



**CS547 – ADVANCED TOPICS IN SOFTWARE
ENGINEERING
ASSESSED COURSEWORK #2**

Group: CS547Assignment2Group18

Members:

1. HARSHAN RETHINAVELU SELVAKUMAR – 202480548 (Contribution – 50%)
2. MANOJ KUMAR DHARMARAJ – 202468855 (Contribution – 50%)

1 Review of related work

[1] Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S. and Yin, J. (2024). ChatUniTest: A Framework for LLM-Based Test Generation. doi:<https://doi.org/10.1145/3663529.3663801>.

Technology employed: ChatUniTest is used, a framework for automated unit test generation that leverages large language models (LLMs). ChatUniTest core, a common library that executes the primary workflow along with the ChatUniTest Toolchain (Chen et al., 2024).

Evaluation Strategy: EvoSuite and TestSpark are two representative tools used to establish baselines for comparison in the evaluation of ChatUniTest. ChatUniTest surpassed the performance of EvoSuite and TestSpark by attaining a reliable line coverage across a variety of projects (Chen et al., 2024).

Strengths: ChatUniTest follows a generation-validation-repair to address errors in the produced unit tests (Chen et al., 2024).

Limitations: Does not support a variety of programming languages, so that the preparation and validation can be broaden to accommodate more programming languages (Chen et al., 2024).

[2] Alshahwan, N., Chheda, J., Finogenova, A., Beliz Gokkaya, Harman, M., Harper, I., Marginean, A., Sengupta, S. and Wang, E. (2024). Automated Unit Test Improvement using Large Language Models at Meta. doi:<https://doi.org/10.1145/3663529.3663839>.

Technology employed: Meta's TestGen-LLM tool is used, which leverages large language models (LLMs) to enhance existing tests that are human-written (Alshahwan et al., 2024).

Evaluation Strategy: The experiment on Instagram Reels and Stories produced that 75% of test classes are built correctly, 57% of test classes are built correctly and passed reliably, 25% of test classes are built correctly, passed as well as enhances the line coverage compared to all other test classes (Alshahwan et al., 2024).

Strengths: TestGen-LLM operates in collaboration with human engineering efforts, insights and domain knowledge rather than replacing them. It follows an ensemble-style learning approach (Alshahwan et al., 2024).

Limitations: Does not include other test improvement criteria like Mutation Coverage and further research is necessary to establish application-specific techniques for converting the probability distribution of LLM into executable code (Alshahwan et al., 2024).

[3] Schäfer, M., Nadi, S., Eghbali, A. and Tip, F. (2023). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering, [online] pp.1–21. doi:<https://doi.org/10.1109/tse.2023.3334955>.

Technology employed: TestPilot, an adaptive tool is used for test generation based on LLMs specifically designed for JavaScript. This tool automatically produces unit tests for methods in a specified project's API (Schäfer et al., 2023).

Evaluation Strategy: TestPilot is evaluated on 25 npm packages with a total of 1684 API functions which shows that the generated tests achieve a median statement coverage of 70.2% and branch coverage of 52.8% (Schäfer et al., 2023).

Strengths: TestPilot does not necessitate fine-tuning or a parallel corpus of functions and tests. Rather, it incorporates contextual information regarding the function under test into the prompt (Schäfer et al., 2023).

Limitations: Focuses on JavaScript, which limits the use of other programming languages (Schäfer et al., 2023).

[4] Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y. and Peng, X. (2023). No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. [online] arXiv.org. Available at: <https://arxiv.org/abs/2305.04207>.

Technology employed: ChatTESTER, an innovative approach to unit test generation that utilizes ChatGPT to enhance the quality of the tests it produces (Yuan et al., 2023).

Evaluation Strategy: According to the results of the evaluation, ChatTESTER significantly shows improvement in both the compilation rate and passing rates of the generated tests compared to the default ChatGPT (Yuan et al., 2023).

Strengths: ChatTESTER incorporates an iterative test refiner which subsequently addresses the compilation errors in the tests produced by the initial test generator through a process of repeated refinement (Yuan et al., 2023).

Limitations: The threat lies in the randomness in ChatGPT as ChatTESTER leverages ChatGPT (Yuan et al., 2023).

[5] Guilherme, V. and Vincenzi, A. (2023). An initial investigation of ChatGPT unit test generation capability. doi:<https://doi.org/10.1145/3624032.3624035>.

Technology employed: Open AI's gpt-3.5-turbo is used to evaluate the quality of Java unit tests (Guilherme et al., 2023).

Evaluation Strategy: For the evaluation, 33 unit tests were generated automatically to collect the code line coverage, mutation coverage and success rate of test execution. The results indicated that the Open AI LLM test set exhibited comparable performance across all aspects compared to the other existing automated java test generation tools (Guilherme et al., 2023).

Strengths: Incorporates fully automated test generation with no human intervention (Guilherme et al., 2023).

Limitations: Focussed on Java, thus limiting the exploration to other programming languages (Guilherme et al., 2023).

