NaturalCC: An Open-Source Toolkit for Code Intelligence

Yao Wan[1*], Yang He[2], Zhangqian Bi[1], Jianguo Zhang[3], Yulei Sui[2], Hongyu Zhang[4], Kazuma Hashimoto[5], Hai Jin[1], Guandong Xu[2], Caiming Xiong[6], Philip S. Yu[3]

[1]National Engineering Research Center for Big Data Technology and System, Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, Wuhan, China

[2] University of Technology Sydney, Australia [3] University of Illinois at Chicago, USA

[4] University of Newcastle, Australia [5]Google Research, USA [6]Salesforce Research, USA

ABSTRACT

We present NaturalCC, an efficient and extensible open-source toolkit for machine-learning-based source code analysis (i.e., code intelligence). Using NaturalCC, researchers can conduct rapid prototyping, reproduce state-of-the-art models, and/or exercise their own algorithms. NaturalCC is built upon Fairseq and PyTorch, providing (1) a collection of code corpus with preprocessing scripts, (2) a modular and extensible framework that makes it easy to reproduce and implement a code intelligence model, and (3) a benchmark of state-of-the-art models. Furthermore, we demonstrate the usability of our toolkit over a variety of tasks (e.g., code summarization, code retrieval, and code completion) through a graphical user interface. The website of this project is http://xcodemind.github.io, where the source code and demonstration video can be found.

CCS CONCEPTS

• **Software and its engineering** → **Reusability**.

KEYWORDS

Code intelligence, deep learning, code representation, code embedding, open source, toolkit, benchmark

ACM Reference Format:

Yao Wan[1*], Yang He[2], Zhangqian Bi[1], Jianguo Zhang[3], Yulei Sui[2], Hongyu Zhang[4], Kazuma Hashimoto[5], Hai Jin[1], Guandong Xu[2], Caiming Xiong[6], Philip S. Yu[3] . 2022. NaturalCC: An Open-Source Toolkit for Code Intelligence. In *44th International Conference on Software Engineering Companion (ICSE '22 Companion), May 21–29, 2022, Pittsburgh, PA, USA.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3510454.3516863

1 INTRODUCTION

Code intelligence is about applying machine learning, including deep learning techniques to analyze the big corpora of source code collected from open-source platforms (e.g., GitHub and StackOverflow). In recent years, many code intelligence approaches have been proposed for automating various programming tasks, such as code summarization [3, 25–27], code retrieval [7, 8, 24], and code completion [15], with the aim of improving developer productivity.

However, there still exist several limitations that hinder the development of machine learning-based source code analysis. There are two aspects to be investigated in this work. (a) *Lack of standardized algorithm implementation and toolkit for reproducing the results of existing methods*: nowadays deep learning methods are widely used, but they are not always easily reproducible due to their sensitivity to data and algorithm implementations; therefore, it is beneficial to build a toolkit with different algorithms integrated within a unified framework. (b) *Lack of benchmarks for fair comparisons between models*: for a given task, a research paper usually declares that a performance gain has been achieved; it is important to build a benchmarking framework to understand whether the performance gain is from the model design itself, hyperparameter tuning, or unfair settings.

There exist many established toolkits such as Fairseq [19], AllenNLP [6], and Stanza [20] in the area of *natural language processing* (NLP), but it is difficult to directly apply them to analyze source code written in programming languages. In particular, Fairseq was originally designed for modeling sequence-to-sequence tasks for natural languages (e.g., neural machine translation and language model pre-training). On the other hand, AllenNLP and Stanza are designed to model various kinds of NLP tasks. In these toolkits, the input is usually plain natural language text. When adapting these toolkits to programming languages, the biggest challenge is to incorporate the structural properties of source code such as AST (*abstract syntax tree*) and CFG (*control-flow graph*). In addition, the nature of source code-related tasks is often different from that of NLP tasks. Notable contemporary work is CodeXGLUE [18], which aims to build a benchmark dataset for code understanding and generation based on CodeBERT [5] and GraphCodeBERT [9]. Unlike their work, we focus more on building the infrastructures of various model implementations and enabling users to conduct rapid prototyping. In addition, our NaturalCC also integrates plentiful compiler tools and scripts for data preprocessing.

In this paper, we propose NaturalCC (stands for Natural Code Comprehension), a comprehensive platform for analyzing source code corpora to achieve code intelligence using machine learning techniques. We demonstrate NaturalCC with a graphical user interface, using three application tasks, i.e., code summarization, code retrieval, and code completion. We believe researchers from software engineering or other communities can be benefited from the toolkit for fast model prototyping and reproduction. We also

*Correspondence to Yao Wan (wanyao@hust.edu.cn).

