COSC 3P99 – Project Course
Software Testing with Large Language Models: Semantic Coverage

Rouvin Personal Report

Name: Rouvin Rebello
Student ID: 7098080
Email: rr20jk@brocku.ca
Github: https://github.com/Rouvin-rebello/COSC_3P99
Supervisor: Naser Ezzati-Jivan

COSC 3P99
Software Testing with Large Language Models: Semantic Coverage

Project

Created a framework to assess semantic coverage in various projects.

Semantic Coverage Definition

Semantic coverage assesses the alignment between the tests and the code's core objectives, ensuring thoroughness and relevance of testing in a software project. It ensures that testing efforts are focused on the most critical aspects of the code, thereby improving the effectiveness and reliability of the testing process.

Literature Review

General Understanding of Software Testing

Exploring the Practical Applications of Large Language Models in Software Testing:

- **Report:** Large Language Models (LLMs) are increasingly utilized in software testing for tasks such as test case preparation, bug reporting, and bug fixing, due to their advanced capabilities in understanding and analyzing code. This enhances the efficiency and accuracy of software testing processes, ensuring that software meets high quality standards before release.
- The software testing life cycle is a dynamic process involving several key activities: requirements analysis to understand testing needs, test planning to outline strategies and schedules, test case design to cover various scenarios, test execution to identify defects, regression testing to check for new issues from recent changes, and detailed bug reporting to help developers fix issues effectively.
- Prominent LLMs in software testing include Codex, known for its diverse code training; ChatGPT, used for generating data inputs and conversational debugging; and CodeT5, effective in test case generation and bug fixing. These models significantly improve traditional testing methods by automating repetitive tasks, providing sophisticated code analysis, and facilitating effective bug resolution.

Comprehensive Study of Software Testing

- **Report:** Manual testing, also known as static testing, follows a systematic process beginning with understanding requirement documentation to ensure alignment with the intended functionalities. This is followed by writing test cases, which guide the testing process through various functions and scenarios within the software product or application. Afterward, the tests are conducted while keeping meticulous track of the success rates. Any identified bugs are then logged with detailed information to be passed to the development team. The final step involves creating a comprehensive test report to document the findings and provide a clear overview of the testing outcomes.

- In software testing, several strategies are employed to ensure the robustness and reliability of the product. Unit testing is the simplest form of testing, focusing on the smallest testable parts of the software, known as units. Techniques such as functional testing and structural testing are used to ensure each unit performs as expected. Integration testing follows, combining two or more tested units into an integrated structure to verify that they work correctly together, ensuring seamless interaction between different parts of the software.
- System testing is the next strategy, where all individual units are combined and tested as a whole system. This comprehensive testing ensures that the entire software system functions correctly when all components are integrated. The final stage is acceptance testing, which evaluates whether the developed product meets the specified criteria and requirements provided by the customer or client. This quality assurance step ensures that the software is ready for deployment and satisfies the end-user's needs and expectations.

Software Testing with Large Language Models: Survey, Landscape, and Vision

- **Report:** The challenges of software testing, particularly in automated test case generation, persist despite various methods like search-based, constraint-based, or random-based techniques. Unit testing, crucial in software development, involves a method under test with a corresponding unit test comprising a test prefix and test oracle. The test prefix drives the method to a testable state, while the test oracle checks if the method's behavior aligns with expectations. Unit tests can be generated effectively through prompts or pre-training models with various learning methods like zero-shot or one-shot learning. In evaluating unit tests, factors such as coverage, correctness, and relevance to developers, users, or quality assurance specialists are significant. Notably, Codex - HumanEval performed cabest, followed by ChatGPT - CodeSearchNet in generating unit tests.

Understanding Unit Tests:

The Role of Testing Professionals in Unit Testing

- **Report:** Unit testing focuses on assessing small software units like functions or methods to ensure their individual accuracy and functionality. These tests prioritize simplicity, isolation, automation, repeatability, and comprehensive coverage to uphold system quality. Structured-based testing involves creating unit tests for each code statement and meticulously examining conditions and loops to achieve thorough coverage. Path-based testing requires crafting tests for all potential code execution paths, thoroughly inspecting each conditional structure to cover all possible routes, thus enhancing test coverage and code resilience. Requirement-based testing aligns unit testing with software specifications, determining test scenarios based on typical use cases to ensure comprehensive coverage.

