of pre-trained code models. In the long run, a suite of probing tasks could be used to evaluate novel pre-trained source code models, thereby forming a pseudo-benchmark during the development phase, making sure that these models do encode important source code characteristics.

Reproducibility. All code, datasets, and experimental results are made available online for replication purposes on github[1].

[1]https://github.com/giganticode/probes

REFERENCES

[1] Yossi Adi, Einat Kermany, Yonatan Belinkov, Ofer Lavi, and Yoav Goldberg. Fine-grained analysis of sentence embeddings using auxiliary prediction tasks, 2017.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization, 2020.

[3] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Unified pre-training for program understanding and generation, 2021.

[4] Guillaume Alain and Yoshua Bengio. Understanding intermediate layers using linear classifier probes, 2018.

[5] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. *CoRR*, abs/1711.00740, 2017.

[6] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code, 2019.

[7] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code, 2018.

[8] Yonatan Belinkov, Nadir Durrani, Fahim Dalvi, Hassan Sajjad, and James Glass. What do neural machine translation models learn about morphology? In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 861–872, Vancouver, Canada, July 2017. Association for Computational Linguistics.

[9] Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs, 2019.

[10] Luca Buratti, Saurabh Pujar, Mihaela A. Bornea, J. Scott McCarley, Yunhui Zheng, Gaetano Rossiello, Alessandro Morari, Jim Laredo, Veronika Thost, Yufan Zhuang, and Giacomo Domeniconi. Exploring software naturalness through neural language models. *CoRR*, abs/2006.12641, 2020.

[11] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021.

[12] Colin B. Clement, Dawn Drain, Jonathan Timcheck, Alexey Svyatkovskiy, and Neel Sundaresan. Pymt5: multi-mode translation of natural language and python code with transformers, 2020.

[13] Alexis Conneau, German Kruszewski, Guillaume Lample, Loïc Barrault, and Marco Baroni. What you can cram into a single $&!#* vector: Probing sentence embeddings for linguistic properties. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2126–2136, Melbourne, Australia, July 2018. Association for Computational Linguistics.

[14] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[15] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. Codebert: A pre-trained model for programming and natural languages, 2020.

[16] 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. Graphcodebert: Pre-training code representations with data flow, 2021.

[17] Vincent J. Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. Global relational models of source code. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[18] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *arXiv:1909.09436 [cs, stat]*, September 2019. arXiv: 1909.09436.

[19] Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. What does BERT learn about the structure of language? In *Proceedings of the 57th Annual Meeting of the Association for Computational Linguistics*, pages 3651–3657, Florence, Italy, July 2019. Association for Computational Linguistics.

[20] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. Pre-trained contextual embedding of source code. *CoRR*, abs/2001.00059, 2020.

[21] Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages, 2020.

[22] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 50k-c: A dataset of compilable, and compiled, java projects. In *2018 IEEE/ACM 15th International Conference on Mining Software Repositories (MSR)*, pages 1–5, 2018.

[23] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing, 2015.

[24] Matthew E. Peters, Mark Neumann, Luke Zettlemoyer, and Wen-tau Yih. Dissecting contextual word embeddings: Architecture and representation. *CoRR*, abs/1808.08949, 2018.

[25] Anna Rogers, Olga Kovaleva, and Anna Rumshisky. A primer in bertology: What we know about how bert works. *Transactions of the Association for Computational Linguistics*, 8:842–866, 2020.

[26] Xing Shi, Inkit Padhi, and Kevin Knight. Does string-based neural MT learn source syntax? In *Proceedings of the 2016 Conference on Empirical Methods in Natural Language Processing*, pages 1526–1534, Austin, Texas, November 2016. Association for Computational Linguistics.

[27] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intellicode compose: Code generation using transformer, 2020.

[28] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R. Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R. Bowman, Dipanjan Das, and Ellie Pavlick. What do you learn from context? probing for sentence structure in contextualized word representations. *CoRR*, abs/1905.06316, 2019.

[29] Ian Tenney, Patrick Xia, Berlin Chen, Alex Wang, Adam Poliak, R Thomas McCoy, Najoung Kim, Benjamin Van Durme, Samuel R. Bowman, Dipanjan Das, and Ellie Pavlick. What do you learn from context? probing for sentence structure in contextualized word representations, 2019.

[30] Michele Tufano, Dawn Drain, Alexey Svyatkovskiy, and Neel Sundaresan. Generating accurate assert statements for unit test cases using pretrained transformers, 2020.

[31] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need, 2017.

[32] Huihui Wei and Ming Li. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 3034–3040, 2017.

[33] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. Lambdanet: Probabilistic type inference using graph neural networks, 2020.

[34] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, and Jamie Brew. Huggingface's transformers: State-of-the-art natural language processing. *CoRR*, abs/1910.03771, 2019.