Report on Assignment

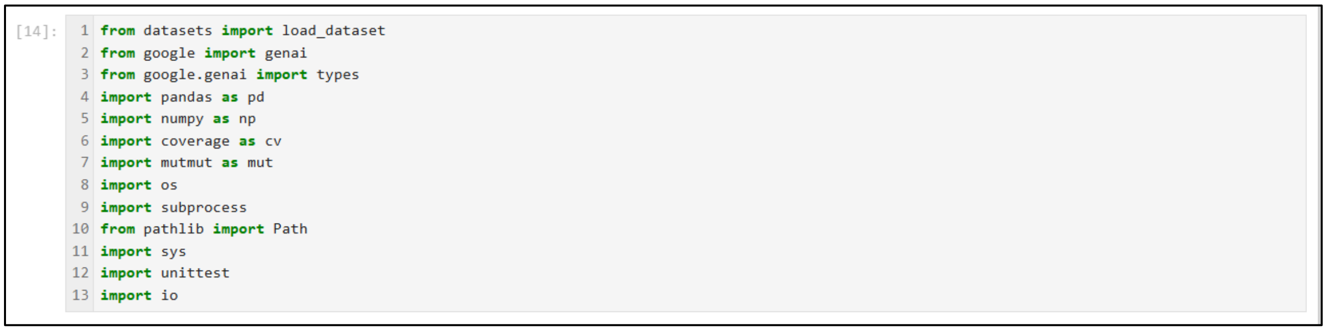
EECS 6466: Software Defect Detection, Tolerance and Repair

Submitted by: Md. Ehsan Shahmi, S#222297790

Github Repo of the Assignment

❖ Environment set-up:-

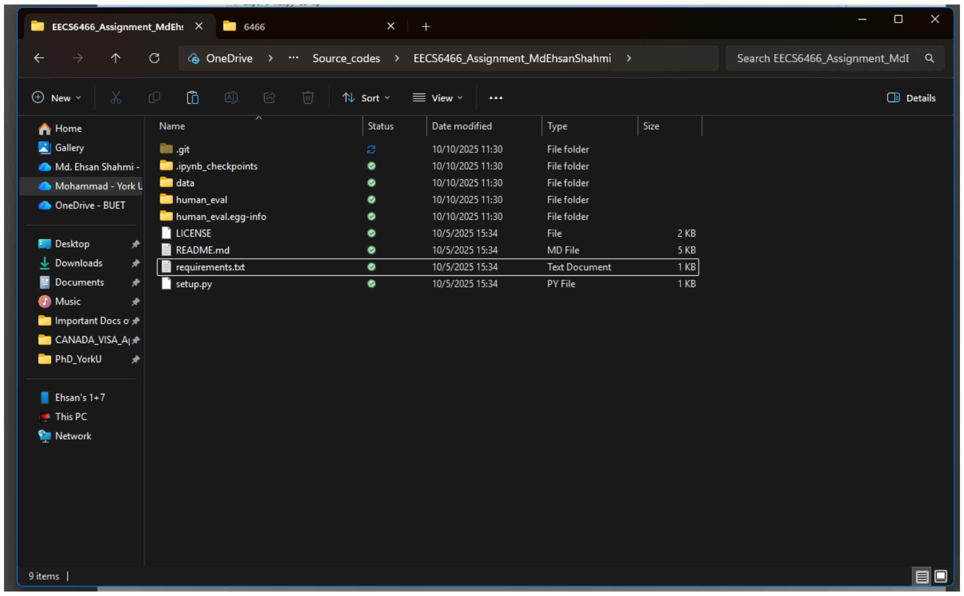
For the assignment, we have used Jupyter Notebook and Anaconda. We needed several dependencies and libraries to work with. The set of libraries used are given in Fig. 1. Each of them is installed separately, had they not come with the installation of Jupyter.



```
[14]:
from datasets import load_dataset
from google import genai
from google.genai import types
import pandas as pd
import numpy as np
import coverage as cv
import mutmut as mut
import os
import subprocess
from pathlib import Path
import sys
import unittest
import io
```

Fig. 1: The dependencies and libraries utilized.