ICSE '22 Companion, May 21–29, 2022, Pittsburgh, PA, USA

ACM ISBN 978-1-4503-9223-5/22/05…$15.00
https://doi.org/10.1145/3510454.3516863

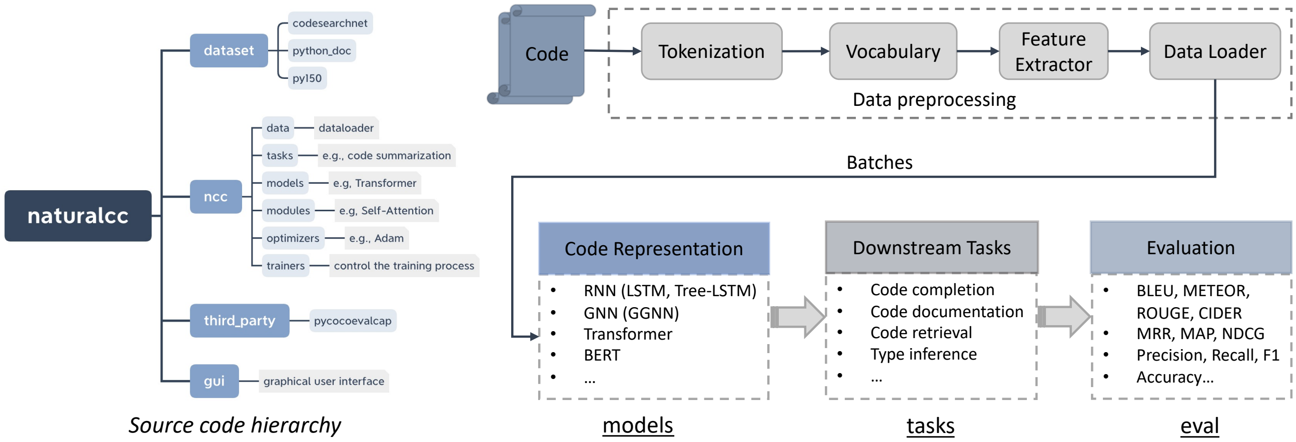

Figure 1: The pipeline of NaturalCC.

encourage researchers to integrate their state-of-the-art approaches into NaturalCC, to promote the research in both communities.

In summary, NaturalCC features the following contributions.

- **A collection of code corpus with preprocessing scripts.** We have cleaned and preprocessed three public datasets (i.e., CodeSearchNet [11], Python-Doc [25], and Py150 [22]). We provide data preprocessing scripts for extracting multiple code features, using compiler tools such as LLVM [17].
- **An extensible framework.** Based on the registry mechanism implemented in Fairseq [19], our framework is well modularized and can be easily extended to various tasks. In particular, when implementing a new task, users only need to implement models by instantiating one of our templates and then register them.
- **Performance benchmark.** We have benchmarked three downstream tasks (code summarization, code retrieval, and code completion) over three datasets using NaturalCC, achieving state-of-the-art or competitive performance.

2 DESIGN AND IMPLEMENTATION

Figure 1 shows the pipeline of our NaturalCC. Given a dataset of code snippets, we first preprocess the data and then feed each mini-batch of samples into the code representation module, which is a fundamental component for several downstream tasks. In the code representation module, we have implemented many state-of-the-art encoders (e.g., RNN, GNN, Transformer, and BERT). Based on the code representation, NaturalCC supports various downstream tasks, e.g., code summarization, code retrieval, and code completion.

2.1 Dataset and Data Preprocessing

We have collected three related datasets which have been widely adopted in the evaluation of different tasks. CodeSearchNet [11] is a public dataset of 6,452,446 source code snippets from GitHub, written in six programming languages, ranging from Java, Python, PHP, Javascript, Go, to Ruby. In this dataset, nearly 32% code snippets are with description, while others are not. This dataset has been widely used for the evaluation of source code retrieval and code summarization. Python-Doc [25] is a dataset of parallel Python code snippets with corresponding descriptions, which has been widely adopted for code summarization. Py150 [22] is a collection of 150*k* Python source code files, which has widely used for evaluating code completion.

In the data preprocessing stage, we first tokenize the source code by a tokenizer (e.g., space tokenizer or BPE [14] tokenizer) and then build a vocabulary for these tokens. In addition to code tokens, we also extract some domain-specific features such as AST, intermediate representation, control-flow graphs, or data-flow graphs. The goal of this process is to build a series of mini-batches for training. We put all data-related processing scripts in the `data` and `dataset` folders.