Unit Testing and Coding: Why Testable Code Matters.

- **Report:** A unit test is a method that tests a small portion of an application independently from other parts. It typically involves three phases: initializing the system under test, applying stimuli to the system, and observing the outcomes. In

state-based unit testing, the focus is on verifying that the system produces correct results or maintains the correct state. When writing unit tests, it's essential to ensure they are easy to code, readable, fail only due to bugs in the system under test, fast for repeated runs, and independent of integration with other units.

Properties of Good Unit Tests for Software Quality Assurance

- **Report:** Good tests possess several key properties, collectively referred to as A-TRIP. Firstly, they are Automatic, meaning they run without human intervention, often continuously on checked-in code. These tests must autonomously determine their pass or fail status, with smaller unit tests aiding in issue localization. It's essential for unit tests to be automatically invoked and results checked in at least two ways. Secondly, they are Thorough, testing every aspect of the system, including lines of code, branches, exceptions, and critical error-prone areas like boundary conditions and malformed data. Test coverage, spanning code, scenarios, and specifications, is
- Moreover, good tests are Repeatable, consistently yielding the same results regardless of how often they are run. This repeatability ensures reliability and stability in the testing process. Additionally, they are Independent, functioning autonomously from the environment and each other. Each test should focus on testing one specific aspect, method, or feature, without relying on other tests to have been executed beforehand. This independence promotes flexibility and robustness in testing procedures.
- Furthermore, good tests adhere to Professional standards of software design, maintaining principles like encapsulation, the DRY (Don't Repeat Yourself) principle, and low coupling. Following these design rules ensures clarity, maintainability, and scalability of the testing framework. Considering the Cost of tests, the efficiency of tests directly impacts programmer productivity, emphasizing the importance of fast-running tests for long-term cost reduction and overall project success.

Detecting Code Smells in Python Programs

- **Report:** To identify test smells, we adopt a test smell detection tool called tsDetector because it can detect a comprehensive list of test smells (i.e., 18 test smells in total)

The secret life of test smells - an empirical study on test smell evolution and maintenance

- **Report:** Early approaches to detecting code smells often rely on manual methods, which can be inefficient. Automated tools like Pysmell improve this by consistently analyzing and detecting code smells in large codebases. Pysmell identifies 11 types of code smells in Python programs, including Large Class (LC), Long Parameter List (LPL), and Long Method (LM). It has demonstrated high effectiveness, finding 285 instances of code smells with an average precision of 97.7

Which Static Code Metrics Can Help to Predict Test Case Effectiveness? New Metrics and Their Empirical Evaluation on Projects Assessed for Industrial Relevance

- **Report:** The objective of the study was to develop a model using code quality metrics to estimate test effectiveness without relying on mutation testing. Metrics like Cyclomatic Complexity and Lines of Code (LOC) were calculated to identify complex or lengthy test methods. PIT was used to compute mutation scores, applying 13 mutation operations. The study considered 67 code quality factors across five dimensions: Code Coverage, Test Smells, Code Metrics, Code Smells, and Readability.

PREDICTING AND ESTIMATING EXECUTION TIME OF MANUAL TEST CASES-A CASE STUDY IN RAILWAY DOMAIN

- **Report:** Due to resource constraints in software testing, selecting a cost-effective subset of test cases from a larger test suite is crucial. Criteria such as execution time, cost, requirement coverage, and test case dependencies play a vital role in this selection process, with execution time being a significant factor in the overall expense of testing. Accurately predicting and estimating test case execution time aids in efficient test planning, reducing costs. This paper leverages historical data and natural language processing (NLP) to predict the execution time of manual test cases, improving the efficiency of the software testing process.

Topic modelling:

Topic modeling in software engineering research

- **Report:** Topic modeling is a powerful text mining technique used to extract coherent word clusters, or topics, from large collections of textual documents, helping to uncover hidden semantic structures within the text. The process begins by converting the text data into a structured vector-space model, typically involving pre-processing steps like tokenization, stemming, and the removal of stop words. Various models are then applied to identify topics, including Latent Dirichlet Allocation (LDA), which extracts human-readable semantic topics, Latent Semantic Indexing (LSI), which leverages word frequencies and co-occurrences, Probabilistic Latent Semantic Indexing (pLSI), and Singular Value Decomposition (SVD). Each of these methods offers a unique approach to analyzing and revealing the underlying concepts in textual data.

