NATURALCC: A Toolkit to Naturalize the Source Code Corpora

Yao Wan[1] **Yang He**[2] **Jian-Guo Zhang**[3] **Yulei Sui**[2] **Kazuma Hashimoto**[4]
Hai Jin[1] **Guandong Xu**[2] **Philip S. Yu**[3] **Caiming Xiong**[4]
[1]Services Computing Technology and System Lab, Cluster and Grid Computing Lab, School of Computer Science and Technology, Huazhong University of Science and Technology, China
[2] School of Computer Science, University of Technology Sydney, NSW, Australia
[3] Department of Computer Science, University of Illinois at Chicago, Illinois, USA
[4]Salesforce Research, Palo Alto, USA
{wanyao,haij}@hust.edu.cn {yulei.sui,guandong.xu}@uts.edu.au
{jzhan51,psyu}@uic.edu cxiong@salesforce.com

Abstract

We present NATURALCC, an efficient and extensible open-source toolkit to bridge the gap between natural language processing and programming language research to perform machine learning-based code analysis. Using NATURALCC, researchers from both the research communities can conduct rapid prototyping, reproduce state-of-the-art baselines, and/or exercise their own algorithms. NATURALCC is built upon Fairseq and PyTorch, providing (1) a collection of code corpora with preprocessing scripts, (2) a modular and extensible framework that makes it easy to reproduce and implement an approach for large-scale code analysis, and (3) a benchmark of state-of-the-art models. Furthermore, we demonstrate usability of our toolkit over a variety of tasks (e.g., code summarization, code retrieval, code completion, and type inference) through a command line interface as well as a graphical user interface.

1 Introduction

Benefiting from the naturalness hypothesis shared among natural languages and programming languages, recent years have witnessed a major resurgence of applying natural language processing (NLP) techniques to modeling the big corpora of source codes from open-source platforms (e.g., GitHub[1] and StackOverflow[2]). It is widely studied to conduct machine learning-based code analysis for automate programming tasks, such as code summarization (Wan et al., 2018; Alon et al., 2018), code retrieval (Gu et al., 2018; Wan et al., 2019), code completion (Kim et al., 2020), and type inference (Hellendoorn et al., 2018), so as to facilitate programming for developers.

Despite a few efforts having been made recently, there still exists several limitations hindering the development of automated programming using NLP techniques. There are two aspects to be investigated in this work. *(a) Lack of standardized algorithm implementation and toolkit to reproduce results of existing methods*: deep learning methods are widely used, but they are not always easily reproducible due to their sensitivity to data and model architectures; 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 or through tuning parameters under various or unfair settings.

There exist many established toolkits such as Fairseq (Ott et al., 2019), AllenNLP (Gardner et al., 2017), Stanza (Qi et al., 2020) in the area of NLP, but it is not trivial how to directly apply them to analyzing 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), and AllenNLP and Stanza are designed to model various kinds of NLP tasks. In these toolkits, the input is usually expected to be plain natural language text. When adapting these toolkits to programming languages, the biggest challenge is that a source code program is always represented as structured data, such as tree and graph. In addition, the nature of code-related tasks is often different from that of NLP tasks. Notable contemporary work is CodeXGLUE (Lu et al., 2021), which aims to build a benchmark dataset for code understanding and generation, based on CodeBERT (Feng et al., 2020) and GraphCode-

[1]https://github.com
[2]https://stackoverflow.com

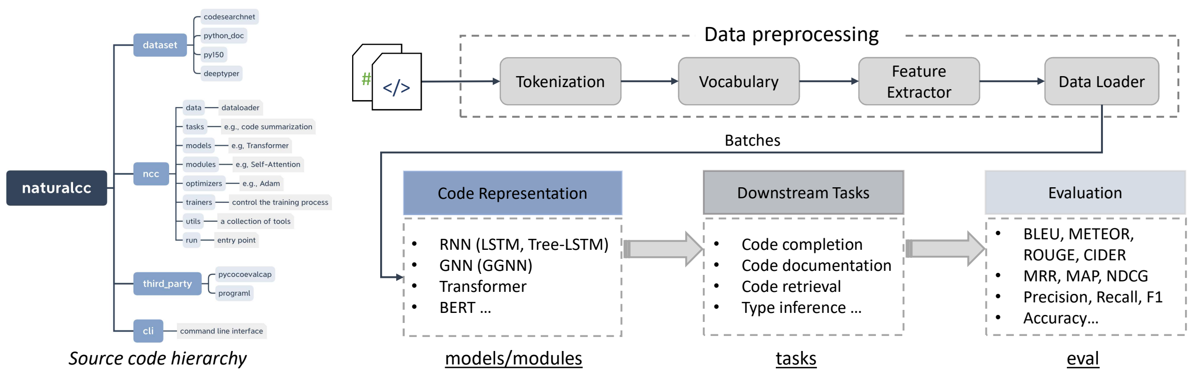


Figure 1: A pipeline of NATURALCC.

