The Effectiveness of Pre-Trained Code Embeddings

Ben Trevett
Heriot-Watt University
Edinburgh, United Kingdom
bbt1@hw.ac.uk

Donald Reay
Heriot-Watt University
Edinburgh, United Kingdom
d.s.reay@hw.ac.uk

N. K. Taylor
Heriot-Watt University
Edinburgh, United Kingdom
n.k.taylor@hw.ac.uk

***Abstract*—Few machine learning applications applied to the domain of programming languages make use of transfer learning. It has been shown that in other domains, such as natural language processing, that transfer learning improves performance on various tasks and leads to faster convergence. This paper investigates the use of transfer learning on machine learning models for programming languages - focusing on two tasks: method name prediction and code retrieval. We find that, for these tasks, transfer learning provides improved performance, as it does to natural languages. We also find that these models can be pre-trained on programming languages that are different from the downstream task language and that even pre-training models on English language data is sufficient to provide similar performance as pre-training on programming languages. We believe this is because these models ignore syntax and instead look for semantic similarity between the named variables in source code.**

***Index Terms*—machine learning, neural networks, clustering, transfer learning**

I. Introduction

The aim of transfer learning is to improve the performance on task $\mathcal{T}_i$ by using a model that has been first trained on task $\mathcal{T}_j$, i.e. the model has been *pre-trained* on task $\mathcal{T}_j$. For example, a model that is first trained to predict a missing word within a sentence is then trained to predict the sentiment of a sentence. Pre-training and using pre-trained machine learning models is commonly used in natural language processing (NLP). Traditionally transfer learning in NLP only pre-trained the *embedding layers*, those that transformed words into vectors, using methods such as word2vec [1, 2], GloVe [3] or a language model.

Recently the NLP field has moved on to pre-training all layers within a model and using a task specific "head" that contains the only parameters that which not pre-trained. Examples of this approach are: ULMFiT [4], ELMo [5] and BERT [6]. The use of these pre-trained models have been shown to achieve state-of-the-art results in NLP tasks such as text classification, question answering and natural language inference.

There have been advances in applying machine learning to modelling programming languages, specifically deep learning using neural networks. Common tasks include method name prediction [7, 8, 9] and code retrieval from natural language queries [10].

Ben Trevett is funded by an Engineering and Physical Sciences Research Council (EPSRC) grant and the ARM University Program.

Pre-training models to be used for transfer learning requires a substantial amount of training data. For example, BERT [6] was trained on a dataset containing billions of words. There is similar data available for programming languages, e.g. open source repositories on websites such as `GitHub`, which can be used to take advantage of pre-training and transfer learning techniques. However, there has been little effort in this domain.

In this paper, we explore transfer learning on programming languages. We test the transfer learning capabilities in two common tasks in the programming language domain: code retrieval and method name prediction. We pre-train our models on datasets with different characteristics: one that is made solely of the downstream task language, one that contains data in the downstream task language and other programming languages, a dataset of programming languages that does not contain the downstream task language and also, a dataset that does not containing any programming languages at all. We show that transfer learning provides performance improvements on tasks in programming languages. Our results using models pre-trained on the different datasets suggest that semantic similarity between the variables and method names are more important than the source code syntax for these tasks.

Our contributions are: 1) We propose a method for performing transfer learning in the domain of programming languages. 2) We show that transfer learning improves performance across the two tasks of code retrieval and method name prediction. 3) We show that the programming language of the pre-training dataset does not have to match that of the downstream task language. 4) We show that the pre-training dataset English language data provides comparable results to pre-trained on programming languages data.

As far as the authors are aware, this is the first study into the use of pre-training on code which investigates the use of datasets containing data that does not match that of the downstream task, and also datasets which do not contain data in the downstream task language.

II. Related Work

A. Transfer Learning

For transfer learning, the traditional methods, such as word2vec [1, 2] and GloVe [3] are only able to pre-train the embedding layers within a model. These methods have been succeeded by recent research on contextual embeddings using language models, such as ULMFiT [4], ELMO [5] and BERT [6] which have shown to provide state-of-the-art performance

TABLE I
CODESEARCHNET CORPUS STATISTICS

Language	Number of Examples
Go	347 789
Java	542 991
JavaScript	157 988
PHP	717 313
Python	503 502
Ruby	57 393
Total	2 326 976

in many NLP tasks, but have not yet been widely applied to programming languages.

