Analyzing Software using Deep Learning
– Project Description, Summer Semester 2022 –

Prof. Dr. Michael Pradel, Islem Bouzenia, Moiz Rauf

May 17, 2022

1 Introduction

Runtime exceptions are commonly used to abort the execution when a piece of code receives inappropriate input or reaches an illegal state. We call code that raises an exception under a specific condition an *if-condition-raise statement*. Ideally, such a statement should raise the exception exactly under the appropriate condition, and when an exception gets raised, it should provide an informative error message that explains what went wrong. To ensure these properties, the condition under which an exception gets raised and the exception itself, including its error message, must be consistent. The first row of Table 1 illustrates a consistent if-condition-raise statement from the Keras project.

Unfortunately, not all if-condition-raise statements are consistent, which may harm the robustness of a program and make debugging a program crash unnecessarily difficult. There are two main reasons for inconsistent if-condition-raise statements. On the one hand, the condition may not accurately reflect when the developer intends to raise an exception. For example, consider the second row of Table 1. The condition is equivalent to `not (len(bits) == 4 and len(bits) == 6)`, which will always evaluate to `True`, while the message states that the reason for the exception is that `len(bits)` is not among the values `(4,6)`. On the other hand, the kind of exception or its associated error message may be incomplete, misleading, or even outright wrong. For example, the third row of Table 1 shows a real-world example of this scenario. The exception message incorrectly claims the problem to be that `n1 > n2`, while the condition actually is raised when `n1 < n2`.

In general, we say an *if-condition-raise* statement is inconsistent if the condition does not semantically imply the exception message or if the exception message does not imply the condition.

Table 1: Examples of consistent and inconsistent if-condition-raise statements extracted from real-world projects.

Id	If-condition-raise statement	Type	Error Location	Project

1	`if mode not in {'caffe', 'tf', 'torch'}:`
	`  raise ValueError('Expected mode to be one of ‘caffe‘, ‘tf‘ or ‘torch‘. Received: mode={mode}')`
		Consistent	–	Keras

2	`if len(bits) != 4 or len(bits) != 6 :`
	`  raise template.TemplateSyntaxError("%r takes exactly four or six arguments (second argument must be 'as')" % str(bits[0]))`
		Inconsistent	Condition	Pinax

3	`if n2 > n1 :`
	`  raise ValueError('Total internal reflection impossible for n1 > n2')`
		Inconsistent	Message	Sympy

2 Goal

The goal of this project is to design, implement, and evaluate a neural network-based program analysis that identifies *inconsistent if-condition-raise statement* in Python source code. For a given piece of code, the analysis should extract if-condition-raise statements and predict an inconsistency score for each statement.

The target language of the analysis is Python. Thus, the dataset and evaluation examples are all extracted from Python code, more specifically Python 3. The implementation must be in Python and work with Python version 3.8. The neural network part of the implementation should build on Pytorch (and work with Pytorch version 1.11.0).

3 Dataset & Resources

To train such a model, we need a dataset containing both consistent and inconsistent instances of if-condition-raise statements. As a starting point for creating such a dataset, we provide a corpus of function definitions containing consistent if-condition-raise statements. It is up to the student to propose effective approaches for generating inconsistent instances from the consistent ones. Specifically, we provide a dataset of 100K unique function definitions extracted from open-source projects on GitHub. Most of the function definitions contain at least one if-condition-raise statement. The raw source code of the functions is stored in a list in textual format and saved into a JSON file. The entire dataset has a size of about 100MB. Students should use the provided dataset, and are not allowed to train their model on any additional code or other data.

We also provide a set of pre-trained models that might help in some steps of the project:

- FastText embedding model: We pre-trained the model on 6.5 million if-condition-raise statements for 300 epochs. The embedding dimension is 32.
- Byte Level BPE tokenizer: We pre-trained this tokenizer on a dataset of 1.5 million function definitions. It has a vocabulary of 100K tokens.

Finally, we attach examples of loading the data, of using the pre-trained FastText model, of tokenizing and vectorizing the code, and also of parsing and manipulating the code using the LibCST AST library.

4 Milestones

This project has been broken down into three milestones for better guidance throughout the semester. Each milestone has a progress meeting, in which each student meets with their mentor, presents the deliverable, and seeks help if needed. Some code templates have been prepared for each milestone, in which the students will implement their solutions.

4.1 Milestone 1 (May 30 and 31, 2022)

By this progress meeting, the students are expected to have extracted all *if-condition-raise-statement* from the provided list of function definitions. We suggest using a parser to turn the raw code into an AST, and to then extract the if-condition-raise statements. However, we do not impose an AST-based approach, and any other approach that allows achieving the same goal is accepted. In the case of using an AST-based parser, we suggest the LibCST package, for which we provide some usage examples in the shared resources. We also do not impose any specific format to represent the *if-condition-raise-statement* after extraction. It is up to the students to decide which format they want to use for the next steps.

