Large Language Models – Based Test Case Generation

A Comprehensive Analysis of How AI is Revolutionizing Automated Testing

NIKHIL YATES

Automating Code Generation

AGENDA

- Background and Motivation
  - Code Automation
  - GPT-4
  - Pynguin
- Experiment Design
- Findings
  - The Data
  - Subjective Interpretations
- Discussion and Applications
- Q&A

Current General Exploratory Research

RANDOM TESTS VS COVERAGE-BASED TESTS

- Random approach to test case generation is stronger than the coverage-based approach
- Random approach benefits:
  - Simplicity
  - Minimizes bias
  - Performs well in contexts with large input sizes

EMPIRICAL STUDIES, LLMS

- LLM-based test generation tools are incredibly powerful
  - Tests are realistic
  - Tests could contain assertions
- LLM's are ***not*** expected to cover edge cases
  - Training data does not encompass non-typical use cases
- Transformer architecture drives high performance on sequential (textual) data
  - LLMs are better for generation regression tests

HUMAN-ORIENTED TEST GENERATION

- All studies complement the 'human' aspect of LLM-based tests
- Reports that include any non-empirical observations do not incorporate them directly into the study
  - Usually an additional statement/comment

LACK OF CRITICISM

- Critical observations about the relationship between empirical and non-empirical qualities of LLM-based tests
- Lack of critical research comparing qualitative attributes of traditional tools and LLM-based tools

Generative Pre-Trained Transformer 4

175

BILLION Parameters

Compared to 6 billion in GPT-3.5

- Out-of-box large language model that boasts accuracy, elevated understanding, sophisticated response abilities
- A new animal compared to its predecessors
- New tool = research opportunities

- General Research
  - Strong ability to contextualize input (transformer architecture)
  - Low edge-case inclusion in GPT-3.5
  - GPT-4 stresses the importance of random-case generation
  - Output quality:
    - Clear
    - Relevant
    - Limited to the training data
- Empirical Test Generation Study (*Shäfer et al.)*
  - Realistic
  - No expectation for edge-case coverage
  - Limited qualitative analysis

Pyguin

- Background
  - Unit test generation suite
  - First tool that generates unit tests for dynamically-types languages
  - Currently ***not*** a production-level tool
- Dynamic Generation Approach
  - Generates a basic test
  - Iteratively runs the program against the test and changes the test to improve code coverage
  - Mutants
- Performance limitations increase with codebase complexity
- Most popular *dynamic* test generation tool for python

Experiment Design

Experiment Overview

OBJECTIVE

- Assess the quality of the tests that the LLM-based tool produces
- Compare the LLM-based tool with the traditional tool
  - Performance differences
  - Non-performance differences

DATASET

- TensorFlow GitHub repository
- Rich, diverse data
- Strong and practical codebase

EVALUATION METRICS

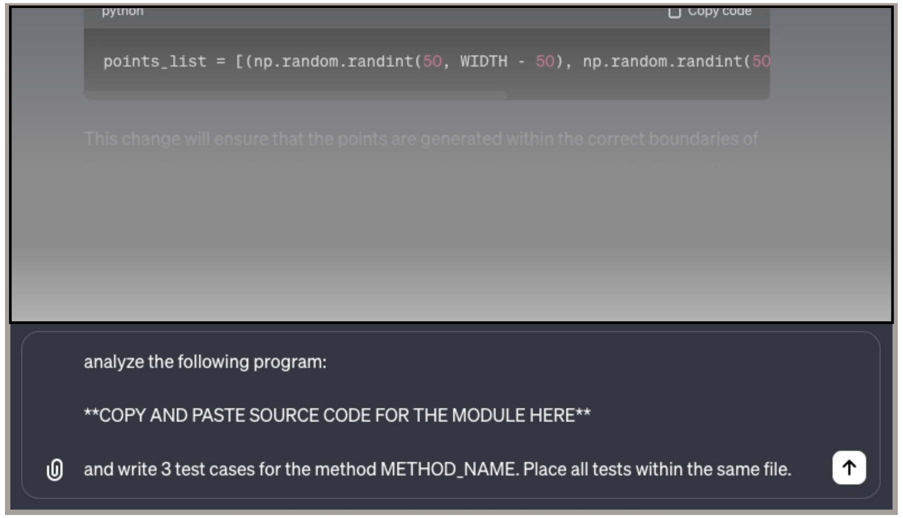
- Quantitative *and* Qualitative
- Filling the gaps of existing research
- Emphasis on the relationship between KPIs, not just face value

