LOG6305 - Techniques avancées de test du logiciel

Assignment 4

LLM-based test generation

Département de génie informatique et de génie logiciel
École Polytechnique de Montréal

Winter 2024

1 Introduction

The automatic generation of unit tests is an important topic in Software Engineering (SE) [1]. It aims to reduce developers' testing efforts. Developing good-quality unit tests can prevent bugs in software products. There are different tools for automatically generating unit tests and test suites that are either based on random test generators [2], dynamic symbolic execution [3], or search-based approaches [4]. However, these techniques have some drawbacks and often generate tests with no assertion or too general assertions, or tests with assertions that cannot effectively assess the intended behavior of the program under test. Considering these shortcomings, researchers have recently been exploring the possibility of leveraging Machine Learning-based code synthesis techniques for generating better unit test. Specifically, these approaches have been exploring the potential of Large Language Models (LLMs) with the transformer architecture. In this assignment, you will get familiar with how you can use an LLM for unit test generation.

2 Objectives

The objectives of this lab are :

1. Explore the current stat-of-the-art LLMs. Those that can be run on powerful cloud infrastructure, like Lamma2 [1] and those that can be run locally, as COdet5 [2].
2. Learn how to use the LLMs for unit test generation in practice with zero-shot and few-shot inference techniques.

3 Interacting with Large Language Models : Understanding Prompting Methods

Understanding different prompting strategies is crucial for leveraging LLMs, especially in coding and problem-solving contexts. Here, we introduce two primary prompting methods : zero-shot, few-shot, and chain-of-thought.

3.1 Zero-Shot Prompting

LLMs are tuned to follow instructions and are trained on large amounts of data ; so they are capable of performing some tasks "zero-shot". Zero-shot prompting refers to the scenario where the model is asked to perform a task without any prior examples. In this approach, the prompt must be self-contained and explicitly describe the task. The model relies solely on its pre-trained knowledge to generate a response. For instance, when working with code, a zero-shot prompt might directly ask the model to write a function based on a description of its functionality, without providing any examples of similar functions.

Example :

Prompt :

1. `https://huggingface.co/docs/transformers/main/en/model_doc/llama2`
2. `https://github.com/salesforce/CodeT5`

```
Generate a unittest for the function isPalindrome
[FUNCTION]
def isPalindrome(x):
    if x < 0 or (x != 0 and x % 10 == 0):
        return False

    half = 0
    while x > half:
        half = (half * 10) + (x % 10)
        x = x // 10

    return x == half or x == half // 10
    [\FUNCTION]
```

LLM output :

```
class TestIsPalindrome(unittest.TestCase):
def test_palindrome(self):
    self.assertTrue(isPalindrome(121), "121 is a palindrome")
    self.assertTrue(isPalindrome(0), "0 is a palindrome")
```

3.2 Few-shot prompting

Few-shot prompting involves giving the LLM a few examples of the task to be performed before presenting the actual task. This method helps the model understand the context and the specific format or style of the expected output. In the context of coding, a few-shot prompt might include several examples of functions that perform similar but not identical tasks, followed by a description of the new function to be generated. The model uses these examples to better understand the task and produce a more accurate response.
Example Prompt :

```
Generate a unit tests for the function isPalindrome.

[FUNCTION]
def isPalindrome(x):
    if x < 0 or (x != 0 and x % 10 == 0):
        return False

    half = 0
    while x > half:
        half = (half * 10) + (x % 10)
        x = x // 10

    return x == half or x == half // 10
```

```
    [\FUNCTION]

    This is an example of a unit test to follow.

    [EXAMPLES]
    def test_palindrome():
        self.assertTrue(isPalindrome(1331), "1331 is a palindrome")
        self.assertTrue(isPalindrome(0), "0 is a palindrome")

    def test_not_palindrome():
        self.assertFalse(isPalindrome(123), "123 is not a palindrome")
        self.assertFalse(isPalindrome(-121), "-121 is not a palindrome")
        self.assertFalse(isPalindrome(10), "10 is not a palindrome")
    [\EXAMPLES]
```

output :

```
class TestIsPalindrome(unittest.TestCase):

    def test_positive_palindromes(self):
        self.assertTrue(isPalindrome(1331), "1331 is a palindrome")
        self.assertTrue(isPalindrome(0), "0 is a palindrome")
        self.assertTrue(isPalindrome(12321), "12321 is a palindrome")
        self.assertTrue(isPalindrome(1), "1 is a palindrome")

    def test_negative_palindromes(self):
        self.assertFalse(isPalindrome(-121), "-121 is not a palindrome ")
        self.assertFalse(isPalindrome(-12321), "-12321 is not a palindrome")
```

For more information regarding prompt engineering please refer to this link `https://www.promptingguide.ai/fr`

4 The tasks

4.1 Required set-up

Operational system : Linux/macOS/Windows, **Python** : version = 3.9. Clone the following repository : `https://github.com/log6305/HIV_2024_TP4`

Setting Up the Model

1. Model Selection : The lab exercises will utilize the CodeLLAMA available at Hugging Face [3]. This model has been specifically designed for coding tasks.

3. `https://huggingface.co/chat/`

2. Model configuration in chat interface : Navigate to the provided link and ensure to set "CodeLlama-70b-Instruct-hf" as the current model in the chat interface. This can be done by selecting it as the current model from the model selection dropdown.