Code Token. Like tokenizing natural languages, we support tokenizing source code in different granularities, including character-level, word-level, and sub-word level (e.g., BPE). We split each word by character, space, or camel word. We use the `sentencepiece` module [16] for sub-word level tokenization.

Intermediate Representation (IR). *Intermediate Representation* (IR), formalized as *three-address code*, is a data structure used internally by a compiler when translating source code into low-level machine code. IR is independent on programming languages and machines, and has a much smaller vocabulary than that are built from lexical token modality. Therefore, it has a great potential for representing multi-lingual programming languages. In this paper, we adopt the IR generated by LLVM.

Abstract Syntax Tree (AST). *Abstract Syntax Tree* (AST) represents the abstract syntactic structure of source code in tree-format. We extract ASTs of code by using the `tree-sitter`[1] parser, and store them in JSON format.

Code Graph Building. To capture the structural properties of source code, we build the flow graphs, including control-flow graph, data-flow graph, and call graph using LLVM Clang[2], and store them in the Google `protocol buffer`[3] format.

[1]https://tree-sitter.github.io/tree-sitter
[2]https://clang.llvm.org
[3]https://github.com/protocolbuffers/protobuf

Table 1: A summary of state-of-the-art models designed for different code-related tasks, and the datasets for evaluation.

Task	Dataset	Model
Code Summarization	Python-Doc	Seq2Seq [12], Transformer [1], PLBART [2]
Code Retrieval	CodeSearchNet	NBow, Conv1D, Bi-RNN, SelfAttn [11]
Code Completion	Py150	LSTM [10], GPT-2, TravTrans [15]

2.2 Code Representation

Code representation, which aims to learn an embedding vector, is one of the most critical components for big code analysis. In NaturalCC, we have included most state-of-the-art neural network encoders to represent the source code and their extracted features. For example, we have implemented RNN-based models to represent the sequential tokens or (linearized) AST of code. We implement *graph neural networks* (GNNs) such as *gated graph neural networks* (GGNNs) to represent the graph structure features of code (e.g., control-flow and data-flow graphs). We have also included the Transformer network, which serves as the replacement of the RNN network, with its fast computation and ability to handle long-range dependent sequence. In addition, NaturalCC also supports the masked pre-trained models, e.g., BERT, RoBERTa, and BART. We put all the code representation networks in the `models` and `modules` folders.

Code Pre-training. As the pre-training technology (i.e., BERT and GPT) has achieved great success in representation learning, recently there have been several efforts (e.g., CuBERT [13], CodeBERT, PLBART [2], and GraphCodeBERT) in pre-training a BERT or GPT for source code. In NaturalCC, we have also integrated the pre-training techniques. For example, we have included PLBART [2] for code summarization and GPT-2 for code completion.

2.3 Tool Implementation

The source code structure of NaturalCC is shown in Figure 1. The `dataset` folder is for data preprocessing. The `ncc` folder is the core module. The `third_party` folder contains packages for model evaluation. The `gui` folder is for graphical user interface. We implement NaturalCC based on Fairseq and PyTorch. By adopting the outstanding registry mechanism designed in Fairseq, NaturalCC also has good extensibility with a modular design.

Registry Mechanism. We have implemented a `register` decorator in the entry to build a task, model or module (cf. `__init__.py` in each folder). In brief, the registry mechanism is to design a global variable to store each task of model objects for the off-the-shelf fetching. This registry mechanism is easy for extension and rapid prototyping, as we only need to include this decorator when defining a new task/model/module in the corresponding function. Therefore, we can integrate new tasks or datasets, such as CodeXGLUE [18].

Efficient Training. Following Fairseq, we use the `NCCL` library and `torch.distributed` to support model training on multiple GPUs. Every GPU stores a copy of model parameters, and the global optimizer functions as synchronous optimization in each GPU. Furthermore, NaturalCC can also support both full precision (FP32) and half-precision floating point (FP16) for fast training and inference. To preserve model accuracy, the parameters are stored in FP32 while updated by FP16 gradients.

Flexible Configuration. Unlike using `argparse` for command-line options in Fairseq, we propose to create a `yaml` file as configurations for each model and its variants. We believe it is more flexible to modify the yaml configuration files for model explorations.

Table 2: Performance of code summarization on Python-Doc.

