Instructions of DAN's Artifact

Introduction

`DAN` (Dependency-aware Naturalness) is a novel method for measuring code naturalness by incorporating the rich dependency information present in the code. DAN extracts multiple sequences of code lines by traversing the program dependency graph, where different code lines are connected by dependencies in each sequence. The naturalness of the code is then measured by taking each sequence as a whole, effectively capturing the dependency information.

An extensive study was conducted to evaluate the influence of code dependencies on measuring code naturalness using DAN, comparing it with state-of-the-art methods (Ngram-NT and CodeBERT-NT) across three emerging application scenarios. This overview will introduce the guidelines for using DAN and replicating the experiments.

Artifact components

The artifact is provided as a docker image. The artifact comprises two main components: the dependency analysis tool and the experiment scripts. The dependency analysis tool extracts code dependencies and connects multiple code lines based on these dependencies. It is provided as a `jar` library, ensuring ease of transfer and use. The experiment scripts leverage the extracted code lines to enhance the naturalness measure and evaluate its effectiveness on downstream tasks. These scripts are provided as source code along with the benchmarks used:

Distinguishing Natural and Unnatural Code (RQ1): First, we investigated DAN's ability to distinguish between natural and unnatural code. The results demonstrate that DAN outperforms Ngram-NT and CodeBERT-NT in this task.

All experiment scripts and benchmarks needed to replicate this evaluation are included in the artifact. The benchmarks are in the folder `/artifacts/dependency-aware-code-naturalness-main/data/rq1_sources`, the script is `/artifacts/dependency-aware-code-naturalness-main/rq1-unnatural.py`. The execution results will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ1-unnatural`.

Distinguishing Buggy and Non-Buggy Code Lines (RQ2): To evaluate DAN in distinguishing buggy and non-buggy code lines, we used benchmarks such as Defects4J, GrowingBugs, and SmartSHARK. The results show that DAN consistently outperformed the compared techniques.

All experiment scripts, including data preprocessing, naturalness measurement, and buggy line prioritization, are provided in the artifact. Due to the large size of the benchmarks (hundreds of GBs), only projects from Defects4J are included as the representative benchmark. Since running through Defects4j also requires significant time, we further provide parallel execution scripts to accelerate the process and small-scale representative examples for scenarios where parallel execution is not feasible.

The benchmarks are in the folder `/artifacts/dependency-aware-code-naturalness-main/data/rq2_sources`, the script for data processing and naturalness measure is `/artifacts/dependency-aware-code-naturalness-main/rq2-step1-d4j`, the script for processing the results is `rq2-step2-result.py`. The execution results will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ2-buggy`.

Cleansing Training Data for Building Better Code Generation Models (RQ3): For cleansing training data to build better code generation models, we used benchmarks such as APPS and HumanEval. The results indicate that DAN consistently outperformed the compared techniques.

All the benchmarks are provided. The artifact also includes experimental scripts for data preprocessing, naturalness measurement, training data sampling, and fine-tuning code models.

The benchmarks are in the folder `/artifacts/dependency-aware-code-naturalness-main/data/rq3_sources`, the script for data processing and naturalness measure is `/artifacts/dependency-aware-code-naturalness-main/rq3-step1-count`, the script for sampling training data is `rq3-step2-*.py`. The execution results will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ3-buggy`.

Specifically, we built DAN using two underlying models: Ngram and CodeBERT. In our artifact, we provide the DAN$_{ngram}$ vs. Ngram-NT version and omit the DAN$_{codebert}$ vs. CodeBERT-NT comparison. This omission is due to two reasons: (1) executing the latter requires significant time and computing resources because of the complexity of CodeBERT-NT; and (2) the underlying models are orthogonal to DAN and not the main focus of this artifact.

Hardware Dependencies

The artifact is tested on a server with Intel(R) Xeon(R) Silver 4214 @ 2.20GHz CPU, 256GB memory, and Ubuntu 18.04 operating system. The artifact requires linux operating system and at least 64GB memory to exercise.

Getting Started Guide

Download the whole package from the URL, there should be three files: `dan_artifact.tar`, `source_code.tar` and `d4j_projects.tar`. The first one is a docker image, the second one is the source code, the third one is the benchmark needed.

Unzip the source code, this should result in a folder named `artifacts` in current dir:

```
tar -xvf source_code.tar
```

Then unzip the data in the `./artifacts/dependency-aware-code-naturalness-main/data/rq2_sources/dataset_sources/d4j` folder:

```
tar -xvf d4j_projects.tar
```

This should result in a folder named d4j_projects in `./artifacts/dependency-aware-code-naturalness-main/data/rq2_sources/dataset_sources/d4j` dir.

