BEACHELOR THESIS

KOGNITIVE INFORMATIK

# SYSTEMATIC GENERATION OF UNIT-TESTS WITH DIFFERENT LLMS AND PROMPT-DESIGN

RICHARD PAMIES

*University of Bielefeld*

SUPERVISED BY

PROF. DR.-ING. BENJAMIN PAASSEN,

M.SC. JESPER DANNAT

JULY 18, 2024

# CONTENTS

# 1 INTRODUCTION

Unit-test are really important and useful. But most often those tests don't get written because of laziness, so the automatic generation of those tests arise as a possible solution.
Large Language Models (LLMs) offer for the the possibility of creating useful unit-tests. But those generated test are flawed because those LLMs dont work with reason, but instead the statistically most probable answer.
Prior work has tried under different circumstances to solve this problem.[paper 1,2,3].
My goal in this work is to:

1. measure the code coverage and statement coverage of generated unit-test
2. generate these unit-tests with different LLMs
3. generate these unit-tests with different prompt design strategies

and then at the end i want to compare the different approaches.

# 2 BACKGROUND AND RELATED WORK

## 2.1 UNIT-TESTS

Unit-tests are tests written for already existing code to make sure the implemented functionalities still work after further work on the code is done. The usefulness of unit-tests is not controversial. Rather the time it takes to write them and the metrics used to test their usefulness. Code coverage and statement coverage are two well-known and often used metrics in the context of unit-tests. There are more metrics like the mutation-score but code and statement coverage are the most implemented and available. For all metrics concerning unit-test and above it is important to keep in mind that there a no metrics that can be used to guarantee bug-free code. It is more an indication of where more tests might be needed rather than making any statement about the quality and conceptual coverage that the tests provide for the actual code. So in the end there are no perfect, singular metrics that can be used to assess the quality and conceptual coverage of unit-tests. Experience and the combined use of several different metrics to arrive at an indication is the best way to create high-quality unit-tests.

### 2.1.1 *code coverage*

Code coverage is a measure of how much of a program's source code is executed during testing. It works by tracking which lines of code, functions, or branches are run when tests are performed.

### 2.1.2 *statement coverage*

Statement coverage on the other hand measures the percentage of executable statements in a program that are executed during testing. However, it has limitations as it doesn't account for different paths through decision points

### 2.1.3 *mutation coverage*

Mutation score is a measure of test effectiveness that assesses how well tests detect intentional code changes made by mutations. It works by automatically introducing small alterations to the source code, such as changing operators or variable values, and then running the test. The score is the percentage of mutations detected by failing tests. Mutation score is important because it evaluates the quality and sensitivity of tests, revealing weaknesses in test cases that other metrics might miss. A high mutation score suggests that tests are good at catching

potential bugs. However, generating and testing mutations can be computationally expensive.

### 2.1.4 *cyclomatic code complexity*

Cyclomatic complexity is a software metric that measures the number of linearly independent paths through a program's source code. It works by analyzing the control flow graph of a program, counting decision points like if statements and loops. A higher cyclomatic complexity indicates more complex code. This metric is important because it helps identify overly complicated functions or methods that may be difficult to test, maintain, or understand.

## 2.2 LLMS

Large Language Models(LLMs) are computational models that normaly used an enormous data set for learning and are able to achieve general-purpose language generation and other tasks like categorization and other multi-modal tasks. An LLM is only as good as its architecture and its training data. In the case of unit-tests Generating unit-tests for mainstream code might be sufficient. But it is really easy to write code which covers niche subjects. And in those cases the LLM would need critical thinking, understanding and reasoning to fully understand the code and write useful and encompassing unit-tests.

### 2.2.1 *GPT*

ChatGPT is an advanced conversational AI model developed by OpenAI. It works by using a large language model trained on diverse internet text to generate human-like responses to user inputs. ChatGPT can engage in dialogue, answer questions, and assist with various tasks across multiple domains.