	BLEU	METEOR	ROUGE-L	Cost
Seq2Seq+Attn	25.57	14.40	39.41	0.09s/Batch
Tree2Seq+Attn	23.35	12.59	36.49	0.48s/Batch
Transformer	30.64	17.65	44.59	0.26s/Batch
PLBART	**32.71**	**18.13**	**46.05**	0.26s/Batch

3 PERFORMANCE BENCHMARK

NaturalCC currently supports three downstream tasks, code summarization, code retrieval, and code completion, to showcase the effectiveness of the proposed framework. The implementations of the tasks in this toolkit can serve as baselines for fair comparisons in future research work. Table 1 gives a summary of the state-of-the-art models designed for the targeted source code-related tasks.

Note that we have carefully implemented and verified all the models to ensure the performances are on par with the original papers. We have also built a leaderboard so that users can provide their performance results for model competition.

3.1 Code Summarization

Summarizing code snippets into natural language descriptions is an effective way for understanding source code. We provide implementation of several representative models of code summarization, including Seq2Seq [12], Tree2Seq [4], Transformer [1], and PLBART [2]. For the Seq2Seq model, we tokenize each code snippet by white space and build a vocabulary of size 50K. For the Transformer models, we use BPE to get the sub-word vocabulary of size 50K. Both models are trained using four V100 GPUs with a learning rate of $1e$-4 and a batch size of 64. We pretrain a BART model for source code, named PLBART [2]. We first perform the pretraining on CodeSearchNet for 50,000 iterations, and then fine-tune it on the Python-Doc dataset [25]. We evaluate each model on the Python-Doc dataset using the BLEU, METEOR, and ROUGE metrics. The performance of different models implemented in NaturalCC is summarized in Table 2. We also record the computational cost (time cost per batch) for training each model.

3.2 Code Retrieval

Searching relevant code snippets given a natural language query can help developers with code reuse. We used the CodeSearchNet dataset [11] along with the MRR evaluation metric, and have implemented its four baseline models in [11], including NBOW, Conv1D, BiRNN, and SelfAttn. We tokenize each code snippet by BPE and build a sub-word vocabulary of size 10K. Both models are trained on a single RTX 6000 GPU with a learning rate of $1e$-2 and a batch size

Table 3: MRR of code retrieval on CodeSearchNet.

	Go	Java	JavaScript	PHP	Python	Ruby	Cost
NBOW	66.59	59.92	47.15	54.75	63.33	42.86	0.16s/Batch
Conv1D	70.87	60.49	38.81	61.92	67.29	36.53	0.30s/Batch
BiRNN	65.80	48.60	23.23	51.36	48.28	19.35	0.74s/Batch
SelfAttn	**78.45**	**66.55**	**50.38**	**65.78**	**79.09**	**47.96**	0.25s/Batch

Table 4: MRR@10 of code completion on Py150.

	Attr.	Num.	Identifier	Param.	All Tokens	Cost
LSTM	51.67	47.45	46.52	66.06	73.73	0.31s/Batch
GPT-2	70.37	62.20	63.84	**73.54**	82.17	0.43s/Batch
TravTrans	**72.08**	**68.55**	**76.33**	71.08	**83.17**	0.43s/Batch

of 1,000. The performance of the evaluated models is summarized in Table 3.

3.3 Code Completion

Code completion, which provides the developers a shortlist of probable code candidates according to the current information, is a primary feature of most modern IDEs. We have implemented the LSTM [23], Transformer-based GPT-2 [21], and TravTrans [15] models for reference. We evaluate LSTM and GPT-2 on next token prediction, and TravTrans on next leaf token prediction. We categorize the prediction tokens into five classes, namely attributes (Attr.), numeric constant (Num.), identifier name (Identifier), function parameter name (Param.) and all tokens, according to their annotated function from AST. We tokenize each code snippet by white space and build a vocabulary of size 50K. Both models are trained using four V100 GPUs with an effective batch size of 128. We evaluate each model on Py150 dataset using the MRR@10 metric. The performance of the evaluated models is summarized in Table 4.

4 TOOL USAGE

In this section, we show how to explore NaturalCC through a proof-of-concept example, as well as a graphical user interface.