4.2 Milestone 2 (June 13 and 14, 2022)

By the second milestone, we expect the students to have trained a binary classifier to classify if-condition-raise statements as consistent or inconsistent. To that end, the students should use the data they extracted in Milestone 1. By formulating the task as binary classification, the students will need both consistent and inconsistent instances of if-condition-raise statements. In Milestone 1, the extracted statements are all consistent. To obtain inconsistent if-condition-raise statements we propose a simple generation heuristic called random recombination. Random recombination randomly selects two if-condition-raise statements from the extracted data and then combines the condition of the first statement with the exception message of the second statement, and vice versa. Below, we give an example of random recombination.

```
# statement 1: consistent
if x < 0:
    raise ValueError("x should be positive, got x={v}".format(v=x))

# statement 2: consistent
if not isinstance(result, dict):
    raise TypeError("expected result to be of type dict")

## recombination 1: likely inconsistent
if x < 0:
    raise TypeError("expected result to be of type dict")

## recombination 2: likely inconsistent
    raise ValueError("x should be positive, got x={v}".format(v=x))
```

Student should implement the random recombination heuristic based on the format they used to extract and represent the data in Milestone 1. Once the dataset of both consistent and inconsistent statements is created, students should use the resulting dataset to train a binary classifier that reasons about the consistency of a statement. The classifier should output the inconsistency score of a given statment, which can be interpreted as the probability that a statement is inconsistent. In other words, the value 1.0 corresponds to an inconsistent statement and the value 0.0 corresponds to a consistent statement.

The implementation should use two files provided by us. Specifically, students should implement a `Train.py` file, that reads a JSON file formatted as in the provided dataset of raw function definitions, and outputs a serialized Pytorch model[1]. Students should also implement a `Predict.py` file, that takes a serialized model and a JSON file containing a list of n functions definitions as the input, and outputs a JSON file containing a list of n dictionaries. Each dictionary represents the inconsistency score(s) for one function. The key in these dictionaries correspond to the line number at which the exception is raised, where each function definition starts at line 1. The value associated to that key is the inconsistency score of the raised exception w.r.t. its condition(s). A sample output file is provided alongside the template code. In the example below, we give an instance of the input and the format of the expected output:

```
## Code example to pass to predict.py. The input code is saved in a JSON file containing a
    list of functions definitions.
[   # function definition 1
    """
    def test_func(a, b, c): # line number = 1
        if a == 0:
            raise Exception('a should not be zero')
        elif b == 0:
            raise Exception('b cannot be zero when a is zero')
        else:
            return c/a + c/b
    """,
    # function definition 2
    ...
]
## Output file format for the previous example
[
    # prediction output for function definition 1
    {
        "3": 0.,
```

[1]https://pytorch.org/tutorials/beginner/saving_loading_models.html

```
        "5": 1.,
    },

    # prediction output for function definition 2
    ...
]
```

4.3 Milestone 3 (June 27 and 28, 2022)

In the shared resources, you find two files real_consistent.json and real_inconsistent.json. Each file contains a list of ten function definitions extracted from projects on GitHub. The real_consistent.json file contains consistent instances, while the real_inconsistent.json contains inconsistent version of the same instances. By this milestone, the students are supposed to improve their solutions to perform well on these real-world instances. The improved solution should include:

- New strategies for generating inconsistent instances of if-condition-raise statements. The random re-combination approach from Milestone 2 tends to produce unrealistic inconsistent examples, which the students should improve upon in Milestone 3.
- An improved model architecture and better fine-tuning of the hyperparameters.

The evaluation of this milestone will be based on a held-out test set of real instances of if-condition-raise statements. The main criteron is that your solution generalizes to real cases. The input and output format are the same as in Milestone 2.

Similar to Milestone 2, students should implement a `Train.py` and a `Predict.py`. The format of the inputs and outputs is the same as in Milestone 2.

5 Mentoring and How to Submit

Each student has a mentor, either Islem Bouzenia (islem.bouzenia@iste.uni-stuttgart.de) or Moiz Rauf (moiz.rauf@iste.uni-stuttgart.de). Students must meet their mentor at least three times during the semester, on the dates indicated for the milestones. In addition, students may consult their mentor to resolve any questions, to ensure progress in the right direction, and to help students submit a successful project in time. We also recommend to ask general questions in the Ilias forum.

The deadline for submitting the project is July 15, 2022. This deadline is firm. The submission will be via Ilias and should include:

- A scientific report of at most four pages that summarizes the approach taken and the results obtained.
- A .zip file with your implementation. The implementation must be usable via the scripts we provide as a template. *Please do not change the interface of these scripts.*

Each person must present the project on July 18 or 19, 2022, in a short talk, followed by a question and answer session.

The project is individual, i.e., students must work by themselves and not share their solution with anyone else. Any form of collaboration that results in similar or identical solutions being submitted will be punished according to the usual rules for plagiarism. In this case, the punishment affects both students, i.e., both the student who shared (parts of) a solution and the student who used it.

Grading will be based on the following criteria:

Criterion	Contribution
Progress meetings and final presentation (progress made, clarity, illustration, quality of answers)	25%
Implementation (completeness, documentation)	25%
Results (soundness, reproducibility)	25%
Report (clarity, illustration, discussion and interpretation of the results)	25%