Firstly, we downloaded the **Human-Eval dataset**. We have used "git clone" into our repository directly to get and see the folders associated with it. Our repository looks like Fig.2. In our workspace, we have loaded the dataset directly using libraries. We loaded it into two types of objects: a **Dataset dictionary** object and a **pandas dataframe** object to work upon the dataset.



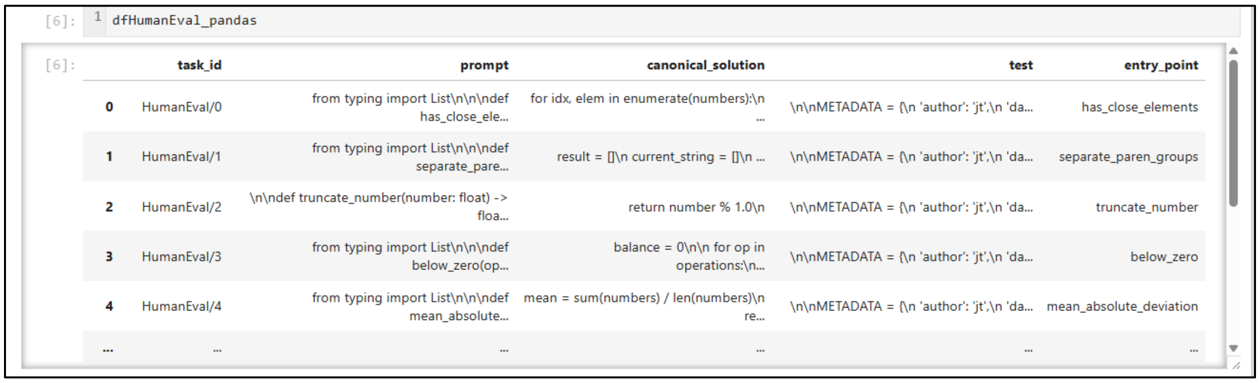
OneDrive > ··· Source_codes > EECS6466_Assignment_MdEhsanShahmi >

Name	Status	Date modified	Type	Size
.git		10/10/2025 11:30	File folder	
.ipynb_checkpoints		10/10/2025 11:30	File folder	
data		10/10/2025 11:30	File folder	
human_eval		10/10/2025 11:30	File folder	
human_eval.egg-info		10/10/2025 11:30	File folder	
LICENSE		10/5/2025 15:34	File	2 KB
README.md		10/5/2025 15:34	MD File	5 KB
requirements.txt		10/5/2025 15:34	Text Document	1 KB
setup.py		10/5/2025 15:34	PY File	1 KB

9 items

Fig. 2: A screenshot of the repository containing the Human-Eval dataset.

It is really easy to work with the dataset as a pandas dataframe object. After loading into the workspace, we also store the dataset in a .csv file in our repo. Loading the dataset in the workspace shows that the dataset has 5 columns: **task_id, prompt, canonical_solution, test and entry_point**.



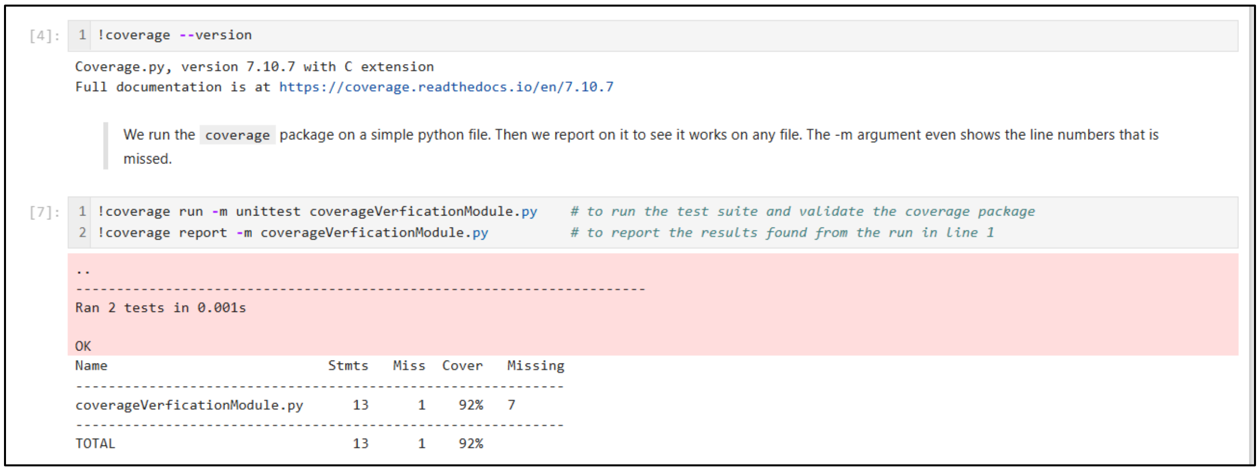
```
[6]: dfHumanEval_pandas
```

[6]:

	task_id	prompt	canonical_solution	test	entry_point
0	HumanEval/0	from typing import List\n\n\ndef has_close_ele...	for idx, elem in enumerate(numbers):\n ...	\n\nMETADATA = {\n 'author': 'jt',\n 'da...	has_close_elements
1	HumanEval/1	from typing import List\n\n\ndef separate_pare...	result = []\n current_string = []\n ...	\n\nMETADATA = {\n 'author': 'jt',\n 'da...	separate_paren_groups
2	HumanEval/2	\n\ndef truncate_number(number: float) -> floa...	return number % 1.0\n	\n\nMETADATA = {\n 'author': 'jt',\n 'da...	truncate_number
3	HumanEval/3	from typing import List\n\n\ndef below_zero(op...	balance = 0\n\n for op in operations:\n...	\n\nMETADATA = {\n 'author': 'jt',\n 'da...	below_zero
4	HumanEval/4	from typing import List\n\n\ndef mean_absolute...	mean = sum(numbers) / len(numbers)\n re...	\n\nMETADATA = {\n 'author': 'jt',\n 'da...	mean_absolute_deviation
...	...	...	...	...	...

Fig. 3: Screenshot of the dataset loaded in the workspace, as a dataframe

After dataset, we install the packages of **coverage** and **mutmut** using the simple command line statement: ***pip install coverage*** and ***pip install mutmut***. After installation, we have tested the working of the coverage library, using displaying its version as well as creating a simple python file ("*coverageVerificationModule.py*") and execute on it to the coverage percentage of the test cases inside the file. These are provided in Fig 4 and Fig 5.



```
[4]: !coverage --version

Coverage.py, version 7.10.7 with C extension
Full documentation is at https://coverage.readthedocs.io/en/7.10.7
```

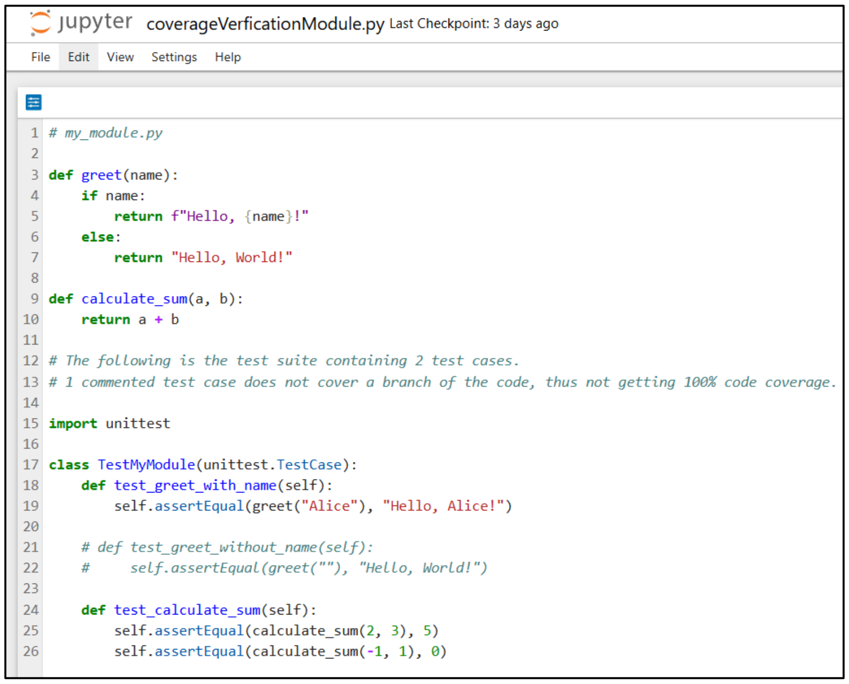
We run the `coverage` package on a simple python file. Then we report on it to see it works on any file. The -m argument even shows the line numbers that is missed.

```
[7]: !coverage run -m unittest coverageVerficationModule.py    # to run the test suite and validate the coverage package
     !coverage report -m coverageVerficationModule.py          # to report the results found from the run in line 1

..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
Name                           Stmts   Miss  Cover   Missing
------------------------------------------------------------
coverageVerficationModule.py      13      1    92%   7
------------------------------------------------------------
TOTAL                             13      1    92%
```