4.1 A Proof-of-Concept Example

We take code completion as an example to show the pipeline of how to implement a new task in NaturalCC quickly.

```
@register_task('completion')
class CompletionTask(NccTask):
    @classmethod
    def setup_task(cls, args, **kwargs):
        dictionary = cls.load_dictionary(args)
        return cls(args, dictionary)
    def build_model(self, args):
        model = super().build_model(args)
        return model
```

Listing 1: `tasks/completion/completion.py`

Building a Task. In the first step, we create a `CompletionTask` in the `ncc/tasks/completion.py`, with a decorator `register_task` around. Listing 1 shows the whole processing of building a new task. This class provides a function `build_model` for building a model according to the arguments defined by users.

```
@register_model('seqrnn')
class SeqRNNModel(NccLanguageModel):
    @classmethod
    def build_model(cls, args, config, task):
        decoder = LSTMDecoder(...)
        return cls(args, decoder)
```

Listing 2: `models/completion/seqrnn.py`

Building a Model. Listing 2 shows the process of building a RNN model for code completion. We define a new class `SeqRNNModel` in the `ncc/models/completion/seqrnn.py`, which inherits the `NccLanguageModel`. In this class, we build a decoder neural network `LSTMDecoder`, which is implemented in the `modules` folder.

```
# 1. Setup task, e.g., completion, comment generation, etc.
task = tasks.setup_task(args)
# 2. Build model and criterion
model = task.build_model(args)
criterion = task.build_criterion(args)
# 3. Build trainer
trainer = Trainer(args, task, model, criterion)
while (
    lr > args['optimization']['min_lr']
    and epoch_itr.next_epoch_idx <= max_epoch
    and trainer.get_num_updates() < max_update
):
    task.train_step(samples)
```

Listing 3: `trainer/ncc_trainer.py`

Model Training. We have designed a trainer (`ncc_trainer.py`) module to control the whole training process of models. Listing 3 shows the construction of a `Trainer` object and the training steps. Core parameters are stored in this process such that pre-trained models can be precisely restored during inference or fine-tuning.

4.2 Graphical User Interface

We have also provided a graphical user interface for users to easily access and explore the results of each trained model through a Web browser. The design of our website is based on the open-source demo of AllenNLP [6]. We have deployed it on the Nginx server and provided flexible APIs via the Flask engine.[4]

As shown in Figure 2, we have integrated three popular software engineering tasks for demonstration, i.e., code summarization, code retrieval, and code completion. Taking code summarization as an example, by default, we have implemented this task based on the Transformer. Given a code snippet of Python, when clicking the *Run* button, a user-selected trained model will be invoked for inference and the generated summary will be displayed at the bottom of the webpage.

5 CONCLUSION

This paper presents NaturalCC, an efficient and extensible open-source toolkit for machine-learning-based source code analysis (i.e, code intelligence). Currently, NaturalCC has implemented many state-of-the-art models for three popular source code-related tasks, which can serve as benchmarks for fair comparisons. Other researchers can extend our framework to implement new models or support new tasks. We have also provided a Web-based graphical

[4]https://flask.palletsprojects.com

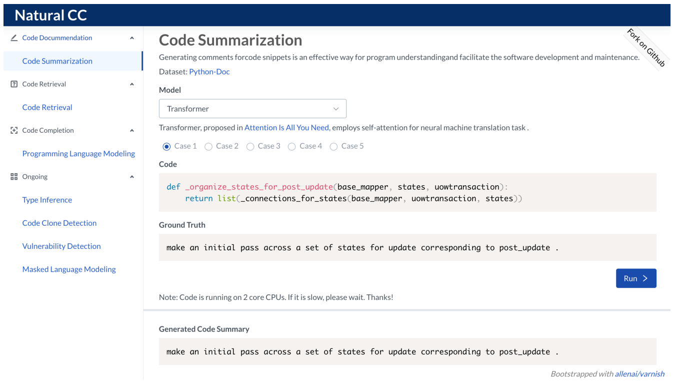

Figure 2: A screenshot of the graphical user interface of NaturalCC for demonstration.

user interface for users to explore the results. In our future work, more state-of-the-art models in code intelligence tasks will be integrated, such as code clone detection, program translation, and vulnerability detection. We will also support automatic evaluation of the submitted models.

Artifacts and Resources. All the source code and materials are publicly available at http://github.com/CGCL-codes/naturalcc.[5] Our project webpage is http://xcodemind.github.io, where the demonstration video can be found. NaturalCC is still under development. We encourage researchers and developers to join us to further promote the development of NaturalCC.

ACKNOWLEDGMENTS

This work is supported by National Natural Science Foundation of China under grand No. 62102157. This work is also partially sponsored by Tencent Rhino-Bird Focus Research Program of Basic Platform Technology. We would like to thank all the anonymous reviewers for their constructive comments on improving this paper.