BERT (Guo et al., 2020). The major difference is that we focus more on building the infrastructures of various model implementations and enabling users to conduct rapid prototyping.

To mitigate the aforementioned issues, we have developed NATURALCC,[3] a comprehensive platform for modeling programming languages using NLP techniques for code corpora. The researchers from software engineering or NLP communities can benefit from the toolkit for fast model training and reproduction. Overall, NATURALCC features the following contributions:

- **A Collection of Code Corpora with Preprocessing Scripts.** We have cleaned and preprocessed four public datasets (i.e., CodeSearchNet (Husain et al., 2019), PythonDoc (Barone and Sennrich, 2017; Wan et al., 2018), Py150 (Raychev et al., 2016), and DeepTyper (Hellendoorn and Devanbu, 2017)) for four code-related tasks including code summarization, code retrieval, code completion and type inference. We provide data preprocessing scripts for extracting code features in multiple modalities based on LLVM (Lattner and Adve, 2004).

- **Extensibility and Modularity.** Based on the registry mechanism implemented in Fairseq (Ott et al., 2019), our framework is well modularized and can be easily extended to various tasks. In particular, when implementing a new task, we only need to implement models by instantiating one of our templates and then register them.

- **State-of-the-Art Performance.** We have benchmarked four downstream tasks over three datasets using NATURALCC, achieving state-of-the-art or competitive performance. This shows reliability of NATURALCC.

We demonstrate NATURALCC with a command line interface as well as a graphical user interface, using three application tasks, i.e., code completion, code comment generation, and code retrieval. NATURALCC is an ongoing open-source toolkit maintained by the *CodeMind* team. We hope NATURALCC can bridge the research gap between programming language and natural language. We also encourage researchers to integrate their state-of-the-art approach into NATURALCC, to promote the research in both communities.

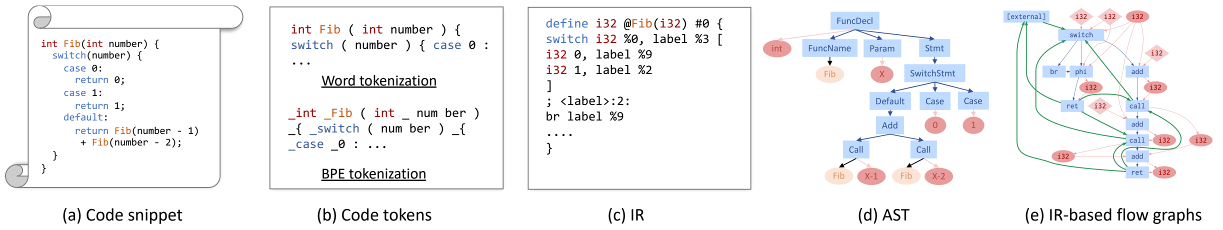
2 Design and Implementation

Figure 1 shows a pipeline of our NATURALCC. Given a dataset of code snippets, we first preprocess the data in the data preprocessing stage and then feed each mini-batch of samples into the code representation module, a fundamental component for several downstream tasks. In the code representation module, we have implemented many state-of-the-art encoders (e.g., RNN (Hochreiter and Schmidhuber, 1997), GNN (Wu et al., 2020), Transformer (Vaswani et al., 2017) and BERT (Devlin et al., 2018)). Based on the code representation, NATURALCC can also support various downstream tasks, e.g., code summarization, code retrieval, code completion and type inference. The designed `Trainer` controls model training for each task.

2.1 Dataset and Data Preprocessing

We have collected four related datasets which have been widely adopted in the evaluation of different tasks. CodeSearchNet (Husain et al., 2019)

[3]The term NaturalCC also represents natural code comprehension, which is a fundamental task that lies in the synergy between the programming languages and NLP.



```
int Fib(int number) {
  switch(number) {
    case 0:
      return 0;
    case 1:
      return 1;
    default:
      return Fib(number - 1)
       + Fib(number - 2);
  }
}
```

(a) Code snippet

```
int Fib ( int number ) {
switch ( number ) { case 0 :
...
```

Word tokenization

```
_int _Fib ( int _ num ber )
_{ _switch ( num ber ) _{
_case _0 : ...
```

BPE tokenization

(b) Code tokens

```
define i32 @Fib(i32) #0 {
switch i32 %0, label %3 [
i32 0, label %9
i32 1, label %2
]
; <label>:2:
br label %9
....
}
```

(c) IR

Figure 2: Different modalities extracted from source code.

