NATURALCC: A Toolkit to Naturalize the Source Code Corpus

Yao Wan[1], Yang He[2], Jian-Guo Zhang[3],
Yulei Sui[2], Hai Jin[1], Guandong Xu[2], Caiming Xiong[4], Philip S. Yu[3]
[1]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]Department of Computer Science, University of Illinois at Chicago, Illinois, USA
[4]Salesforce Research, Palo Alto, USA
{wanyao,hjin}@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 toolkit to bridge the gap between natural language and programming language, and facilitate the research on big code analysis. Using NATURALCC, researchers both from natural language or programming language communities can quickly and easily reproduce the state-of-the-art baselines and implement their approach. NATURALCC is built upon Fairseq and PyTorch, providing (1) an efficient computation with multi-GPU and mixed-precision data processing for fast model training, (2) a modular and extensible framework that makes it easy to reproduce or implement an approach for big code analysis, and (3) a command line interface and a graphical user interface to demonstrate each model's performance. Currently, we have included several state-of-the-art baselines across different tasks (e.g., code completion, code comment generation, and code retrieval) for demonstration. The video of this demo is available at https://www.youtube.com/watch?v=q4W5VSI-u3E&t=25s.**

***Index Terms*—Natural language processing, programming language analysis, big code, toolkit.**

I. INTRODUCTION

The rapid growth of machine learning (ML), especially of deep learning (DL) based natural language processing (NLP), brings great opportunities to explore and exploit NLP techniques for various tasks of software engineering (SE), e.g., code documentation [1]–[3], code completion [4], [5] and code retrieval [6]. The underlying insights for learning-based code analysis is the naturalness hypothesis shared among natural languages and programming languages. Despite the flourishing study, many state-of-the-art approaches have been suffering from the replicability and reproducibility issues. This is due to the fact that the performance of deep learning approaches is sensitive to datasets and hyperparameters, and currently, no unified open-source toolkit is available to our communities.

To fill this gap, this paper introduces NATURALCC,[1] a comprehensive platform to facilitate research NLP-based big code analysis. Both of the researchers from SE community or NLP community can benefit from the toolkit for fast training and reproduction. NATURALCC features the following advantages:

- **Efficient Data Preprocessing.** We have cleaned and preprocessed four public datasets (i.e., CodeSearchNet [7], Py150 [8], and Python [1]) for different tasks. All the data loaders and processes in our model training can be parallelized in multiple GPUs. Besides, our toolkit also supports the mixed-precision for numerical calculation.
- **Extensibility and Modularity.** Based on the registry mechanism implemented in Fairseq [9], 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 the task and models in the corresponding folders and then register them.
- **Flexible Interface.** We provide flexible APIs for developers to easily invoke the trained models for other applications.

Additionally, 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 facilitate the research of software engineering with natural language processing. We also encourage researchers to integrate their state-of-the-art approach into NATURALCC, to promote the research in both communities.

All the source code and materials are publicly available via GitHub: http://github.com/xcodemind/naturalcc.[2] We also build a website for our team and will post the updates in http://xcodemind.github.io.

II. ARCHITECTURE DESIGN

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.,

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

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

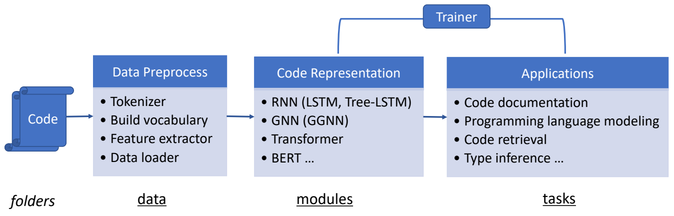


Figure 1: A pipeline of NATURALCC

RNN [10], GNN [11], Transformer [12] and BERT [13]). Based on the code representation, NATURALCC can also support various downstream tasks, e.g., code documentation, programming language modeling, code retrieval and type inference. The designed `Trainer` controls model training for each task.

A. Data Preprocessing

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, we can also extract some domain-specific features (e.g, AST [1], [2], control-flow graphs, or data-flow graphs [15]). The goal of this process is to build a series of mini-batches for training. We put all the data-related processes in the `data` and `dataset` folders.

B. 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 [12], which serve as the replacement of the RNN network, with its fast computation and ability to handle long-range dependent sequence. In addition, the NATURALCC also supports the masked pre-trained models, e.g., BERT and RoBERTa [16]. We put all the code representation networks in the `models` and `modules` folders.

C. Applications

Our NATURALCC supports many different downstream tasks. We have currently implemented three tasks, i.e., code completion, code comment generation, and code retrieval, to validate the effectiveness of the proposed framework. All the referred baselines on each task have been carefully checked and evaluated when compared against the released source code from the original papers. The implementations and the tasks in this toolkit will serve as baselines for fair comparison for future research use. We organize all the tasks in the `tasks` folder.

1) Code Completion: Code completion, which predicts the next code element based on the previously written code, has become an essential tool in many IDEs. It can boost developers' programming productivity. In this task, we have implemented the referred model SeqRNN [17] and TravTrans [4]. We train the models in the Py150 dataset and evaluate them using the MRR metrics.