### 2.2.2 *Open source models*

## 2.3 RELATED WORK

# 3 METHODS

This high level description what it should accomplish detailed description for reproduction for experienced user use illustrating diagrams as much as possible (no source code, but perhaps pseudo code) provide a good reason for each design choice(source or math) use a running example to illustrate the steps

## 3.1 LLMS

I used chat-gpt(GPT) 3.5, GPT 4o, OPEN$_S$*OURCEandOPEN*$_S$*OURCE.IusedGPTbecauseofitsaccessibilityan*
*tests*.

## 3.2 PROMPT-DESIGN

A strong leverage to control the resulting output quality of LLMs is prompt-design. Prompt-design describes the mostly iterative, model-specific and experience driven design process of how the textual input prompt is worded and structured to achieve the optimal result. In my experiments i validated my prompts with an example exercise and GPT 3.5. My prompt, that was send in the end to the model, consisted of four parts that were togglable:

1. My static initialization prompt, which instructed the model to generate a unit-test with the information's available.
2. A Task description, that was copied from the Task description and, if necessary, slightly adjusted.
3. The code of the example solution,
4. The function name and signature,
5. Unit-test examples of the example unit-test,
6. The error message in the case of failure, so that errors could in the best case be specifically addressed and fixed.

There were two main static prompts needed for my application:

- The instruction prompt, that combined with the dynamically read task attributes forms the prompt for the model.

  **Instruction prompt** The things above are the parts of the description of a programming task. Now write me a Unit test in python for to validate the code that was given above. Think of the imports. Be thorough.

  I was forced to re-design the prompt a few times because GPT 3.5 was suddenly forgetting, or misunderstanding important aspects.

Examples were that suddenly no code was generated, only a textual directive on how to write a unit-test.

- The role description, which describes the role of the model in the interaction. I mainly used the role description to control the output of the model.

  **Role description:** You are an outstanding programmer, but secluded and efficient. As a professional programmer you answer as concise and precise as possible without any unnecessary words

  This resulted in comparatively reliable results where the amount of unnecessary text was reduced.

## 3.3 TRAINING-DATA

I received 22 exercises which were exercises for the python entry course of Prof. Paaßen. Each of the 22 folders contained at least a Task description,an example unit-test and an example solution for the code.

## 3.4 PRE-PROCESSING

I prepared each of the 22 Tasks manually by creating a "prompt" text-file and pasting the information's that are necessary in defined categories. That enabled me to then parse the information's for each task programmatically.

## 3.5 POST-PROCESSING

When receiving the code i wrote a small parser that extracted the code from the resulting string.

## 3.6 VALIDATION

## 3.7 STRATEGIES FOR FAILURES

re-sending with the attached failure message removal of faulty test-line

The structured prompt could look like this:

**Lemma** Let the coefficients of the polynomial

$$a_0 + a_1 x + a_2 x^2 + \cdots + a_{m-1} x^{m-1} + x^m$$

be integers. Then any real root of the polynomial is either integral or irrational.

3.8 provide overview diagram illustrating my steps

### 3.8 provide overview diagram illustrating my steps

# 4 IMPLEMENTATION

## 4.1 FOR EACH OF THE STEPS HOW IT WAS IMPLEMENTED IN THE CODE

# 5 EXPERIMENTS

5.1 WHAT I WANT TO SHOW

5.2 HOW I WANT TO SHOW IT

5.3 BASELINES AND ASSUMPTIONS

5.4 WHAT WAS MY INITIAL ASSUMPTION

5.5 TABLE FOR EXPERIMENTS

5.6 DISCUSSION OF THE RESULTS

# 6 CONCLUSION

# 7 LATEX FEATURES

sparse[1]

Did you know? The discrete fourier transformation shown in equation 7.1 is the backbone of the modern information-society.

$$f_m \tag{7.1}$$

For words of science, see [**botsch2010polygon**]. Unfortunately, that book has nothing about wolves.

1 This is a footnote