RESEARCH QUESTIONS

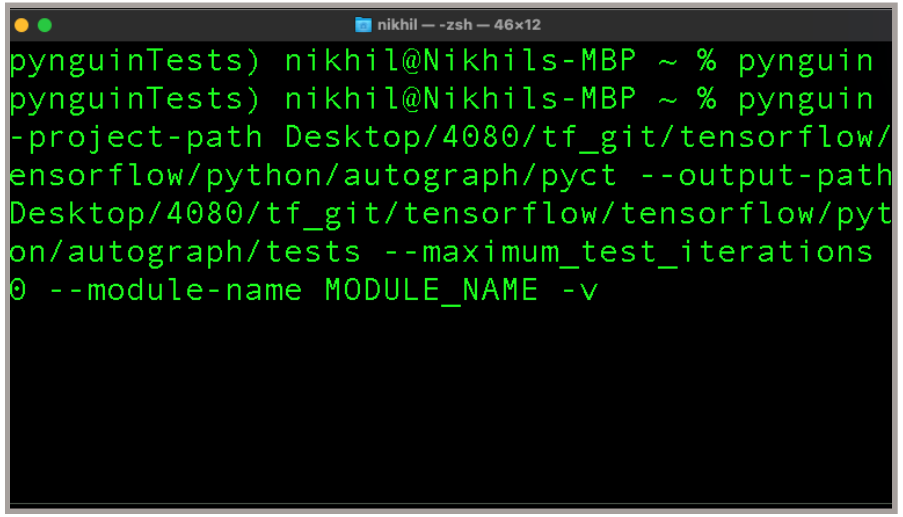
- To guide research and experimentation
- The *essence* of the project

Generation Process

- GPT-4
  1. Copy the entire python module into the system
  2. Method specification
  3. *pytest runnable*
  4. Repository compatibility*



- Pynguin
  1. Configuration analysis
  2. Standard execution
  3. Repository compatibility*