2) Code Comment Generation: Generating comments for code snippets is an effective way for program understanding and facilitate the software development and maintenance. In this task, we have implemented the referred model NeuralTransformer [18]. We trained the model in Python and Java datasets and evaluated them using the BLEU [19] and Rouge [20] metrics.

3) Code Retrieval: Searching semantically similar code snippets given a natural language query can provide developers a series of templates as reference for rapid prototyping. In this task, we have implemented four benchmark baselines (i.e., NBOW, 1D-CNN, biRNN and SelfAttn) and evaluated them by using the MRR metrics on CodeSearchNet dataset [7].

Additional tasks with state-of-the-art models are still under development. They will be released soon, including code clone detection, type inference, vulnerability detection, and masked language modeling for code pre-training.

D. Trainer

We have designed a trainer (`ncc_trainer.py`) module to control the whole training process of models. Furthermore, we have designed and implemented a simpler trainer in a universal way (`ncc_trainer_simple.py`) for those beginners who are not familiar with our framework.

III. IMPLEMENTATION

We have implemented NATURALCC based on the Fairseq and PyTorch. Following the outstanding registry mechanism designed in Fairseq, NATURALCC has good extensibility with the modularized design.

A. Registry Mechanism

```python
def register_task(name):
    def register_task_cls(cls):
        if name in TASK_REGISTRY:
            raise ValueError('Duplicate
            ↪  error...')
        TASK_REGISTRY[name] = cls
        TASK_CLASS_NAMES.add(cls.__name__)
        return cls

    return register_task_cls
```

Listing 1: The registry mechanism in `__init__.py`

We have implemented a closure `register` function in the entry of building a task, model or module (cf. `__init__.py` in each folder). Listing 1 shows the workflow of registering a new task. In brief, the registry mechanism is to design a global variable to store each task of model objects for off-the-shelf fetching. This registry mechanism can provide us the ability of extension, as we only need to include this closure function when defining a new task/model/module in the corresponding function.

B. 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 to mitigate Multi-GPU computation lagging.

C. Mixed-Precision

NaturalCC can 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.

D. An Implementation Example

We take code completion as an example to show the pipeline of how to quickly build a new task in NaturalCC. Note that, in this section, we only describe the main steps in each file, and more details are referred to the corresponding source code files.

Building a task. In the first step, we create a `CompletionTask` in the `ncc/tasks/completion.py`, with a closure function `register_task` around. Listing 2 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.

```python
@register_task('completion')
class CompletionTask(NccTask):
    def __init__(self, args, dictionary):
        super().__init__(args)
        self.dictionary = dictionary

    @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 2: `tasks/completion/completion.py`

Building a model. Listing 3 shows the process of building an 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 network `LSTMDecoder`, which is implemented in the `modules` folder.

```python
@register_model('seqrnn')
class SeqRNNModel(NccLanguageModel):
    def __init__(self, args, decoder):
        super().__init__(decoder)
        self.args = args

    @classmethod
    def build_model(cls, args, config, task):
        decoder = LSTMDecoder(
            ...
        )
        return cls(args, decoder)
```

Listing 3: `models/completion/seqrnn.py`

Model training. Listing 4 shows the construction of a `Trainer` and the pipeline of train steps. Core parameters are involved in this `class` such that pre-trained models can be precisely restored during inference or fine-tuning.

```python
# 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 4: `trainer/trainer.py`

IV. Demonstration

A. Command Line Interface

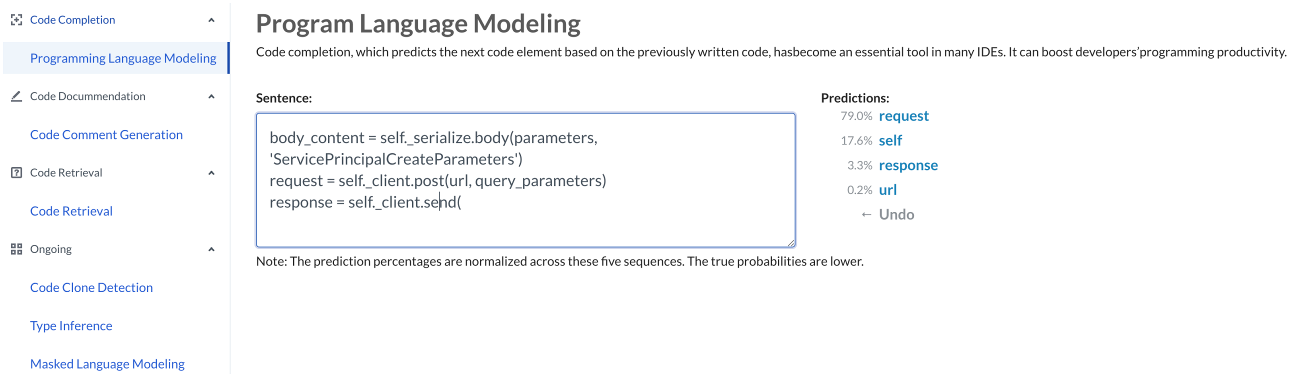
NaturalCC provides a command line interface that enables researchers and developers to simply explore the included state-of-the-art models. For each code analysis related tasks, users can try this command:

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

where `-m` is the pre-trained model directory, and `-i` is the corresponding user input (a partial code snippet in the code completion task). NaturalCC will automatically load model parameters, process the user input and return inference information in details.