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. Among this dataset, nearly 32% are with description, while others are not. This dataset has been widely used for the evaluation of source code retrieval and code summarization. PythonDoc (Barone and Sennrich, 2017; Wan et al., 2018) is a dataset of parallel Python code snippets with corresponding descriptions, which has been widely adopted for code summarization. Py150 (Raychev et al., 2016) is a collection of 150k Python source code files, which has been widely used for evaluating code completion. DeepTyper (Hellendoorn et al., 2018) is a dataset of Typescript code snippets, which has been used for evaluating type inference.

In the data preprocessing stage, we first tokenize the source code by a tokenizer (e.g., space tokenizer or BPE (Karampatsis et al., 2020) tokenizer) and then build a vocabulary for these tokens. In addition to code tokens, we can also extract some domain-specific features (modalities) such as AST (Wan et al., 2018), intermediate representation (VenkataKeerthy et al., 2019), control-flow graphs (Wan et al., 2019), or data-flow graphs (Allamanis et al., 2017). 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 granularity, including character-level, word-level and sub-word level (e.g., BPE). Simply, we can split each word by character, space or camel word, and we use the `sentencepiece` module (Kudo and Richardson, 2018) 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. It is interesting to mention that the IR is independent to programming languages and machines, and has a much smaller vocabulary than that are built from lexical token modality. In this paper, we adopt the IR generated by LLVM (Lattner and Adve, 2004).

Abstract Syntax Tree (AST) The Abstract Syntax Tree (AST) represents the abstract syntactic structure of source code in tree-format. We extract ASTs of codes by using the `tree-sitter`[4] parser, and store them in JSON format.

Code Graph Building To model the control, data, and call dependencies of a program for better representation, we build the flow graphs, including control-flow graph, data-flow graph and call graph using LLVM Clang[5], and store them in the Google `protocol buffer`[6] format.

Figure 2 shows a detailed code snippet written in C, as well as its corresponding code tokens, IR, AST, IR-based control-, call- and data-flow graphs.

2.2 Code Representation

Code representation/understanding, 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 advanced Transformer networks (Vaswani et al., 2017), which serve 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

[4]https://tree-sitter.github.io/tree-sitter
[5]https://clang.llvm.org
[6]https://github.com/protocolbuffers/protobuf

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

Task	Dataset	Model
Code Summarization	Python-Doc	Seq2Seq (Iyer et al., 2016), Transformer (Ahmad et al., 2020), PLBART (Ahmad et al., 2021)
Code Retrieval	CodeSearchNet	NBow, Conv1D, Bi-RNN, SelfAttn (Husain et al., 2019)
Code Completion	Py150	LSTM (Hellendoorn and Devanbu, 2017), GPT-2 (Radford et al., 2019), TravTrans (Kim et al., 2020)
Type Inference	Typescript	DeepTyper (Hellendoorn et al., 2018), Transformer (Vaswani et al., 2017)

and RoBERTa (Liu et al., 2019). 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, and recently there have been several efforts (e.g., CuBERT (Kanade et al., 2020), CodeBERT, PLBART (Ahmad et al., 2021) and GraphCodeBERT) in pre-training a BERT or GPT for code corpus. In NATURALCC, we have also integrated the pre-training techniques. We have included PLBART for code summarization and GPT-2 for code completion.

2.3 Implementation

We have implemented NATURALCC based on the Fairseq and PyTorch. Following the outstanding registry mechanism designed in Fairseq, NATURALCC also has good extensibility with a modular design. To have a quick start, we put a proof-of-concept example in the supplementary materials.

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 (Lu et al., 2021).

Multi-GPU 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. Gradients accumulation is also supported multi-GPU computation.

Mixed-Precision NATURALCC can also support both full precision (FP32) and half-precision floating point (FP16) for fast training and inference. From our experience, setting the FP16 option can largely reduce the memory usage and further save the training time. 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 think that it is more flexible to modify the yaml configuration files for model explorations.