B. Machine Learning on Source Code

The use of machine learning on source code has received an increased amount of interest recently [11]. On tasks such as code retrieval [10], method name prediction [8, 9, 7], generating natural language from source code or vice versa [12, 13] and correcting errors in code [14].

C. Pre-training on Source Code

There has been little work on pre-training for source code. Chen and Monperrus [15] have performed a literature study. Wainakh et al. have evaluated [16] representations learned from source code. Research that uses embeddings pre-trained on source code which are then applied to downstream tasks is limited. Two examples are NL2Type [17], which predicts types for JavaScript functions and DeepBugs [18], which detects certain classes of bugs within code. Both of these works only use the word2vec algorithm for pre-training the embeddings and do not perform transfer learning on separate datasets. Recently Feng et al. [19] introduced CodeBERT, which pretrains a Transformer model for the code retrieval task, but does not perform experiments on the scenario where there is a mismatch between pretraining and downstream task languages.

III. TASKS

We perform two tasks: code retrieval and method name prediction. Both tasks use the CODESEARCHNET CORPUS [1] [10], statistics for which are shown in table I. This corpus contains 2 million methods and their associated documentation, represented as a natural language query. The dataset contains 6 programming languages: Go, Java, JavaScript, PHP, Python and Ruby. We evaluate both tasks only on the Java examples within the dataset.

A. Code Retrieval

The code retrieval task is to accurately pair each method, $\boldsymbol{c}_i$, with query, $\boldsymbol{d}_i$, where both the method and query are a sequence of tokens. This is done by *encoding* both the code and query tokens into a high-dimensional representation and then measuring the distance between these representations. The goal is to have $f(\boldsymbol{c}_i) \approx g(\boldsymbol{d}_i)$ and $f(\boldsymbol{c}_i) \neq g(\boldsymbol{d}_j)$ for $i \neq j$, where f and g represent the code and query encoders, respectively. Performance is measured in MRR (Mean Reciprocal Rank) as in [10], measured between 0 and 1.

[1]https://github.com/github/CodeSearchNet

B. Method Name Prediction

The method name prediction task is to predict the method name, $\boldsymbol{n}_i$ given the method body, $\boldsymbol{b}_i$. The method body and names are a sequence of tokens, where the method name has been split into sub-tokens. Wherever the method name appears in the method body it has been replaced by a `<blank>` token. The model takes the method body as input and outputs the method name sub-tokens, one at a time. Performance is measured by F1 score as in [9], measured between 0 and 1.

IV. METHODOLOGY

A. Models

For both tasks we use both the *Transformer* [20] and neural bag-of-words (NBOW) models. The Transformer uses multi-head self-attention mechanisms and learns to attend over the relevant tokens within the input sequence to produce a final output representation for each token. We chose this model for two reasons: it provided the best results, on average, over the 6 programming languages in the CODESEARCHNET CORPUS, and BERT [6], a variant of the Transformer, is commonly used for state-of-the-art NLP tasks, especially when pre-trained to be used for transfer learning. We use the default hyper-parameters from the Transformer model provided for the CODESEARCHNET CORPUS. The NBOW model is used as a baseline and has a single embedding layer which embeds the sequence of input tokens into a sequence of vectors.

For both tasks, only the task-specific head of the model is changed. In the code retrieval task the head performs a weighted sum over the outputs of the model, as in [10], where the weights are learned by the head itself. For the method name prediction task, the head is a gated recurrent unit (GRU) [21], similar to the architectures used in [9] and [7], which uses a weighted sum over the outputs of the model as its initial hidden state.

For comparison we train both models without any transfer learning, i.e. it is randomly initialized.

We pre-train the models as masked language models, following [6], with an affine layer head used to predict the masked token. They are trained until convergence, i.e. until the validation loss stops decreasing. For the code retrieval task, only the code encoder is pre-trained, the query encoder is learned from scratch every time.

To perform transfer learning, we take the pre-trained model, replace its head with the task-specific head and fine-tune it on the desired task. Again, it is trained until convergence. Each experiment is ran 5 times with different random seeds, the results of which are averaged together.