4.2 Your tasks

1. Question 1 (Exploration) : In this exercise, we will use the CodeLLAMA model available at Hugging Face[4]. This model has been specifically designed for coding tasks. Navigate to the provided link and ensure to set "CodeLlama-70b-Instruct-hf" as the current model in the chat interface. This can be done by selecting it as the current model from the model selection dropdown. In the "poly_llm/to_test/" folder of your repository, you will find 5 files with functions to test (5 functions) : . We also refer to them as programs under tests (PUTs).

   -Question 1.1. Generate the unit tests with zero shot prompting technique for each of the functions. Construct a prompt similar to the one shown in the examples in the previous sections of this assignment. Record the model output in a separate file named "test_zero_shot_function_name.py". With the help of pytest measure the line and branch coverage achieved. Record the values in the report (2 points).

   -Question 1.2. Generate the unit tests with few shot prompting technique for each of the functions. To get the examples for the "few shot" prompting, you can use the assertions existing in the function file or use your own assertions. Use up to 3 examples. Record the model output in a separate file named "test_few_shotfunction_name.py". With the help of pytest measure the line and branch coverage achieved. Note the values in the assignment report (2 points). Discuss how the results are different from those obtained with "zero shot technique" (2 point) i.e. discuss the difference in the semantics of the tests as well as in the coverage they achieve. Does adding more few-shot examples improve the quality of the tests (according to your judgement as a tester) (1 points) ?

2. Question 2 : Based on the provided examples in "getting_started.ipynb" generate tests for 5 provided inputs file with an LLM that runs on your machine locally or on Google Collab. For using google collab you can refer to an example in "example_google_collab.ipynb".

   -Question 2.1. Measure the initial coverage (line and branch coverage) of the program under test (PUT) by the assertions that are specified in the same file using the "AbstractExecutor" class provided. Record the values in the report (5 points i.e. 1 point for each program)

   -Question 2.2. Promt the LLM using the zero-shot technique to generate unit tests for the 5 PUTs. Use "PromptGenerator" class to produce the prompts. Record the line coverage and branch coverage (in %) achieved by the tests from LLM in the report. By executing the already existing assertions and the tests produced by LLM estimate if the new tests produced by LLM increase the initial coverage obtained by existing assertions (either line or branch coverage) (10 points i.e. 2 points for each PUT).

   -Question 2.3. Repeat the steps in the question 2.2, but use the "few-shot" prompting technique. Experiment with 1 few shot example. To generate the example you

4. `https://huggingface.co/chat/`

can use the existing assertions. If you read and insert the existing assertions via code you earn additional 3 points. You also have the option to manually copy the existing assertions, but you don't earn the extra points. Record the obtained coverage (line and branch) and compare with the baseline (existing assetions). (10 points i.e. 2 points for each PUT).

-Q2.4. Your goal is to maximize the additional coverage obtained from the LLM generated tests. Experiment with different few-shot examples. To change the examples, try removing or adding some new assertions. Try at least 3 different few-shot promts. You can use up to 2 few-shot examples. Record all the obtained tests generated with LLMs over the combinations you tried, and report the total coverage achieved by the baseline assertions and the LLM generated tests. Do LLMs help improve the initial coverage ? (10 points i.e. 2 points for each PUT). Save your implementation in "question2.py" file.

3. Question 3 (using a new LLM). In this question you should find another LLM for code generation available on-line, for instance on HuggingFace [5] and perform few shot prompting for the 5 PUTs. Use only one few shot example. Describe what LLM you used and whether it generates better tests, than the previous LLM in terms of line and branch coverage. We recommend using a Google COllab for this question, as LLMs can be computationally expensive (10 points i.e. 2 points for each PUT). Save your implementation in "question3.py" file.

Important information :

1. The context window for the LLM is rather small (512 tokens i.e. words). Therefore, long prompts (more than 512 tokens) may not function as expected.
2. We recommend presenting the obtained branch and line coverage in tables..

5 Expected deliverables

The following deliverables are expected :

— Link to your repository with the implemented fuzzers or a .zip archive of the repository.

— The report for this practical work in the PDF format.

You should submit all the files to Moodle. When adding the PDF file, do not put into a zip. Additionally, you should add a ".txt" file with the following content : "Your full name, your student id" (this might be used for the grading automation). This assignment is individual.

The report file must contain the title and number of the laboratory, your name and student id.

5. `https://huggingface.co/models`

6 Important information

1. Check the course Moodle site for the file submission deadline
2. A delay of [0.24 hours] will be penalized by 10%, [24 hours, 48 hours] by 20% and more than 48 hours by 50%.
3. No plagiarism is tolerated (including ChatGPT). You should only submit code created by you.

Références

[1] Arghavan Moradi Dakhel, Amin Nikanjam, Vahid Majdinasab, Foutse Khomh, and Michel C Desmarais. Effective test generation using pre-trained large language models and mutation testing. *arXiv preprint arXiv :2308.16557*, 2023.

[2] Marija Selakovic, Michael Pradel, Rezwana Karim, and Frank Tip. Test generation for higher-order functions in dynamic languages. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA) :1–27, 2018.

[3] Koushik Sen, Darko Marinov, and Gul Agha. Cute : A concolic unit testing engine for c. *ACM SIGSOFT Software Engineering Notes*, 30(5) :263–272, 2005.

[4] Gordon Fraser and Andrea Arcuri. Evolutionary generation of whole test suites. In *2011 11th International Conference on Quality Software*, pages 31–40. IEEE, 2011.