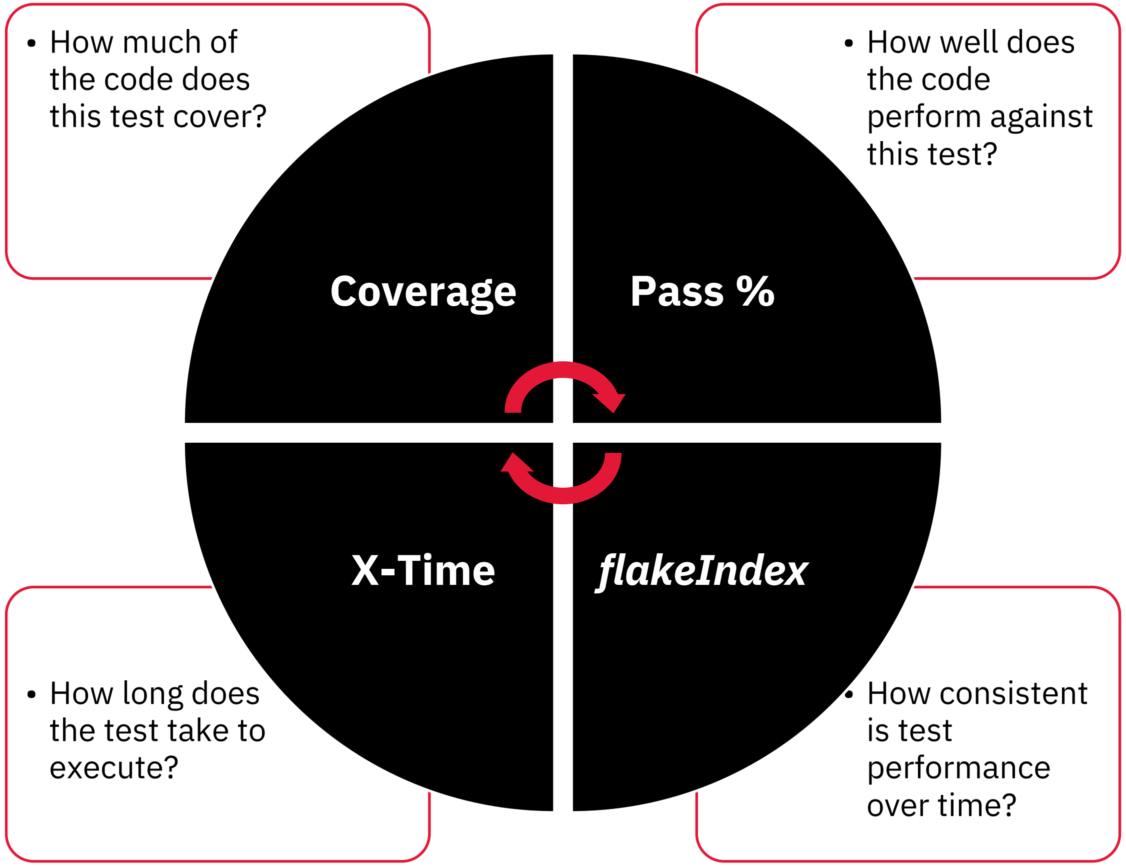


Dataset information

- TensorFlow/python/autograph/pyct
- 50 methods
- Diversity across the data:
  - methods type (instance, class, static methods)
  - Variable functionality
- Randomness Motivation:
  - Edge case generation
  - Manifested function: LLM training
    - Increased understanding of future code
    - Benchmarking
    - Validation
    - Data richness

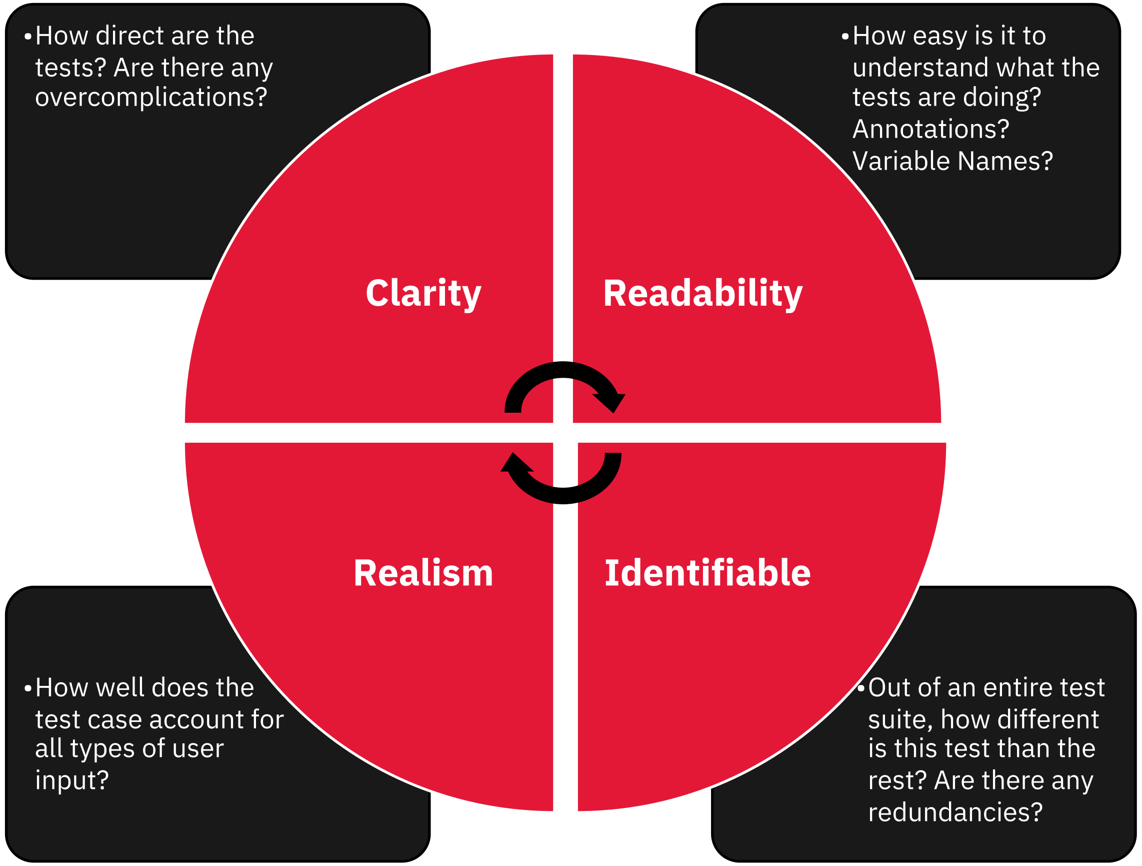
Quantitative Evaluation Metrics

- ***quantScore***
  - Summation of quantitative KPI scores
  - Each KPI is judged using a 2-point grading scale
  - */8
- ***flakeIndex***
  - Score card to assess the consistency of tests
- Quantitative KPIs are co-dependent
  - Specific tests
  - Function of the method



Qualitative Evaluation Metrics

- ***qualScore***
  - Summation of qualitative KPI scores
  - Each KPI is judged using a 2-point grading scale
  - */8
- **Unique nature** of qualitative KPIs
  - Subjectiveness
  - Independence
  - Extensiveness
- Qualitative KPIs are 'complete' but inconsistent
  - Structure of the tool's output
    - GPT-4 output can be extensively modified



Guiding Questions

RQ1

(Performance) How well does the LLM-based test generation tool perform with respect to the criteria for strong test cases?