B. Datasets

We pre-train the model on 4 different datasets: `Java`, `6L`, `5L` and `English`. `Java` is only the Java code within the

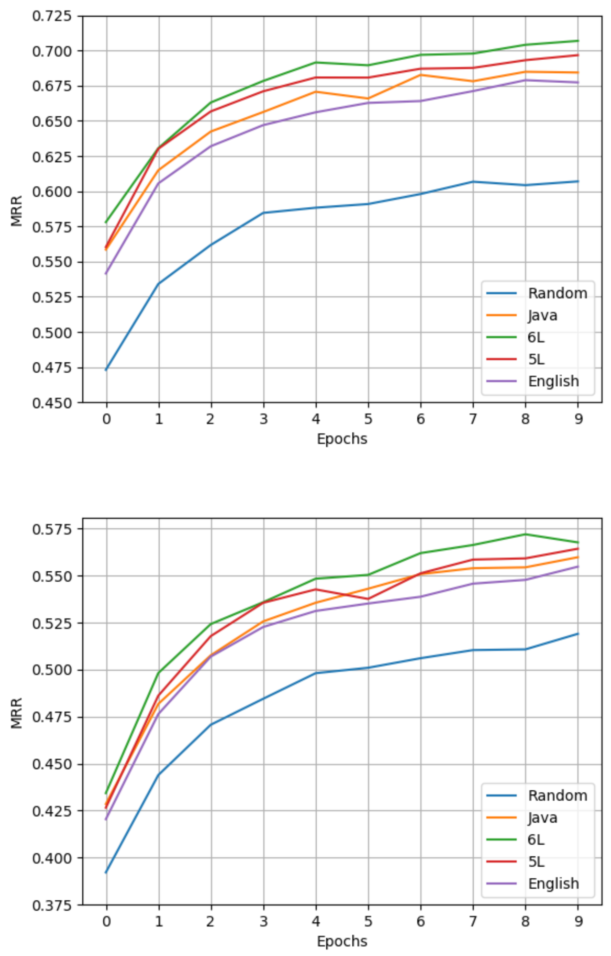

Fig. 1. Test MRR on code retrieval task for the Transformer (top) and NBOW (bottom) models.

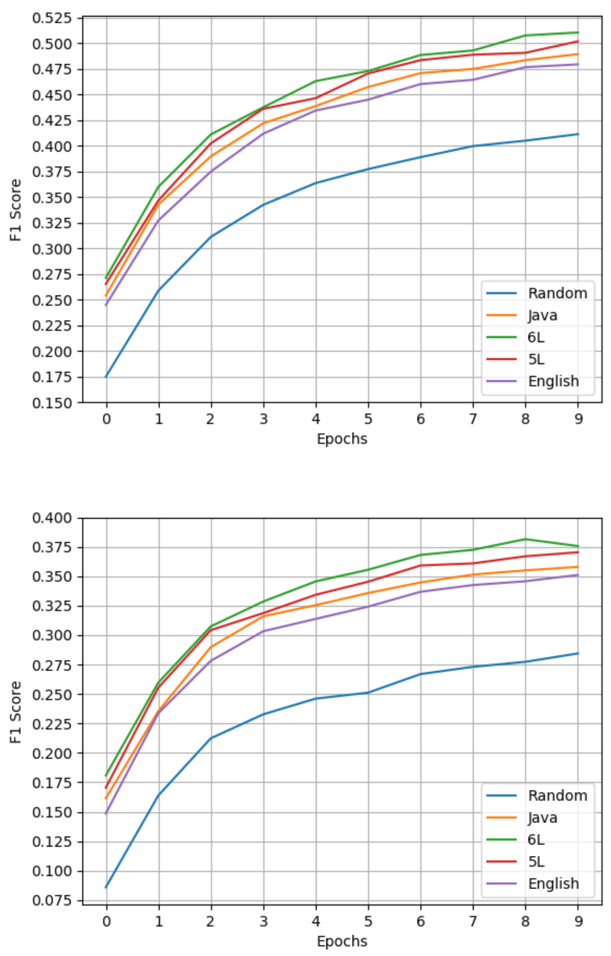

Fig. 2. Test F1 score on method name prediction task for the Transformer (top) and NBOW (bottom) models.

TABLE II
TEST MRR ON CODE RETRIEVAL TASK FOR THE TRANSFORMER (LEFT) AND NBOW (RIGHT) MODELS.