2 Choice of LLM

Model: CodeT5+

Name: Salesforce CodeT5-P (770M)

2.1 Characteristics and features

- CodeT5+ represents a series of large language models specifically designed to tackle various tasks related to code understanding and generation (Wang et al., 2023).
- The framework encompasses a varied mixture of pretraining objectives that utilize both unimodal and bimodal data (Wang et al., 2023).
- CodeT5+ is an encoder-decoder based model (Wang et al., 2023).
- Despite an encoder-decoder based model, CodeT5+ demonstrates flexibility by functioning effectively in encoder-only, decoder-only and encoder-decoder configurations, thereby accommodating a variety of downstream applications (Wang et al., 2023).
- CodeT5+ is pretrained by using a variety of tasks such as span denoising, causal language modelling, contrastive learning and text-code matching to acquire comprehensive representations from both unimodal and bimodal code-text data (Wang et al., 2023).

2.2 Rationale for choice

- CodeT5+ is readily available and accessible in Huggingface as it is an open-source model. So it is easy to access and use.
- The model is effectively balanced between its size and performance, making it an appropriate choice for evaluation as the hardware resources are constrained.
- The model is specifically trained for code purposes.

3 Choice of Task

3.1 Task Definition

The task is to generate unit tests for a python function designed to calculate the factorial of a given number. The code defines a function `fact(a)` that computes the factorial of the integer `a`:

```
def fact(a):  
    if a==0:  
        return 1  
    else:  
        return a*fact(a-1)
```

Figure 1 Python function for factorial

3.2 Rationale for the choice of task

The python function for computing factorial is a basic and simple mathematical operation, making it suitable for assessing the efficiency of the model to produce accurate unit tests. The straightforward nature of this task enables to evaluate the capacity of the model to understand the function and generate appropriate unit tests.

Rather than focussing on complex modifications of the code, the task completely focuses on the testing and the verification of the code by generating unit tests. These generated unit tests are used to test whether the function works properly and provides the desired output. This is essential in the software development cycle.

Also, the task aims to reduce the manual efforts and errors in the automated generation of unit tests for a function.

4 Report on the results

The Salesforce CodeT5-P (770M) model was readily available and accessible via Hugging Face. This model was incorporated into a Google Colab notebook, where it was run on a CPU. The Google colab facilitated the efficient execution of the model with minimal configuration needed, making it a suitable environment for testing. The incorporation of the model in the Google Colab is as follows:

```
!pip install transformers
from huggingface_hub import login
login(token="hf_czi0QctzSycjf0VKvgPzVXqjRSvwsIAKbb")

from transformers import AutoTokenizer, AutoModelForSeq2SeqLM
import torch

checkpoint = "Salesforce/codet5p-770m-py"
device = torch.device('cpu')

tokenizer = AutoTokenizer.from_pretrained(checkpoint)
model = AutoModelForSeq2SeqLM.from_pretrained(checkpoint, trust_remote_code=True).to(device)
print(model.get_memory_footprint())
prompt="""
#Python function to find factorial of a number
def fact(a):
    if a==0:
        return 1
    else:
        return a*fact(a-1)

#Write unit tests for the above python function 'fact' using unittest.
"""
input=tokenizer(prompt,return_tensors="pt").to(device)
input['input_ids']=input['input_ids'].clone()
result=model.generate(**input,max_length=512,num_return_sequences=1,do_sample=True,temperature=0.7,top_p=0.9,top_k=50)
code=tokenizer.decode(result[0],skip_special_tokens=True)

print(code)
```

Figure 2 CodeT5-P (770M) model in Google Colab

4.1 Prompt

The prompt that was given for generating unit tests for the fact function is as follows:

```
#Python function to find factorial of a number
def fact(a):
    if a==0:
        return 1
    else:
        return a*fact(a-1)

#Write unit tests for the above python function 'fact' using unittest.
```

Figure 3 Prompt for generating unit tests for fact function

The provided prompt is to generate unit tests for the python function that finds the factorial of a number by using the unittest module in python.

4.2 Output of the generated unit tests

```
import unittest

class TestFact(unittest.TestCase):
    def test_fact_1(self):
        self.assertEqual(fact(1),1)

    def test_fact_2(self):
        self.assertEqual(fact(2),2)

    def test_fact_3(self):
        self.assertEqual(fact(3),6)

    def test_fact_4(self):
        self.assertEqual(fact(4),24)

    def test_fact_5(self):
        self.assertEqual(fact(5),120)

if __name__ == "__main__":
    unittest.main()
```

Figure 4 Generated Unit tests

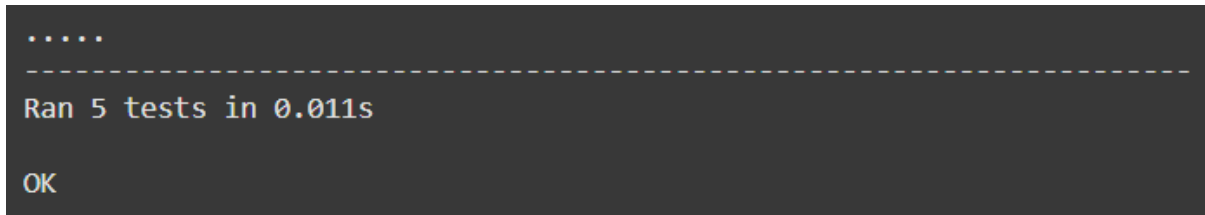
The generated unit tests ensures that the function operates accurately for fundamental positive integer inputs. This demonstrates the effectiveness of the model in the basic recursive implementation of the factorial function.