3 Performance Benchmark

NATURALCC currently supports four downstream tasks, code summarization, code retrieval, code completion, and type inference, to validate the effectiveness of the proposed framework. The implementations and the tasks in this toolkit will serve as baselines for fair comparison for future research use. Table 1 gives a summary of the state-of-the-art models designed for the targeted code-related tasks. Note that we have carefully implemented and verified all the models to ensure the performances are on par with the reference papers. Other tasks with state-of-the-art models are going to be integrated in future, including code clone detection (Hua et al., 2020), code translation (Lachaux et al., 2020), vulnerability detection (Li et al., 2018).

3.1 Code Summarization

Summarizing code snippets into natural language descriptions is an effective way for understanding codes and facilitating software development and maintenance. We provide implementation of several representative models of code summarization, including Seq2Seq (Iyer et al., 2016), Tree2Seq (Eriguchi et al., 2016), Transformer (Ahmad et al., 2020) and PLBART (Ahmad et al., 2021). 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

Table 2: Performance of our model and baseline methods for the task of code summarization over Python-Doc dataset. (Best scores are in boldface.)

	BLEU-4	METEOR	ROUGE-L	Cost
Seq2Seq+Attn	25.57	14.40	39.41	0.09s/B
Tree2Seq+Attn	23.35	12.59	36.49	0.48s/B
Transformer	30.64	17.65	44.59	0.26s/B
Transformer+RPE	31.57	17.74	45.18	0.27s/B
PLBART	**32.71**	**18.13**	**46.05**	0.80s/B

Table 3: MRR of our model and baseline methods for the task of code retrieval over CodeSearchNet dataset. (Best scores are in boldface.)

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

PLBART (Ahmad et al., 2021). We first perform the pretraining on CodeSearchNet for 50,000 iterations, and then fine-tune it on the Python-Doc dataset.

We evaluate each model on the Python-Doc (Barone and Sennrich, 2017; Wan et al., 2018) dataset, by using the BLEU, METEOR and ROUGE metrics as in Wan et al. (2018). The performance of different models implemented in NaturalCC are summarized in Table 2.

3.2 Code Retrieval

Searching semantically similar code snippets given a natural language query can provide developers a series of templates as references for rapid prototyping. We used the CodeSearchNet dataset (Husain et al., 2019) along with the MRR evaluation metric, and have implemented its four baseline models, NBOW, Conv1D, BiRNN, and SelfAttn. We tokenize each code snippet by BPE and build a subword vocabulary of size 10K. To boost the models' robustness, during training, 20% of function names will be randomly replaced with their corresponding natural language queries in docstrings, or 20% of the function names will be randomly removed in the code snippets. Both models are trained on a single RTX 6000 GPU with a learning rate of $1e\text{-}2$ and a batch size of 1,000. The model performance is summarized in Table 3.

3.3 Code Completion

Code completion has become an essential tool in many IDEs. It can boost developers' programming productivity; the task is to predict the next code element based on the previously written code We have implemented the LSTM (Raychev et al., 2014) and

Table 4: MRR of our model and baseline methods for the task of code completion over Py150 dataset. (Attr: attribute; Num: numeric constant; Name: variable, module; Func: function parameter name; Token: all tokens. Best scores are in boldface.)

	Accuracy					Cost
	Attr	Num	Name	Param	Token	
LSTM	51.67	47.45	46.52	66.06	73.73	0.31s/B
GPT-2	70.37	62.20	63.84	**73.54**	82.17	0.43s/B
TravTrans	**72.08**	**68.55**	**76.33**	71.08	**83.17**	0.43s/B

Table 5: Accuracy of our model and baseline methods for the task of type inference over Py150 dataset.

	Acc@1	Acc@5	Acc@1	Acc@5	Cost
	All types		Any types		
DeepTyper	**0.52**	**0.67**	**0.43**	0.67	0.42s/B
Transformer	0.34	0.64	0.37	**0.75**	0.85s/B

TravTrans (Kim et al., 2020) models for reference. We train the models using the Py150 dataset and evaluate them by the MRR metric. To evaluate the models' practical value in a real-world application, we measure performance of different key tokens, e.g., attributes and variables. We categorize those prediction tokens into 5 classes according to their annotated function from AST. We tokenize each code snippet by white space and build a vocabulary of size 50K. Both the models are trained using four V100 GPUs with an effective batch size of 128. The model performance is summarized in Table 4.