Import the docker image:

```
docker import dan_artifact.tar dan_artifact_image
```

Then run the docker container and put the data in the docker:

```
docker run --name dan_artifacts -v /path/to/artifacts:/artifacts -it {image_id} /bin/bash
```

In the docker container activate the reqiured environment:

```
source ~/.bashrc
conda activate dan
```

Then the setup is finished. The artifact is in the folder `/artifacts`.

Step by Step Instructions

Enter the folder `/artifacts/dependency-aware-code-naturalness-main` first.

Distinguishing natural and unnatural code

To replicate RQ1, run:

```
python rq1-unnatural.py
```

The process should take about seven minutes, and the output should look like this:

```
/artifacts/dependency-aware-code-naturalness-main
[2024-07-06 16:20:52,069 - rq1-unnatural.py - <module>] - The running process begins.
[2024-07-06 16:20:52,070 - rq1-unnatural.py - <module>] - Loading the underlying models.
[2024-07-06 16:23:44,758 - rq1-unnatural.py - <module>] - Finished loading the underlying models.
[2024-07-06 16:23:44,759 - rq1-unnatural.py - run] - Begin processing the corpus with the extracted sub-paths.
[2024-07-06 16:24:48,644 - rq1-unnatural.py - run] - Finished processing the corpus with the extracted sub-paths.
[2024-07-06 16:24:48,644 - rq1-unnatural.py - run] - Begin processing the corpus without the sub-path extraction.
[2024-07-06 16:24:50,634 - rq1-unnatural.py - run] - Finished processing the corpus without the sub-path extraction.
[2024-07-06 16:25:10,936 - rq1-unnatural.py - run] - Begin the naturalness measurement.
100%|███████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████| 2142/2142 [00:00<00:00, 4483.95it/s]
100%|████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████████| 2142/2142 [00:09<00:00, 236.76it/s]
[2024-07-06 16:25:20,490 - rq1-unnatural.py - run] - Saving the naturalness results.
[2024-07-06 16:25:20,563 - rq1-unnatural.py - <module>] - The running process finishes!
[2024-07-06 16:25:20,563 - rq1-unnatural.py - <module>] - Begin processing the results!
```

```
[2024-07-06 16:25:53,299 - rq1-unnatural.py - <module>] - Finished processing the results!
```

The unprocessed row results will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ1-unnatural/rq1_test.pkl`, which is used for further analysis.

The analysed results will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ1-unnatural/table1.csv`, the main conclusion should be consistent with the paper: DAN outperforms all compared techqniues.

To reproduce the results in Table 5, follow these steps:

1. Navigate to the directory:

   ```
   cd /artifacts/dependency-aware-code-naturalness-main/table_5
   ```

2. Install the required dependencies:

   ```
   pip install torch transformers
   ```

3. Download the CodeLlama-7b-Instruct model from Hugging Face. Assume the model is saved to `/path/to/the/model`.
4. Update the path to the model in `llm_entropy.py` at Line 200:

   ```
   200 model_name = "/path/to/the/model"
   201 tokenizer = AutoTokenizer.from_pretrained(model_name)
   202 model = AutoModelForCausalLM.from_pretrained(model_name)
   ```

5. Run the script to print the results to the console:

   ```
   python llm_entropy.py
   ```

Distinguishing buggy nonbuggy lines

First check line 439 and line 440 of `rq2-step1-d4j.py` to set the parallel execution and the benchmarks.

```
# used for parallel execution, n_process represents the available number of processes
n_process = 40
# run the whole benchmark by default, set example_cnt to 10 to run small-scale examples
example_cnt = 'all'
```

Then run the following command to measure the naturalness for all lines in the benchmark:

```
python rq2-step1-d4j.py
```

Then you can check the log file `artifacts/logs/RQ2-buggy.log`, it should look like this (if running the whole benchmark):

```
Todo size 702
wait info: 0/702 case(s) have been done! 0/702 case(s) have crashed!
wait info: 19/702 case(s) have been done! 2/702 case(s) have crashed!
Todo size 699
wait info: 0/699 case(s) have been done! 0/699 case(s) have crashed!
wait info: 110/699 case(s) have been done! 37/699 case(s) have crashed!
wait info: 147/699 case(s) have been done! 37/699 case(s) have crashed!
wait info: 177/699 case(s) have been done! 37/699 case(s) have crashed!
wait info: 208/699 case(s) have been done! 37/699 case(s) have crashed!
wait info: 240/699 case(s) have been done! 38/699 case(s) have crashed!
wait info: 316/699 case(s) have been done! 38/699 case(s) have crashed!
wait info: 395/699 case(s) have been done! 38/699 case(s) have crashed!
wait info: 518/699 case(s) have been done! 38/699 case(s) have crashed!
wait info: 638/699 case(s) have been done! 51/699 case(s) have crashed!
wait info: 641/699 case(s) have been done! 58/699 case(s) have crashed!
Success size 641
Fail size 58
```

The row results will be in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ2-buggy/d4j-done`.