Initialization	MRR	Initialization	MRR
Random	0.6069	Random	0.5191
Java	0.6849	Java	0.5598
6L	**0.7068**	**6L**	**0.5721**
5L	0.6967	5L	0.5643
English	0.6789	English	0.5548

TABLE III
TEST F1 SCORE ON METHOD NAME PREDICTION TASK FOR THE TRANSFORMER (LEFT) AND NBOW (RIGHT) MODELS.

Initialization	F1	Initialization	F1
Random	0.4114	Random	0.2844
Java	0.4895	Java	0.3579
6L	**0.5106**	**6L**	**0.3815**
5L	0.5022	5L	0.3703
English	0.4796	English	0.3511

CODESEARCHNET CORPUS, and is the same data our model will be fine-tuned on for each task, i.e. first the Transformer is pre-trained as a masked language model and then trained on the same data for the desired task.

`6L` is comprised of all 6 languages in the CODESEARCHNET CORPUS. `5L` is made up of 5 languages from the CODESEARCHNET CORPUS: Go, JavaScript, PHP, Python and Ruby. It does not contain any Java code.

The `English` data is the WikiText-2 dataset [22], a collection of 600 English Wikipedia articles, consisting of 2 million tokens. Each dataset is formed into a train/valid/test proportional split of 80-10-10.

V. RESULTS

The test results for the code retrieval and method name prediction tasks are shown in tables II and III, respectively. The training curves for the code retrieval and method name prediction tasks are shown in figures 1 and 2.

For the code retrieval task, all 4 forms of pre-training achieve at least a relative 12% performance increase for the

```java
float getSurfaceArea (int radius) {
      return 4 * Math.PI * radius * radius;
}
```

```java
float getAspectRatio (int height, int width) {
      return height / width;
}
```

```python
def get_surface_area (radius):
      return 4 * math.pi * radius * radius
```

```python
def get_aspect_ratio (height, width):
      return height / width
```

Fig. 3. Two example functions in Java and their Python equivalents

Transformer model and 7% for the NBOW model. The `6L` data provides the best performance improvement for each model, 16% and 10% for the Transformer and NBOW models, respectively.

For the method name prediction task, again, all 4 forms on pre-training provide an increase in performance over the initialized parameters, with at least 16% relative improvement for the Transformer and 23% for the NBOW model. Again, the `6L` data provides the best performance increase, 24% and 34% for the Transformer and NBOW models.

VI. Discussion

A. Code Retrieval

Intuitively, the datasets that contain Java code, `Java` and `6L`, should give the best results. This is because they have been pre-trained on examples in the same language as the downstream task. However, the results for the code retrieval task show this is not the case, and that pre-training on data without any Java code, and even pre-training on data that is not in a programming language, give comparable results.

One potential reason for this is that the code retrieval task does not actually use programming language related tokens within the function. Consider the 2 Java methods in figure 3. For the function `getSurfaceArea` a fitting query would be *"a method that calculates surface area"*, similarly the function `getAspectRatio` would match with the query *"a method that calculates aspect ratio"*. The semantic similarity between the tokens in the function and query give a strong indication about how well they match.

As these models only need to focus on the semantic similarity between named variables and the query, the programming language related tokens, such as the type definitions and semicolons can virtually be ignored for this task. This implies that the presence of semantically sensible method and variable names, which match those in the query, are more important.

Thus, the reasons why the `6L`, `5L` and `English` datasets provide a performance increase is that they contain more examples of the context in which tokens appear. This increased number of contexts allows the model learn more examples of semantic similarity between tokens and then transfer this knowledge when the model is being fine-tuned.

B. Method Name Prediction

Predicting the method name from the method body may also largely rely on semantic similarity between tokens in the method name and the variable names in the method body. Models can learn to predict the method name largely based on using the method variables that have been given sensible semantic names, ignoring the programming language syntax.

Again, looking at the examples of figure 3, the model has to learn that *pi* and *radius* relate to *surface area* and that *height* and *width* are related to *aspect ratio*. Thus, the code specific tokens, such as the braces and semicolons, are seemingly irrelevant, which makes the Python functions shown in figure 3 identical to the Java functions.

This artifact of mainly needing to learn semantic similarity would again explain why the `6L`, `5L` and `English` datasets give a performance increase over the `Java` dataset. The masked language model pre-training learns the context of tokens, either within programming language functions or human language sentences. Knowing that the concept of *surface area* appears in nearby contexts to *pi* and *radius* is useful in the downstream task of predicting a method name from the method body.