A Semantic Framework for Test Coverage

- **Report:** Coverage measures in black-box testing focus on the number of states and transitions visited, while white-box testing examines the number of statements, branches, and paths executed. However, these approaches have limitations: they are based on syntactic features, which may vary with different but behaviorally equivalent models, and they do not account for the criticality of failures. The semantic coverage approach addresses these issues by incorporating a weighted fault model, which assigns importance to potential errors, and fault automata, which represent these errors in a formal structure. This model uses error weight assignments and includes finite depth and discounted weighted fault models to manage coverage and prioritize significant errors. Optimization algorithms are employed to compute

minimal test suites with maximal coverage, and experimental validation shows the approach's feasibility for small protocols, with potential for larger systems.

Formal Semantics of Programming Languages - An Overview

- **Report:**Semantics in programming languages models the computational meaning of programs by focusing on the relevant aspects of execution while ignoring irrelevant details. It includes several levels: Static Semantics checks for type correctness and scope rules before runtime to ensure error-free code at compile time. Dynamic Semantics examines the program's behavior during execution, including state changes and memory allocation. Operational Semantics describes how a program executes through step-by-step state transitions. Denotational Semantics maps language constructs to mathematical objects, representing their meaning independently of specific implementations. Axiomatic Semantics uses logical assertions to model the relationship between pre-conditions and post-conditions. Structural Operational Semantics offers a simple approach to semantic descriptions based on basic mathematics, while Abstract State Machine Semantics specifies computation steps at an abstract level, minimizing focus on control flow and bindings. Action Semantics integrates elements of both denotational and operational semantics.

Learning Deep Semantics for Test Completion

- **Report:** Code semantics refers to information about test or code execution that goes beyond syntax-level data. To address the challenge of writing test methods more efficiently, TeCo was developed, introducing a novel task called test completion. This approach leverages code semantics and execution data to enhance machine learning models for code-related tasks. TECO, the first transformer model trained on extensive code semantics data, excels in test completion and reranking based on execution results, emphasizing the importance of code semantics in accurately modeling test method execution. Static analysis has limitations, such as inaccuracies when values are unknown without code execution, leading to potential overestimations. Code semantics can be categorized into execution results, which involve local variable types and field initialization, and execution context, which includes setup and teardown methods, the last called method, and relevant statements in non-test code.

Optimal test case generation for boundary value analysis

- **Report:** The paper introduces Boundary Coverage Distance (BCD), a metric that assesses the extent to which a test set covers the boundaries of software execution paths. Test coverage evaluates how well test cases cover different execution paths in software, using criteria such as statement coverage (C0), path coverage (PC), branch coverage (BC), decision coverage (DC), decision/condition coverage (D/CC), and modified condition/decision coverage (MC/DC). This study emphasizes evaluating test coverage through Boundary Value Analysis (BVA) by defining and utilizing the BCD metric.

Coverage methods:

Testing Coverage Functions – function coverage

- **Report:** Calculates the total weight of elements in the union of selected subsets from a universe of weighted elements. For any subset of labels from 1 to m, it sums the weights of elements in the corresponding subsets. Coverage functions are a key type of submodular functions, often used in applications like utility functions in combinatorial auctions.

Analysis of Statement Branch and Loop Coverage in Software Testing With Genetic Algorithm – Branch Coverage

- **Report:** This criterion states that you must write enough test cases that each decision has a true and a false outcome at least once. In other words, each branch direction must be traversed at least once

Implementation

- **Testing code coverage:**
  - Ran GitHub Project 1 (source code and test cases) using Coverage
  - Report Generated showing 78% coverage

```
PS C:\Users\Rouvin\PycharmProjects\pythonProject\Projects\11_Brock_COSC_3P99\Metric_testing\Coverage> python -m coverage report
Name                            Stmts   Miss  Cover
---------------------------------------------------
Shopping_cart_project.py           60     16    73%
Shopping_cart_project_tests.py     70     13    81%
---------------------------------------------------
TOTAL                             130     29    78%
PS C:\Users\Rouvin\PycharmProjects\pythonProject\Projects\11_Brock_COSC_3P99\Metric_testing\Coverage>
```