Then run the following command to perform buggy line prioritization:

```
python rq2-step2-result.py
```

The processed result will be `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ2-buggy/processed_results/d4j/table2.csv'`, which should be consistent with the paper.

Data cleansing

First check line 279 of `rq3-step1-count.py` to set the benchmark:

```
dataset = 'APPS'
# dataset = 'humaneval'
```

Run the following command to measure the naturalness of each code in the benchmark:

```
python rq3-step1-count.py
```

The output should be like:

```
/artifacts/dependency-aware-code-naturalness-main
[2024-07-06 16:46:47,319 - rq3-step1-count.py - <module>] - Begin running process for
dataset APPS
[2024-07-06 16:46:47,333 - rq3-step1-count.py - <module>] - Loading the underlying models.
[2024-07-06 16:49:27,567 - rq3-step1-count.py - <module>] - Model loaded, begin measuring
naturalness!
100%|
█████████████████████████████████████████████████████████████████████████████████████████
███████████████████████████████████████████████████████████████| 4973/4973 [00:16<00:00,
293.79it/s]
[2024-07-06 16:52:15,317 - rq3-step1-count.py - <module>] - Naturalness measured! Find the
detailed results in './data/results/RQ3-finetune/APPS/APPS.pkl'!
[2024-07-06 16:52:15,317 - rq3-step1-count.py - <module>] - Sampling training data
according to code naturalness.
[2024-07-06 16:52:15,497 - rq3-step1-count.py - <module>] - Find the sampled results in
'./data/results/RQ3-finetune/APPS/ngram_dataset.json'!
[2024-07-06 16:52:15,497 - rq3-step1-count.py - <module>] - Running process for dataset
APPS finishes!
```

The row results will be in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ3-finetune/{dataset}/{dataset}.pkl`

The processed results will be in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ3-finetune/{dataset}/ngram_dataset.json`. Each key in the json file is a code in the benchmark and the corresponding value is the naturalness measured for it, the json items is ranked according to the naturalness scores.

Then run `rq3-step2-process.py` to get the training data selected by each method.

First set the dataset at line 40:

```
dataset = 'APPS'
# dataset = 'humaneval'
```

then run:

```
python rq3-step2-process.py
```

The sampled training data will be stored in `/artifacts/dependency-aware-code-naturalness-main/data/results/RQ3-finetune/{dataset}/{technique}_top.json` files. Each item in the json corresponds to one data point, including the prompt and the ground-truth.

Reusability Guide

Our tool for dependency extraction is a core component that should be evaluated for reusability. You can find it at:

`/artifacts/dependency-aware-code-naturalness-main/tool/dependency-extraction.jar`

Usage:

```
java -cp ../data/javassist-3.28.0-GA.jar:dependency-extraction.jar Main path-to-the-code
path-to-result
```

Note: Both the source code and the byte code produced by the compiler should be in the `path-to-the-code` (e.g., both `.java` files and corresponding `.class` files are needed). The extracted dependency information will be stored at `path-to-result`.

Example: For the project located at `/artifacts/dependency-aware-code-naturalness-main/data/rq1_sources/mutant_0728_nosub`, the source code is in the `src` folder and the byte code is in the `target` folder.

Navigate to `/artifacts/dependency-aware-code-naturalness-main/tool` and run:

```
java -cp ../data/javassist-3.28.0-GA.jar:dependency-extraction.jar Main
../data/rq1_sources/mutant_0728_nosub test.json
```

This process takes about five minutes and produces output like:

```
1
2174
```

The result file `test.json` will be created in the same folder, where each key represents a file, and the value represents connected lines (sequences with dependencies).

De-Naturalization Tool

Our de-naturalization tool is provided as a JAR file, located at:

`/artifacts/dependency-aware-code-naturalness-main/tool/denaturalize.jar`

Usage:

```
java -jar denaturalize.jar path-to-the-code path-to-result
```

`path-to-the-code` should be a directory containing a batch of compilable Java files, such as:

```
- path-to-the-code
  - 1.java
  - 2.java
  - ...
```

This tool can handle any syntactically correct and compilable Java files.

Example: If the Java files are in `/artifacts/dependency-aware-code-naturalness-main/data/rq1_sources/mutant_0728_nosub/src/main/java`, the command should be:

```
java -jar denaturalize.jar /artifacts/dependency-aware-code-naturalness-main/data/rq1_sources/mutant_0728_nosub/src/main/java path-to-result
```

The mutants will be stored in `path-to-result`.

The source code for de-naturalization is located in the folder `./tool/de-naturalization`. Please refer to the README file in that directory for more details about compiling it from sources.

The `jar` format ensures our tool can be easily transferred to other projects or used independently.