NATURALCC: An Open-Source Toolkit for Code Intelligence

Yao Wan1,*, Yang He2, Zhangqian Bi1, Jianguo Zhang3, Yulei Su2, Hongyu Zhang4, Kazuma Hashimoto5, Hai Jin1, Guandong Xu2, Caiming Xiong6, Philip S. Yu3
1National 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
2University of Technology Sydney, Australia
3University of Illinois at Chicago, USA
4University of Newcastle, Australia
5Google Research, USA
6Salesforce 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:

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
We have collected three related datasets which have 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 150k Python source code files, which has been 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.

Figure 1: The pipeline of NaturalCC.

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 150k Python source code files, which has been 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 preprocessing 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 treewitter\(^1\) 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/

\(^{2}\)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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>Model</th>
</tr>
</thead>
<tbody>
<tr>
<td>Code Summarization</td>
<td>Python-Doc</td>
<td>Seq2Seq [12], Transformer [1], PLBART [2]</td>
</tr>
<tr>
<td>Code Completion</td>
<td>Py150</td>
<td>LSTM [10], GPT-2, TravTrans [15]</td>
</tr>
</tbody>
</table>

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 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 core module 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.

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 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 128.

<table>
<thead>
<tr>
<th>Task</th>
<th>Dataset</th>
<th>BLEU</th>
<th>METEOR</th>
<th>ROUGE-L</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>Seq2Seq+Attn</td>
<td>Python-Doc</td>
<td>25.57</td>
<td>14.40</td>
<td>39.41</td>
<td>0.09s/Batch</td>
</tr>
<tr>
<td>Tree2Seq+Attn</td>
<td>Python-Doc</td>
<td>23.35</td>
<td>12.59</td>
<td>36.49</td>
<td>0.48s/Batch</td>
</tr>
<tr>
<td>Transformer</td>
<td>Python-Doc</td>
<td>30.64</td>
<td>17.65</td>
<td>44.59</td>
<td>0.26s/Batch</td>
</tr>
<tr>
<td>PLBART</td>
<td>Python-Doc</td>
<td>32.71</td>
<td>18.13</td>
<td>46.05</td>
<td>0.26s/Batch</td>
</tr>
</tbody>
</table>
Table 3: MRR of code retrieval on CodeSearchNet.

<table>
<thead>
<tr>
<th>Attr.</th>
<th>Num.</th>
<th>Identifier</th>
<th>Param.</th>
<th>All Tokens</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>NBOW</td>
<td>66.59</td>
<td>59.92</td>
<td>47.15</td>
<td>54.75</td>
<td>46.33</td>
</tr>
<tr>
<td>Conv1D</td>
<td>70.87</td>
<td>60.49</td>
<td>38.81</td>
<td>61.92</td>
<td>67.29</td>
</tr>
<tr>
<td>BiRNN</td>
<td>65.80</td>
<td>48.60</td>
<td>23.23</td>
<td>51.36</td>
<td>48.28</td>
</tr>
<tr>
<td>SelfAttn</td>
<td>78.45</td>
<td>66.35</td>
<td>50.38</td>
<td>65.78</td>
<td>79.09</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Attr.</th>
<th>Num.</th>
<th>Identifier</th>
<th>Param.</th>
<th>All Tokens</th>
<th>Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>LSTM</td>
<td>51.67</td>
<td>47.45</td>
<td>46.52</td>
<td>66.06</td>
<td>73.73</td>
</tr>
<tr>
<td>GPT-2</td>
<td>70.37</td>
<td>62.20</td>
<td>63.84</td>
<td>46.52</td>
<td>62.20</td>
</tr>
<tr>
<td>TravTrans</td>
<td>72.08</td>
<td>68.55</td>
<td>76.33</td>
<td>71.08</td>
<td>83.17</td>
</tr>
</tbody>
</table>

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 feature from AST. We tokenize each code snippet by white space and build a vocabulary with 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 3.

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.

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.

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.

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 how to construct a Trainer object and the training steps. Core parameters are stored in this process so 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.

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 interface.
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.\(^3\) 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 grant No. 620102157. 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**