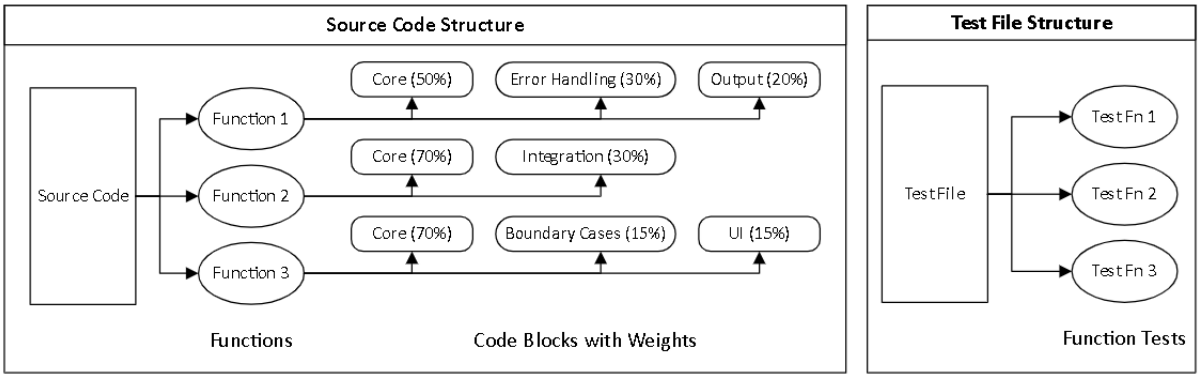
- **Created and tested Statement, Branch and Function coverage scripts.**
  - **Results:**
    - Statement Coverage: 57.14% (Score: 5.71)
    - Branch Coverage: 61.21% (Score: 6.12)
    - Function Coverage: 73.33% (Score: 7.33)
    - Traditional Coverage Score: 6.38
- **Created a Ranking System that compares:**
  - **Semantic coverage:**
    - Mean
    - Median
    - Standard Deviation
    - Min

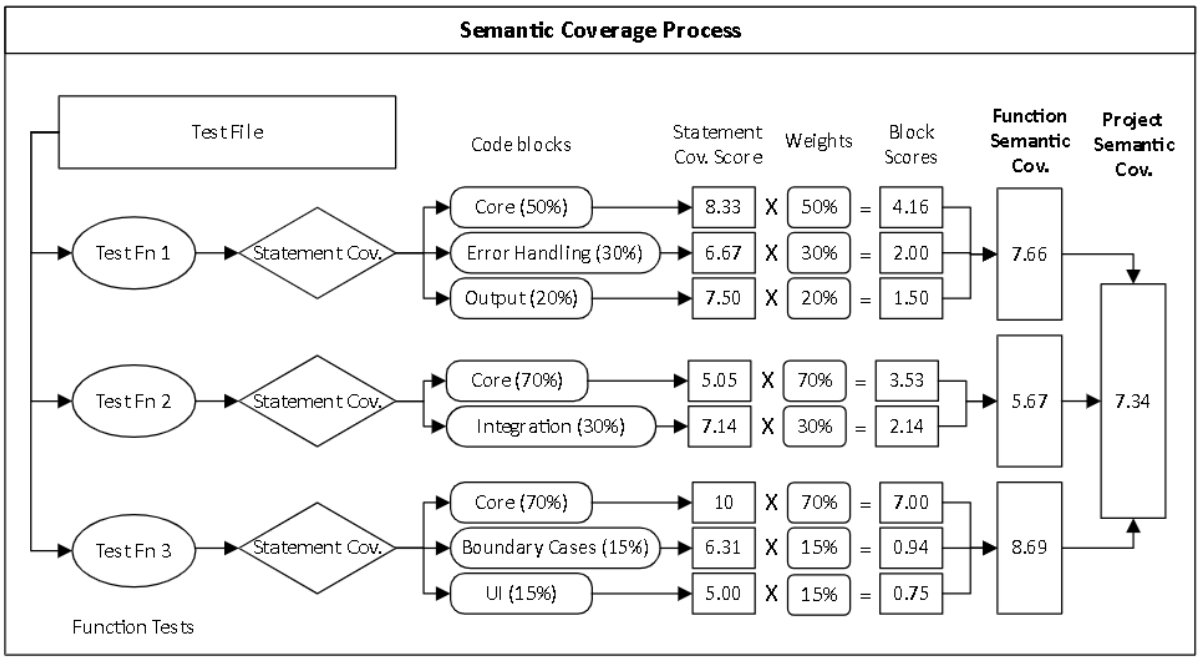
    - Max
  - **Coverage methods:**
    - Statement Coverage
    - Branch Coverage
    - Function Coverage
  - **Manual Testing Methods:**
    - Code Metrics
    - Smell tests
    - Static analysis
    - Test Execution time
  - Analysis Sheet (Summary)
- **Created and evaluated Manual testing methods:**