REFERENCES

[1] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A Transformer-based Approach for Source Code Summarization. In *Proceedings of the 58th Annual Meeting of the Association for Computational Linguistics*. 4998–5007.

[2] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. 2655–2668.

[3] Uri Alon, Shaked Brody, Omer Levy, and Eran Yahav. 2019. code2seq: Generating Sequences from Structured Representations of Code. In *7th International Conference on Learning Representations*.

[4] Akiko Eriguchi, Kazuma Hashimoto, and Yoshimasa Tsuruoka. 2016. Tree-to-Sequence Attentional Neural Machine Translation. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

[5] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Proceedings of Findings of the Association for Computational Linguistics: EMNLP 2020*. 1536–1547.

[6] Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew E. Peters, Michael Schmitz, and Luke Zettlemoyer. 2018. AllenNLP: A Deep Semantic Natural Language Processing Platform. *CoRR* abs/1803.07640 (2018).

[5]The open-source Fairseq toolkit has inspired us a lot, and our open-source project also follows the MIT license.

[7] Xiaodong Gu, Hongyu Zhang, and Sunghun Kim. 2018. Deep code search. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 933–944.

[8] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. 2016. Deep API learning. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (Seattle, WA, USA). ACM, 631–642.

[9] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2021. GraphCodeBERT: Pre-training Code Representations with Data Flow. In *Proceedings of 9th International Conference on Learning Representations*.

[10] Vincent J. Hellendoorn and Premkumar T. 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*. ACM, 763–773.

[11] Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet Challenge: Evaluating the State of Semantic Code Search. *CoRR* abs/1909.09436 (2019).

[12] Srinivasan Iyer, Ioannis Konstas, Alvin Cheung, and Luke Zettlemoyer. 2016. Summarizing Source Code using a Neural Attention Model. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics*.

[13] Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning*, Vol. 119. 5110–5121.

[14] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code != big vocabulary: open-vocabulary models for source code. In *Proceedings of the 42nd International Conference on Software Engineering*. ACM, 1073–1085.

[15] Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2021. Code Prediction by Feeding Trees to Transformers. In *Proceedings of the 43rd IEEE/ACM International Conference on Software Engineering*. IEEE, 150–162.

[16] Taku Kudo and John Richardson. 2018. SentencePiece: A simple and language independent subword tokenizer and detokenizer for Neural Text Processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*. 66–71.

[17] Chris Lattner and Vikram S. Adve. 2004. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *2nd IEEE / ACM International Symposium on Code Generation and Optimization*. IEEE Computer Society, 75–88.

[18] Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, Ming Gong, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie Liu. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. *CoRR* abs/2102.04664 (2021).

[19] Myle Ott, Sergey Edunov, Alexei Baevski, Angela Fan, Sam Gross, Nathan Ng, David Grangier, and Michael Auli. 2019. fairseq: A Fast, Extensible Toolkit for Sequence Modeling. In *Proceedings of NAACL-HLT 2019: Demonstrations*.

[20] Peng Qi, Yuhao Zhang, Yuhui Zhang, Jason Bolton, and Christopher D Manning. 2020. Stanza: A Python natural language processing toolkit for many human languages. *arXiv preprint arXiv:2003.07082* (2020).

[21] Alec Radford, Jeffrey Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language models are unsupervised multitask learners. *OpenAI blog* 1, 8 (2019), 9.

[22] Veselin Raychev, Pavol Bielik, and Martin T. Vechev. 2016. Probabilistic model for code with decision trees. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. ACM, 731–747.

[23] Veselin Raychev, Martin T. 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*. ACM, 419–428.

[24] Yao Wan, Jingdong Shu, Yulei Sui, Guandong Xu, Zhou Zhao, Jian Wu, and Philip S. Yu. 2019. Multi-modal Attention Network Learning for Semantic Source Code Retrieval. In *Proceedings of 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE, 13–25.

[25] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S. Yu. 2018. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 397–407.

[26] Wenhua Wang, Yuqun Zhang, Yulei Sui, Yao Wan, Zhou Zhao, Jian Wu, Philip S. Yu, and Guandong Xu. 2022. Reinforcement-Learning-Guided Source Code Summarization Using Hierarchical Attention. *IEEE Transactions on Software Engineering* 48, 1 (2022), 102–119.

[27] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, and Xudong Liu. 2020. Retrieval-based neural source code summarization. In *Proceedings of the 42nd International Conference on Software Engineering* (Seoul, South Korea). ACM, 1385–1397.