B. 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. We have deployed the graphical demo in the Nginx server and provided flexible APIs via the Flask engine.



Program Language Modeling

Code completion, which predicts the next code element based on the previously written code, hasbecome an essential tool in many IDEs. It can boost developers'programming productivity.

Sentence:

```
body_content = self._serialize.body(parameters,
'ServicePrincipalCreateParameters')
request = self._client.post(url, query_parameters)
response = self._client.send(
```

Predictions:

```
79.0% request
17.6% self
 3.3% response
 0.2% url
← Undo
```

Note: The prediction percentages are normalized across these five sequences. The true probabilities are lower.

Figure 2: A screenshot of our graphical user interface for demonstration

As shown in Figure 2, we have integrated three popular software engineering tasks for demonstration, i.e., code completion, code comment generation, and code retrieval.

Taking code completion as an example. By default, we have implemented this task based on the programming language modeling. Given a series of written tokens by Python, the predicted tokens with corresponding probabilities generated by our model will appear simultaneously when the user enters the next tokens. In this page, the users can also select the trained model according to their requirements.

V. Conclusion and Future Work

This paper presents NaturalCC, an efficient and extensible open-source toolkit to bridge the gap between the programming language and natural language. 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. For demonstration, we have provided a command line tool as well as a graphical user interface for other researchers to do quick prototyping. All the materials about this toolkit can be accessed from http://xcodemind.github.io.

We will extend this toolkit to more software engineering tasks in our future work, including code clone detection, vulnerability detection, and masked language modeling. We also encourage more researchers and developers to join our team to promote the development of this toolkit as well as the whole research community.

Acknowledgements

The Fairseq highly inspires the NaturalCC for natural language processing. We appreciate the Fairseq team for their contribution and the high-quality backbone structure of the framework.

References

[1] Y. Wan, Z. Zhao, M. Yang, G. Xu, H. Ying, J. Wu, and P. S. Yu, "Improving automatic source code summarization via deep reinforcement learning," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 397–407.
[2] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 200–210.
[3] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," *arXiv preprint arXiv:1808.01400*, 2018.
[4] S. Kim, J. Zhao, Y. Tian, and S. Chandra, "Code prediction by feeding trees to transformers," *arXiv preprint arXiv:2003.13848*, 2020.
[5] F. Liu, G. Li, B. Wei, X. Xia, Z. Fu, and Z. Jin, "A self-attentional neural architecture for code completion with multi-task learning," in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 37–47.
[6] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 933–944.
[7] H. Husain, H.-H. Wu, T. Gazit, M. Allamanis, and M. Brockschmidt, "Codesearchnet challenge: Evaluating the state of semantic code search," *arXiv preprint arXiv:1909.09436*, 2019.
[8] "150k python dataset," https://eth-sri.github.io/py150, 2016, [Online; accessed 1-Nov-2020].
[9] M. Ott, S. Edunov, A. Baevski, A. Fan, S. Gross, N. Ng, D. Grangier, and M. Auli, "fairseq: A fast, extensible toolkit for sequence modeling," in *Proceedings of NAACL-HLT 2019: Demonstrations*, 2019.
[10] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural computation*, vol. 9, no. 8, pp. 1735–1780, 1997.
[11] Z. Wu, S. Pan, F. Chen, G. Long, C. Zhang, and S. Y. Philip, "A comprehensive survey on graph neural networks," *IEEE Transactions on Neural Networks and Learning Systems*, 2020.
[12] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.
[13] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "Bert: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[14] R.-M. Karampatsis, H. Babii, R. Robbes, C. Sutton, and A. Janes, "Big code!= big vocabulary: Open-vocabulary models for source code," *arXiv preprint arXiv:2003.07914*, 2020.
[15] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *arXiv preprint arXiv:1711.00740*, 2017.
[16] Y. Liu, M. Ott, N. Goyal, J. Du, M. Joshi, D. Chen, O. Levy, M. Lewis, L. Zettlemoyer, and V. Stoyanov, "Roberta: A robustly optimized bert pretraining approach," *arXiv preprint arXiv:1907.11692*, 2019.
[17] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2014, pp. 419–428.
[18] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," *arXiv preprint arXiv:2005.00653*, 2020.
[19] K. Papineni, S. Roukos, T. Ward, and W. J. Zhu, "Bleu: a method for automatic evaluation of machine translation," in *Proceedings of the 40th annual meeting on association for computational linguistics*. Association for Computational Linguistics, 2002, pp. 311–318.
[20] C. Y. Lin, "Rouge: A package for automatic evaluation of summaries," *Text Summarization Branches Out*, 2004.