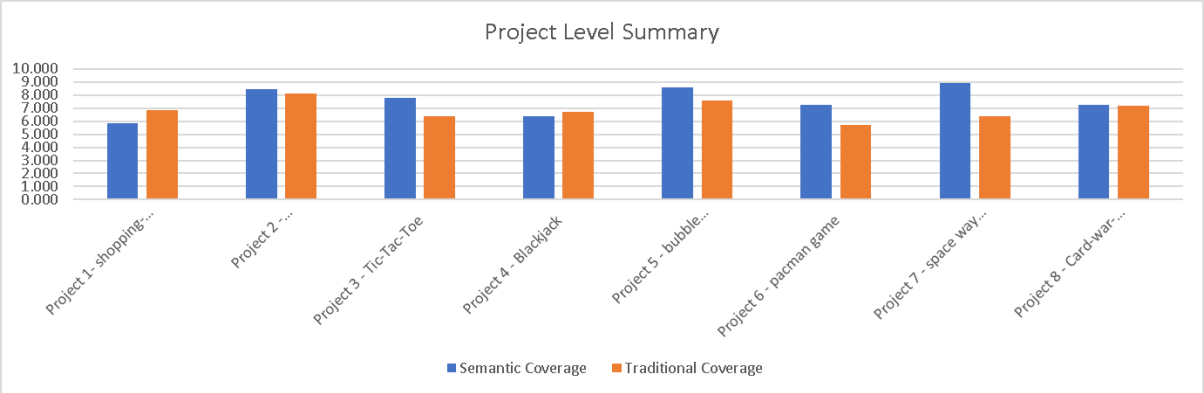
Manual Testing Score							
Project	Function	Code Metrics	Smell Tests	Static analysis	Test Execution Time	Function MTS (avg of all)	Project MTS
Project 1- shopping-cart-unit-tests	calculate_total	8.60	4.00	8.00	8.00	7.15	6.60
	calculate_item_total	8.50	3.60	4.44	8.00	6.14	
	add_item	8.30	4.00	4.62	8.00	6.23	
	calculate_subtotal	8.30	4.30	5.79	8.00	6.60	
	calculate_order_total	7.40	5.00	7.56	8.00	6.99	
	get_reward_points	7.80	4.80	7.24	8.00	6.96	
	get_latest_price	8.30	3.90	5.00	8.00	6.30	
	calculate_item_total	7.80	4.20	5.65	8.00	6.41	
Project 2 - poketerminal-game	calcular_ataque	8.70	3.60	5.88	8.00	6.55	5.62
	definir_vantagens	9.00	3.90	0.00	7.00	4.98	
	_restaurar_pokemons	9.00	3.70	0.00	8.00	5.18	
	batalhar	9.00	4.20	4.44	8.00	6.41	
	capturar_pokemon	9.00	3.30	0.00	8.00	5.08	
	get_inimigo	8.80	4.20	4.76	7.00	6.19	
	_resultado_da_batalha	9.00	3.80	1.67	8.00	5.62	
	apresentar_pokemons	9.00	3.60	0.00	8.00	5.15	
	atacar	6.80	4.60	6.52	5.00	5.73	
	mudar_estrategia_para_luta	8.90	3.50	5.50	7.00	6.23	
	recuperar_vida	8.90	3.40	4.29	7.00	5.90	
	subir_nivel	7.00	4.60	0.00	7.00	4.65	
	_reconfigurar_status	8.80	3.60	6.36	7.00	6.44	
	apresentar_pokemon	9.00	3.10	0.00	8.00	5.03	
	get_pokemon	8.80	3.90	0.00	8.00	5.18	
Project 3 - Tic-Tac-Toe	diagonals	8.90	4.90	0.00	8.00	5.45	6.22
	columns	8.90	4.00	0.00	8.00	5.23	
	update_board	8.80	5.40	2.73	8.00	6.23	
	check_win	8.80	5.30	4.67	8.00	6.69	