RQ2

(Competence) To what extent does the traditional and LLM-based test generation tools differ quantitatively? Qualitatively?

Result Analysis

Initial Observations

- Precision Discrepancies

Tool	Generated Test Files	Generated Test Cases
Pynguin	5	33
GPT-4	50	150

- Experiment Integrity
  - 1:1 Comparisons

GPT-4 Quantitative Analysis – RQ1

Tool	Coverage	Pass %	Exec. Time (s)	Flakiness
GPT-4	76.4%	70%	6.2	0.0
	1.5/2 - Complex dataset - Rigidity of	**2/2** - NO xFail tests generated - Testing thoroughness	**1.8/2** - With respect to dataset complexity, complexity of tests, and test coverage	**2/2** - Consistency over 3 rounds of testing

quantScore = 7.3

GPT-4 Qualitative Analysis – RQ1

Tool	Clarity	Readability	Realism	Identifiable
GPT-4	2	1.7	1.9	2
	▪ Tests were clear and direct ▪ The tests did not do unnecessary work (i.e., irrelevant testing)	▪ Well annotated ▪ Appropriate naming conventions ▪ Some test files contained minimal redundant comments	▪ Very accurate in terms of actual use case ▪ Some cases did not accommodate arbitrary user input	▪ No ”over-testing” ▪ LLM did not generate tests that were similar to previously generated tests ▪ All test cases were distinct