The generated test cases explicitly encompass the range of values from a=1 to a=5, which are frequently used in the evaluation of factorial function.

The assertEquals() serves for the comparison between the expected result and the actual result produced by the fact() function for a specified function. The generated unit tests are accurate as $1! = 1$, $2! = 2 \times 1 = 2$, $3! = 3 \times 2 \times 1 = 6$, $4! = 4 \times 3 \times 2 \times 1 = 24$ and $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$.

5 Analysis of the results

The generated unit tests ran successfully and the results of the test execution were as follows:



```
.....  
-----  
Ran 5 tests in 0.011s  
  
OK
```

Figure 5 Output of the test execution

The tests were executed successfully without any errors or failures. All the five test cases were carried out accurately and completed in a very short period of time (0.011 seconds). The confirmation message 'OK' verifies that all assertions within the unit tests were successfully validated.

5.1 Where it worked and Where it didn't

The unit tests were generated accurate for the small and positive integers within a very short period of time. Over a multiple runs, the test cases generated were repetitive which limits the model's capability to generate new test cases further. Sometimes, the unit tests that were generated for large integers are not precise.

5.2 Strengths

The unit tests successfully verified the functionality of the fact() function with respect to the specified inputs. The results were produced without any errors or failures, which concludes that the factorial function is performing as intended for the values assessed.

The execution duration of 0.011 seconds is very fast, which indicates that both the tests and the factorial function are being processed with high efficiency.

The unit tests that were generated concentrated on simple and straightforward cases, thereby facilitating the interpretation of the results.

5.3 Limitations

The generated unit tests did not encompass negative values, which should ideally trigger an error. Also the unit tests did not address non-integer inputs such as strings or float numbers. The recursive design of the `fact()` function may result in a `RecursionError` with invalid inputs.

The generated unit tests were confined to minimal values of `a`, specifically ranging from 1 to 5.

5.4 Recommendations for future use

Including unit tests for negative integers and non-negative inputs to evaluate the function's behaviour in these cases. The function should ideally manage such inputs effectively by raising exceptions. eg., `fact(-2)` or `fact(str)`.

Implementing unit tests for large values of `a`, for instance `fact(500)`, to assess the performance and scalability of the recursive implementation. This approach helps to identify any possible performance limitations to recursion depth.

Including edge cases such as `fact(0)` and verify that the function processes this input accurately, as it is expected to return 1.

References

- [1] Chen, Y., Hu, Z., Zhi, C., Han, J., Deng, S. and Yin, J. (2024). ChatUniTest: A Framework for LLM-Based Test Generation. doi:<https://doi.org/10.1145/3663529.3663801>.
- [2] Alshahwan, N., Chheda, J., Finogenova, A., Beliz Gokkaya, Harman, M., Harper, I., Marginean, A., Sengupta, S. and Wang, E. (2024). Automated Unit Test Improvement using Large Language Models at Meta. doi:<https://doi.org/10.1145/3663529.3663839>.
- [3] Schäfer, M., Nadi, S., Eghbali, A. and Tip, F. (2023). An Empirical Evaluation of Using Large Language Models for Automated Unit Test Generation. IEEE Transactions on Software Engineering, [online] pp.1–21. doi:<https://doi.org/10.1109/tse.2023.3334955>.
- [4] Yuan, Z., Lou, Y., Liu, M., Ding, S., Wang, K., Chen, Y. and Peng, X. (2023). No More Manual Tests? Evaluating and Improving ChatGPT for Unit Test Generation. [online] arXiv.org. Available at: <https://arxiv.org/abs/2305.04207>.
- [5] Guilherme, V. and Vincenzi, A. (2023). An initial investigation of ChatGPT unit test generation capability. doi:<https://doi.org/10.1145/3624032.3624035>.
- [6] Wang, Y., Le, H., Gotmare, A.D., Bui, N.D.Q., Li, J. and Hoi, S.C.H. (2023). *CodeT5+: Open Code Large Language Models for Code Understanding and Generation*. [online] arXiv.org. doi:<https://doi.org/10.48550/arXiv.2305.07922>.
- [7] Huggingface.co. (2023). Salesforce/codet5p-770m-py · Hugging Face. [online] Available at: <https://huggingface.co/Salesforce/codet5p-770m-py> [Accessed 1 Dec. 2024].