- **Identified Blocks of code for Semantic coverage:**
  - Core Functionality: Testing key business logic and expected outputs.
  - Boundary Conditions and Edge Cases: Ensuring proper handling of boundary values and invalid inputs.
  - Error Handling: Validating exception management and system behavior under unexpected conditions.
  - Integration Points: Assessing the reliability of interactions with external systems, such as APIs and databases.
  - User Interface (UI) Interactions: Testing user inputs, navigation, and overall user experience.
  - Security Features: Verifying secure handling of data and prevention of vulnerabilities.

  - Performance and Scalability: Evaluating system behavior under varying loads and stress conditions.
  - Configuration and Environment: Ensuring the system operates correctly across different environments.
  - Output Consistency: Validating the accuracy and consistency of system outputs, such as logs and reports.
- **Created prompts**
  - To split Source code into function blocks
  - To split function blocks into code blocks
- **Defined method to Evaluate Semantic coverage on function level:**
  - Total Projects Evaluated: 5
    * Project 1- shopping-cart-unit-tests
    * Project 2 - poketerminal-game
    * Project 3 - Tic-Tac-Toe
    * Project 4 - Blackjack
    * Project 8 - Card-war-game-python
  - Total functions evaluated: 46
  - Code Block files created: 181
- **Shared Report:**
  - Added Projects to Data Analysis Section
  - Created Workflow Diagrams
  - Methodology Section
  - Results Section







Attempts

- **Scenario coverage:**
  - For projects with no test cases. Used Chat GPT to generate a suite of test cases for different scenarios for Project 1
  - Scenarios introduced:
    - Basic Order Calculation
    - Order with Discounts
    - Order with Shipping Charges
    - Order with Tax Calculation
    - Order with Dynamically Priced Items
    - Negative Scenarios

    - Order Rewards
  - Coverage report created – 95% coverage

```
Name                Stmts   Miss  CoveronProject\Projects\0_Test>
---------------------------------------
Simple_Program.py      60      5    92%
test.py                46      0   100%
---------------------------------------
TOTAL                 106      5    95%
```

- **Manual Code metrics:**
  - **Smell tests:**
    - Created script to evaluate the following smells and tested on Project 1:
      - Large Class (LC)
      - Long Parameter List (LPL)
      - Long Method (LM)
      - Long Message Chain (LMC)
      - Long Scope Chaining (LSC)
      - Long Base Class List (LBCL)
      - Useless Exception Handling (UEH)
      - Long Lambda Function (LLF)
      - Complex List Comprehension (CLC)
      - Long Element Chain (LEC)
      - Long Ternary Conditional Expression (LTCE)
      - Assertion Roulette
      - Lazy Test
      - Magic Number
      - Redundant Print
    - Output: Generates a list of potential fail points and assigns a score to it.
  - **Result:**
    - Previous: Coverage 7.8/10
    - Smell Test: 5.6
    - **Code metrics:**
      - Created a script to evaluate code metrics:

```
Code smells found:
LPL: Score 4/10, 6 occurrences
LM: Score 6/10, 12 occurrences
LSC: Score 10/10, 4074 occurrences
LBCL: Score 10/10, 2501 occurrences
CLC: Score 6/10, 14 occurrences
LEC: Score 10/10, 4074 occurrences
Magic Number: Score 10/10, 292 occurrences

Final Aggregated Score: 5.6/10

Process finished with exit code 0
```