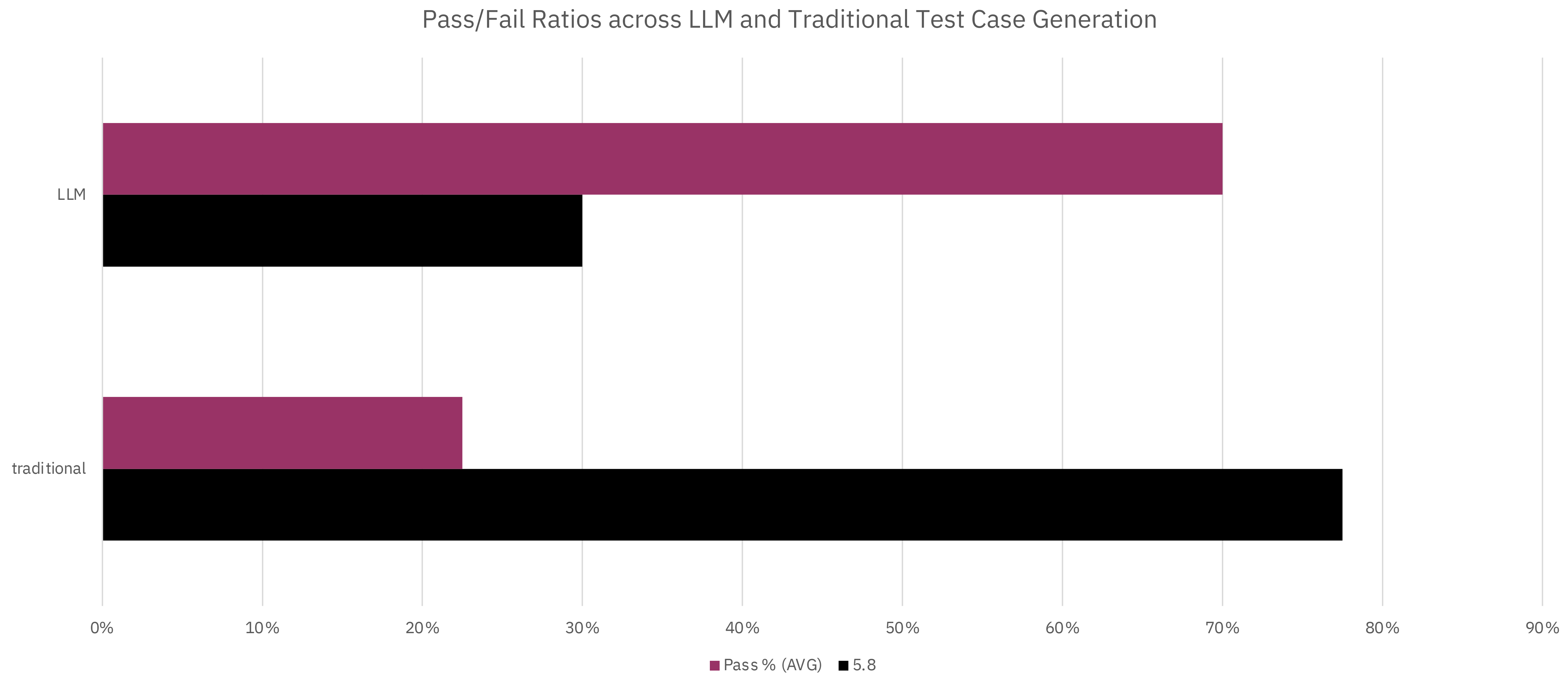
qualScore = 7.6

Comparing Tools – RQ2

QUANTITATIVE COMPARISON					
Tool	Coverage	Pass%	Time (s)	Flakiness	*quantScore*
GPT-4	76.4%	70%	6.2	0.0	7.3
Pynguin	64.2%	21%	5.7	0.1	5.2

QUALITATIVE COMPARISON					
Tool	Clarity	Readability	Realism	Identifiable	*qualScore*
GPT-4	2	1.7	1.9	2	7.6
Pynguin	0.9	1.1	1.5	1.2	4.7

Pass Frequency



LASSONDE SCHOOL OF ENGINEERING | YORK U

Examples

```python
import pytest
import gast
from tensorflow.python.autograph.pyct.ast_util import parallel_walk

class TestParallelWalk:

    # Test Case 1: Parallel walk with identical ASTs
    def test_identical_asts(self):
        node1 = gast.parse("a = 1 + 2")
        node2 = gast.parse("a = 1 + 2")
        for n1, n2 in parallel_walk(node1, node2):
            assert type(n1) == type(n2), "Nodes should be of the same type"

    # Test Case 2: Parallel walk with different ASTs
    def test_different_asts(self):
        node1 = gast.parse("a = 1 + 2")
        node2 = gast.parse("b = 3 + 4")
        with pytest.raises(ValueError):
            for _ in parallel_walk(node1, node2):
                pass

    # Test Case 3: Parallel walk with different structures
    def test_different_structure(self):
        node1 = gast.parse("a = 1 + 2")
        node2 = gast.parse("for i in range(10): a = i")
        with pytest.raises(ValueError):
            for _ in parallel_walk(node1, node2):
                pass

# If you wish to run the tests directly using this script
if __name__ == "__main__":
    pytest.main()
```

```python
# Test cases automatically generated by Pynguin (https://www.pynguin.eu).
# Please check them before you use them.
import pytest
import tensorflow.python.autograph.pyct.inspect_utils as module_0
import inspect as module_1
import tokenize as module_2
import ast as module_3


def test_case_0():
    none_type_0 = None
    var_0 = module_0.isconstructor(none_type_0)
    assert var_0 is False


@pytest.mark.xfail(strict=True)
def test_case_1():
    none_type_0 = None
    var_0 = module_0.islambda(none_type_0)
    module_0.getqualifiedname(none_type_0, none_type_0, none_type_0)


def test_case_2():
    none_type_0 = None
    var_0 = module_1.istraceback(none_type_0)
    var_1 = module_0.isnamedtuple(var_0)
    assert var_1 is False


@pytest.mark.xfail(strict=True)
def test_case_3():
    bool_0 = True
    var_0 = module_0.isbuiltin(bool_0)
    assert var_0 is True
    var_1 = module_2.group()
    module_0.getnamespace(var_1)


@pytest.mark.xfail(strict=True)
def test_case_4():
    none_type_0 = None
    var_0 = module_0.getfutureimports(none_type_0)
    var_1 = module_0.isbuiltin(var_0)
    assert var_1 is False
    var_2 = module_0.isnamedtuple(none_type_0)
    assert var_2 is False
    module_0.getqualifiedname(none_type_0, none_type_0, none_type_0, none_type_0)

```

Conclusion

RQ1

(Performance) How well does the LLM-based test generation tool perform with respect to the criteria for strong test cases?

RQ2

(Competence) To what extent does the traditional and LLM-based test generation tools differ quantitatively? Qualitatively?

Applications and Discussion

Further Research and Applications

- Project focused on developing test cases for the python code base:
  - What about a statically typed language like Java?
- On 50 test cases from the tensorflow/java/ directory:
- Strong *quantScore* of 7.1 and *qualScore* of 7.5
- Comparison against a traditional unit-test generation tool like Randoop, Palus?

- The benefits of test-driven code development
  - Robustness
  - Comprehension
  - Validity
  - Transformation of code
  - Collaboratory benefits
  - And more!
- Test-drive development in schools
  - A meaningful piece of criteria

Questions

Thank You!

Acknowledgements to Prof. Song Wang