This could also explain why the model pre-trained on the `Java` data did not manage to provide a larger improvement increase than the model pre-trained on the `6L` data - the model was over-fitting to contexts within the Java language examples.

C. Further Discussion

A quick experiment carried to test our hypothesis about code tokens being unnecessary was to train the model for each task with randomly initialized embeddings on the Java dataset with all of the code specific tokens removed. Concretely, we removed all type declarations, semicolons and braces. We carried out this experiment for both of the tasks described in this paper and this achieved comparable performance to the randomly initialized embeddings without the code specific tokens removed.

These experiments show similar findings to Hindle et al. [23]. The computer that will run the code does not need to understand the semantic meaning of a method or variable name, programs are written by humans to be understood by humans. Naming variables inside methods with semantically relevant names allows humans to understand the method's functionality easier. As long as humans continue to write code, these patterns can be learned and exploited by machine learning models to achieve beneficial results.

VII. Limitations

Although that we have shown that machine learning models for code should be pre-trained, we have only focused on two models, the Transformer and NBOW, and a single method of pre-training, as a masked language model. The Transformer model provides state-of-the-art transfer learning results for natural languages, but there is no guarantee that is also true of programming languages. Further work would perform transfer learning with different models and methods, comparing how each improves performance, as well as potentially inventing a novel model or method of pre-training specifically for source code.

Our experiments are also performed over a relatively small dataset of 2 million functions, whereas models such as BERT [6] are trained over billions of words. As the largest dataset, `6L` in our experiments generally provided the best performance improvements, we hypothesise that an even bigger dataset to pre-train on would improve this further. However, these datasets need to be of acceptable quality, containing many functions with variables that have names which are semantically relevant to their method names in order to learn semantic similarity.

We have also only experimented on two tasks within the machine learning for programming languages domain. Another common task not explored in this work is generating natural language from source code and generating source code from natural language.

We have also only focused on improvement in quantitative metrics and would be interested to see how much these translate into qualitative improvements by the use of human graders on the outputs of the models for each task.

This work also only focuses on code represented as a sequence of tokens, and not as a tree or graph like in [24, 8, 9, 25]. These models, when compared against token based representation models, usually achieve a higher performance, and have shown to give improved results when pre-trained [26, 27]. However, different programming languages have their own unique abstract syntax tree structure.

VIII. Conclusions

We have shown that applying transfer learning in the programming language domain does provide performance improvements as it does to natural languages. This is true for the two tasks explored in this paper, code retrieval and method name prediction.

We show that a range of datasets can be used for transfer learning in this domain. The dataset can be made of data only in the downstream task language, a dataset containing multiple programming languages - even if it does not contain the downstream task language and when the dataset is solely made up of languages from a different domain, namely natural language. This is due to both tasks only needing to learn the semantic relationships tokens.

Acknowledgment

We would like to thank Ruben Kruiper for proof-reading.

References

[1] Tomas Mikolov et al. "Efficient Estimation of Word Representations in Vector Space". In: *1st International Conference on Learning Representations, ICLR 2013, Scottsdale, Arizona, USA, May 2-4, 2013, Workshop Track Proceedings*. 2013. URL: http://arxiv.org/abs/1301.3781.

[2] Tomas Mikolov et al. "Distributed Representations of Words and Phrases and their Compositionality". In: *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013. Proceedings of a meeting held December 5-8, 2013, Lake Tahoe, Nevada, United States*. 2013, pp. 3111–3119.

[3] Jeffrey Pennington, Richard Socher, and Christopher D. Manning. "Glove: Global Vectors for Word Representation". In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. 2014, pp. 1532–1543.

[4] Jeremy Howard and Sebastian Ruder. "Universal Language Model Fine-tuning for Text Classification". In: *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics, ACL 2018, Melbourne, Australia, July 15-20, 2018, Volume 1: Long Papers*. 2018, pp. 328–339. DOI: 10.18653/v1/P18-1031.

[5] Matthew E. Peters et al. "Deep Contextualized Word Representations". In: *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*. 2018, pp. 2227–2237.

[6] Jacob Devlin et al. "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, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*. 2019, pp. 4171–4186.