- ALU - Assertion Library Usage: Shows if the class uses Assertion Library. Libraries considered: Hamcrest, Assertj, Atrium, Truth, Valid4j, Datasource-Assert.
- NOAIT - Number of Assertions in Test Class
- NOASIT - Number of Assumptions in Test Class (JUnit 5.0 feature)
- NOAU - Number of @After Annotations in Test Class
- NOBU - Number of @Before Annotations in Test Class
- NOCT - Number of Test Cases in Test Class
- Etc.
- Result: 2.5/10

```
C:\Users\Rouvin\AppData\Local\Microsoft\WindowsApps\python3.9.exe C:/Users/Rouvin/PycharmProjects/pythonProject/Projects/11_Brock_COSC_3P99/Metric_Testing/Code_metrics/Code_metrics_script.py
Code Metrics Score: 2.5/10
Extracted Metrics:
ALU: False
NOAIT: 20
NOASIT: 0
NOAU: 0
NOBU: 0
NOCT: 12
NODT: 0
NODV: 21
NOECU: False
NOEET: 0
NOET: 0
NOFS: 0
NOIS: 0
NONO: 0
NOPT: 0
NORT: 0
NOST: 0

Process finished with exit code 0
```

– **Chat GPT Rating:** 8/10

– **Test Execution Time:**

* Script created:
  - Objective: To measure the execution time of tests in a specified file using pytest.

· Calculates the time taken using ‘subprocess.run()‘ and captures both standard output and error messages.

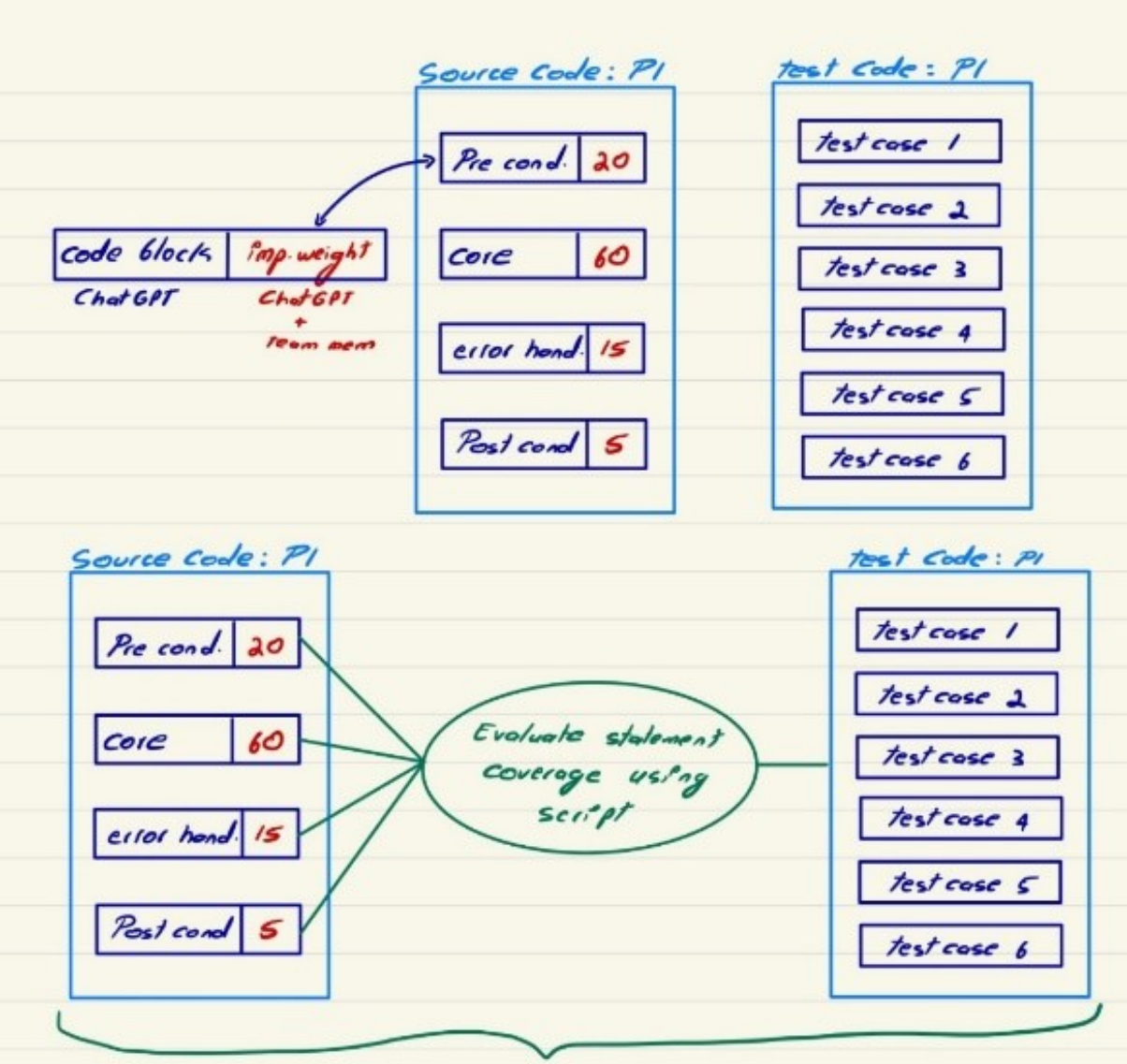
```
C:\Users\Rouvin\AppData\Local\Microsoft\WindowsApps\python3.9.exe C:/Users/Rouvin/PycharmProjects/pythonProject/Projects/1
Test Execution Time: 0.70 seconds
Lines of Code: 111
Performance Score: 6/10

Test Output:

============================= test session starts ==============================
platform win32 -- Python 3.9.13, pytest-8.1.1, pluggy-1.4.0
rootdir: C:\Users\Rouvin\PycharmProjects\pythonProject\Projects\11_Brock_COSC_3P99\Metric_Testing\Test_execution_time
plugins: anyio-4.4.0
collected 29 items

Shopping_cart_project_tests.py .....................EEEEEE            [100%]
```