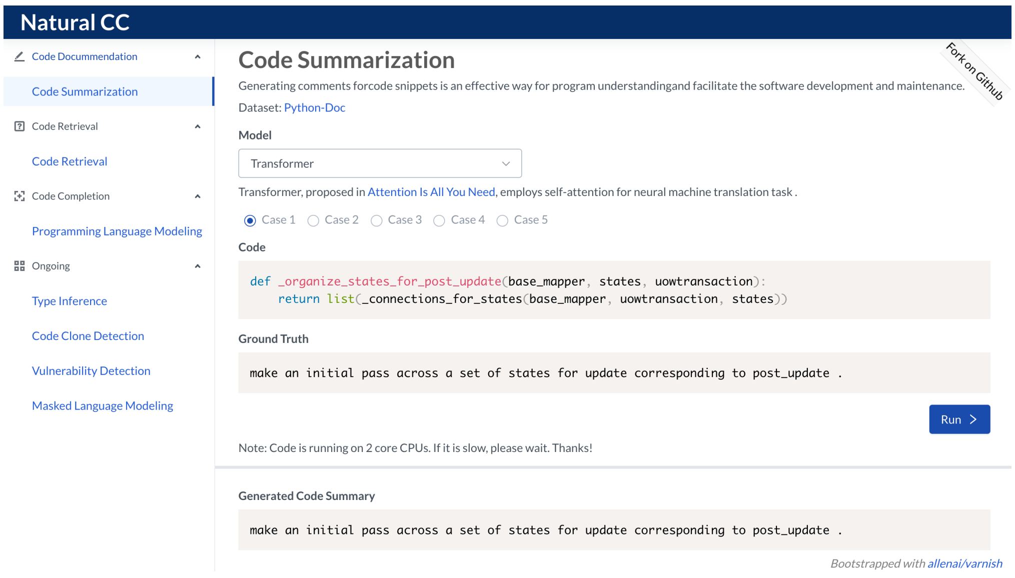
3.4 Type Inference

Automatically reasoning the type of a variable is beneficial for programmers to reduce bugs when using dynamically-typed languages. In this task, we have implemented two state-of-the-art deep learning models, DeepTyper (Hellendoorn et al., 2018) and Transformer (Jain et al., 2020), and evaluated them using Py150 dataset (Raychev et al., 2016), against the Accuracy metric. We evaluate this task under the setting of *all types* and *any types*.

On implementation, we tokenize each code snippet by space and build a vocabulary of size 40K. Both models are trained using four V100 GPUs with a learning rate of $1e\text{-}4$ and a batch size of 16. The performance of different models implemented in NaturalCC are summarized in Table 5.

4 Demonstration

Command Line Interface NaturalCC provides a command line interface that enables client users to simply explore the included state-of-the-art models. For each model, users can do inference using this command:



Code Summarization

Generating comments forcode snippets is an effective way for program understandingand facilitate the software development and maintenance.

Dataset: Python-Doc

Model: Transformer

Transformer, proposed in Attention Is All You Need, employs self-attention for neural machine translation task .

[✓ Case 1] Case 2 Case 3 Case 4 Case 5

Code

```
def _organize_states_for_post_update(base_mapper, states, uowtransaction):
    return list(_connections_for_states(base_mapper, uowtransaction, states))
```

Ground Truth

```
make an initial pass across a set of states for update corresponding to post_update .
```

Note: Code is running on 2 core CPUs. If it is slow, please wait. Thanks!

Generated Code Summary

```
make an initial pass across a set of states for update corresponding to post_update .
```

Figure 3: A screenshot of our graphical user interface for demonstration.

```
$python -m cli.predictor -m <model> -i <input>
```

where the second `-m` is the path of pre-trained model, and `-i` is the path of corresponding user input (e.g., a partial code snippet for the code completion task). To this end, NATURALCC will automatically load model parameters, process the user input and return the inferred results.

Graphical User Interface We have also provided a graphical user interface for users to easily access and explore each trained model's results through an online Web browser. The design of our Web is based on the open-source demo of AllenNLP (Gardner et al., 2017). We have deployed the graphical demo in the Nginx server and provided flexible APIs via the Flask engine[7].

As shown in Figure 3, 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 trained model will be invoked for inference and the generated summary will be displayed at the bottom of the given code. In this page, the users can also select the trained model accordingly.

5 Conclusion and Future Work

This paper presents NATURALCC, an efficient and extensible open-source toolkit to bridge the gap between natural language processing and programming languages. Currently, NATURALCC has implemented several state-of-the-art models across three popular software engineering tasks. We provide a detailed sample as an example to quickly implement a new task. We have provided a command line tool and a graphical user interface for other researchers to perform quick prototyping for demonstration.