[7] Miltiadis Allamanis, Hao Peng, and Charles A. Sutton. "A Convolutional Attention Network for Extreme Summarization of Source Code". In: *Proceedings of the 33nd International Conference on Machine Learning, ICML 2016, New York City, NY, USA, June 19-24, 2016*. 2016, pp. 2091–2100.

[8] Uri Alon et al. "code2vec: learning distributed representations of code". In: *PACMPL* 3.POPL (2019), 40:1–40:29. DOI: 10.1145/3290353.

[9] Uri Alon et al. "code2seq: Generating Sequences from Structured Representations of Code". In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[10] Hamel Husain et al. "CodeSearchNet Challenge: Evaluating the State of Semantic Code Search". In: *CoRR* abs/1909.09436 (2019). arXiv: 1909.09436. URL: http://arxiv.org/abs/1909.09436.

[11] Miltiadis Allamanis et al. "A Survey of Machine Learning for Big Code and Naturalness". In: *ACM Comput. Surv.* 51.4 (2018), 81:1–81:37. DOI: 10.1145/3212695.

[12] Srinivasan Iyer et al. “Mapping Language to Code in Programmatic Context”. In: *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, Brussels, Belgium, October 31 - November 4, 2018*. 2018, pp. 1643–1652.

[13] Patrick Fernandes, Miltiadis Allamanis, and Marc Brockschmidt. “Structured Neural Summarization”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[14] Johannes Bader et al. “Getafix: learning to fix bugs automatically”. In: *PACMPL* 3.OOPSLA (2019), 159:1–159:27. DOI: 10.1145/3360585.

[15] Zimin Chen and Martin Monperrus. “A Literature Study of Embeddings on Source Code”. In: *CoRR* abs/1904.03061 (2019). arXiv: 1904.03061. URL: http://arxiv.org/abs/1904.03061.

[16] Michael Pradel Yaza Wainakh Moiz Rauf. “Evaluating Semantic Representations of Source Code”. In: *CoRR* abs/1910.05177 (2019). arXiv: 1910.05177. URL: http://arxiv.org/abs/1910.05177.

[17] Rabee Sohail Malik, Jibesh Patra, and Michael Pradel. “NL2Type: Inferring JavaScript Function Types from Natural Language Information”. In: *Proceedings of the 41st International Conference on Software Engineering*. ICSE ’19. Montreal, Quebec, Canada: IEEE Press, 2019, pp. 304–315. DOI: 10.1109/ICSE.2019.00045.

[18] Michael Pradel and Koushik Sen. “DeepBugs: a learning approach to name-based bug detection”. In: *PACMPL* 2.OOPSLA (2018), 147:1–147:25. DOI: 10.1145/3276517.

[19] Zhangyin Feng et al. *CodeBERT: A Pre-Trained Model for Programming and Natural Languages*. 2020. arXiv: 2002.08155 [cs.CL].

[20] Ashish Vaswani et al. “Attention is All you Need”. In: *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, 4-9 December 2017, Long Beach, CA, USA*. 2017, pp. 5998–6008.

[21] Kyunghyun Cho et al. “Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation”. In: *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP 2014, October 25-29, 2014, Doha, Qatar, A meeting of SIGDAT, a Special Interest Group of the ACL*. 2014, pp. 1724–1734.

[22] Stephen Merity et al. “Pointer Sentinel Mixture Models”. In: *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. 2017.

[23] Abram Hindle et al. “On the Naturalness of Software”. In: *Proceedings of the 34th International Conference on Software Engineering*. ICSE ’12. Zurich, Switzerland: IEEE Press, 2012, pp. 837–847. ISBN: 9781467310673.

[24] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. “Learning to Represent Programs with Graphs”. In: *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. 2018.

[25] Marc Brockschmidt et al. “Generative Code Modeling with Graphs”. In: *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. 2019.

[26] Weihua Hu et al. “Pre-training Graph Neural Networks”. In: *CoRR* abs/1905.12265 (2019). arXiv: 1905.12265. URL: http://arxiv.org/abs/1905.12265.

[27] Ziniu Hu et al. “Pre-Training Graph Neural Networks for Generic Structural Feature Extraction”. In: *CoRR* abs/1905.13728 (2019). arXiv: 1905.13728. URL: http://arxiv.org/abs/1905.13728.