- **Project Level Semantic Coverage:**
  - The source code of a project is given to a Language Model (LLM) for analysis.
  - Prompting the LLM to:
    - Identify the primary objective of the code.
    - Understand and categorize the core functions based on this objective.
    - Assign a weight to each category reflecting its importance relative to the overall objective.
    - Categorize the lines or blocks of code according to these identified categories.
  - A script is created to:
    - Read and analyze the test files associated with the source code.
    - Determine if the categories with the highest weights (i.e., the most crucial parts of the code) are adequately tested.
    - Generate a score ranging from 1 to 10, where 1 indicates that the highest weighted categories are poorly tested and 10 indicates that they are tested sufficiently.
  - Project tested: `https://github.com/AutomationPanda/shopping-cart-unit-tests.git`
  - Results:
    - Statement Coverage: 57.14% (Score: 5.71)
    - Branch Coverage: 61.21% (Score: 6.12)
    - Function Coverage: 73.33% (Score: 7.33)
    - Traditional Coverage Score: 6.38
  - **Process:**



Projects	Labels	weights	Statement cov.	Semantic coverage
P1	Pre Cond	20	7.81	7.81 * 0.2 + 5.67 * 0.6 +
	core	60	5.67	6.11 * 0.15 + 3.52 * 0.05
	error hand	15	6.11	= 6.05
	Post cond	5	3.52	

* Each source code can be divided into code blocks based on the functionality of the program.
* These blocks are then weighted.
* The test file is measured for statement coverage for each code block, resulting in multiple statement coverage scores.
* The semantic coverage score is then calculated as per the weight of each

code block (weighted average).

* Defined different code blocks that the source code should be split into:
  - Core Functionality
  - Boundary Conditions and Edge Cases
  - Error Handling
  - Integration Points
  - User Interface (UI) Interactions
  - Security Features
  - Performance and Scalability
  - Configuration and Environment
  - Output Consistency
* Updates the test file to account for the modularity of the code blocks (the code blocks need to be tested independently against the test file to generate a statement coverage result).
* Tested one project: `https://github.com/AutomationPanda/shopping-cart-unit-t git`
* Semantic Coverage Score: 5.32
* Traditional Coverage Score: 6.39 (statement, branch, function)
* Manual Testing Score: 4.47 (test execution time, smell test, static analysis, code metrics)