For future work, we consider the following perspectives for further improvement. (a) We will extend this toolkit to more software engineering tasks, including code clone detection, vulnerability detection and translation between programming languages. More baseline models especially those based on GNN are also under our consideration. (b) A dashboard will be built so that users can submit their trained models to the platform for model competition and fair comparisons.

6 Artifacts and Resources

All the source code and materials are publicly available via GitHub: `http://github.com/CGCL-codes/naturalcc`.[8] We also build a website for this project `http://xcodemind.github.io`, and the demonstration video is also in this website. We encourage more researchers and developers to join our team to promote the development of this toolkit as well as the whole research community.

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

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

Broader Impact

Understanding programming languages from the perspective of natural language processing is a booming research area in the intersection of software engineering, programming language and NLP. This paper, for the first time, presents an open source toolkit NaturalCC for modeling the source code corpus from the perspective of NLP. It offers a set of tools ranging from data preprocessing, model implementations and model evaluation, as well as benchmarks many state-of-the-art models on different downstream tasks. Potentially, NaturalCC has the following impacts:

- For the communities of software engineering and programming language, NaturalCC can be seen as a toolkit to facilitate the model replication. The benchmark implemented by NaturalCC is also beneficial for comparing models in a fair way.
- For the NLP community, the provided code corpora in this paper can be considered as special corpora for NLP research, introducing more challenges and opportunities for structural representation learning.
- We also broaden the impact by extracting structural code features from the perspective of program analysis. This gives some inspirations on exploring graph structures to better represent programs' semantics through graph neural networks.

References

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2020. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*.

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

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2017. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*.

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

Antonio Valerio Miceli Barone and Rico Sennrich. 2017. A parallel corpus of python functions and documentation strings for automated code documentation and code generation. *arXiv preprint arXiv:1707.02275*.

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*.

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 (Volume 1: Long Papers)*, pages 823–833.

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*.

Matt Gardner, Joel Grus, Mark Neumann, Oyvind Tafjord, Pradeep Dasigi, Nelson F. Liu, Matthew Peters, Michael Schmitz, and Luke S. Zettlemoyer. 2017. Allennlp: A deep semantic natural language processing platform.

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

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Jian Yin, Daxin Jiang, et al. 2020. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*.

Vincent J Hellendoorn, Christian Bird, Earl T Barr, and Miltiadis Allamanis. 2018. Deep learning type inference. In *Proceedings of the 2018 26th acm joint meeting on european software engineering conference and symposium on the foundations of software engineering*, pages 152–162.

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*, pages 763–773.

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

W. Hua, Y. Sui, Y. Wan, G. Liu, and G. Xu. 2020. Fcca: Hybrid code representation for functional clone detection using attention networks. *IEEE Transactions on Reliability*, pages 1–15.

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*.

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 (Volume 1: Long Papers)*, pages 2073–2083.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph E Gonzalez, and Ion Stoica. 2020. Contrastive code representation learning. *arXiv preprint arXiv:2007.04973*.

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Pre-trained contextual embedding of source code. *ICML*.

Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. 2020. Big code!= big vocabulary: Open-vocabulary models for source code. *arXiv preprint arXiv:2003.07914*.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. 2020. Code prediction by feeding trees to transformers. *arXiv preprint arXiv:2003.13848*.

Taku Kudo and John Richardson. 2018. Sentencepiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. *arXiv preprint arXiv:1808.06226*.

Marie-Anne Lachaux, Baptiste Roziere, Lowik Chanussot, and Guillaume Lample. 2020. Unsupervised translation of programming languages. *arXiv preprint arXiv:2006.03511*.

Chris Lattner and Vikram Adve. 2004. Llvm: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.*, pages 75–86. IEEE.

Zhen Li, Deqing Zou, Shouhuai Xu, Xinyu Ou, Hai Jin, Sujuan Wang, Zhijun Deng, and Yuyi Zhong. 2018. Vuldeepecker: A deep learning-based system for vulnerability detection. *arXiv preprint arXiv:1801.01681*.

Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*.

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*.

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*.

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*.

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

Veselin Raychev, Pavol Bielik, and Martin Vechev. 2016. Probabilistic model for code with decision trees. *ACM SIGPLAN Notices*, 51(10):731–747.

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*, pages 419–428.

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*, pages 5998–6008.

S VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and YN Srikant. 2019. Ir2vec: Llvm ir based scalable program embeddings. *arXiv preprint arXiv:1909.06228*.

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 *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 13–25.

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*, pages 397–407. ACM.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems*.