Fig. 4: The verification of the coverage library

We have installed the latest **coverage** library which is **version 7.10.7**. In our example, we show the process of running coverage using a shell command, that can also be run in the Jupyter notebook.

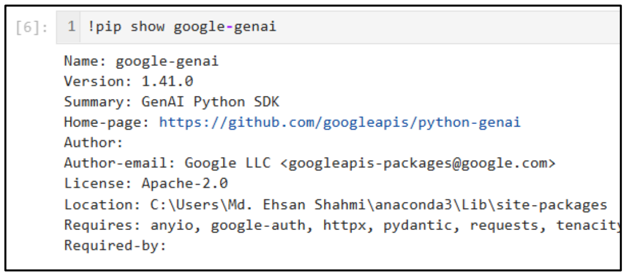
We have also tested the installation of **mutmut** package. Similarly the result of mutmut running was also tested in the workspace as well as its version, which is **version 2.4.4**.


jupyter coverageVerficationModule.py Last Checkpoint: 3 days ago
File
Edit
View
Settings
Help

```python
# my_module.py

def greet(name):
    if name:
        return f"Hello, {name}!"
    else:
        return "Hello, World!"

def calculate_sum(a, b):
    return a + b

# The following is the test suite containing 2 test cases.
# 1 commented test case does not cover a branch of the code, thus not getting 100% code coverage.

import unittest

class TestMyModule(unittest.TestCase):
    def test_greet_with_name(self):
        self.assertEqual(greet("Alice"), "Hello, Alice!")

    # def test_greet_without_name(self):
    #     self.assertEqual(greet(""), "Hello, World!")

    def test_calculate_sum(self):
        self.assertEqual(calculate_sum(2, 3), 5)
        self.assertEqual(calculate_sum(-1, 1), 0)
```

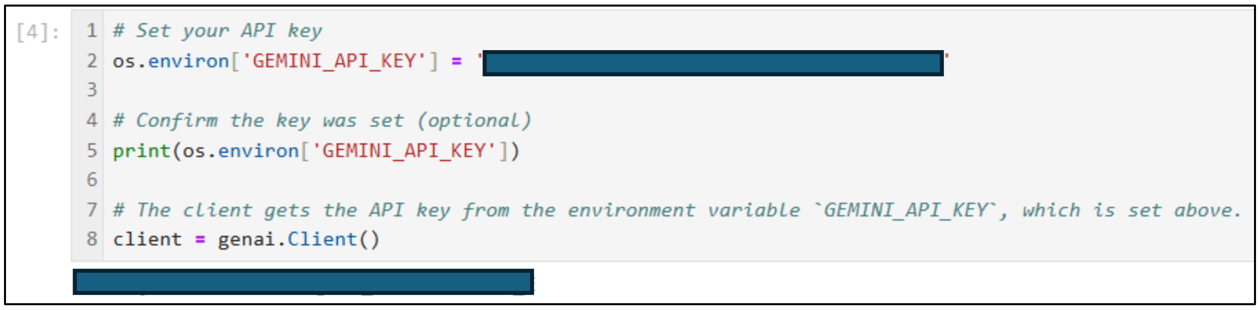
Fig. 5: coverageVerificationModule.py file to test the coverage library

Finally, we have installed and used the **Large Language Model (LLM) gemini-2.5-flash** to carry the main work. Using ***pip install google-genai,*** we downloaded the library to fetch anything to the LLM for our assignment. We have used **version 1.41.0**. In addition, a free API is created by us from the website of Google AI. This API is needed to send the prompts to the LLM directly from our workspace. However, the free API version allows around 250+ requests per 24 hours. We have set-up our API globally, so that we don't need to do this multiple times.



```
[6]:  !pip show google-genai

      Name: google-genai
      Version: 1.41.0
      Summary: GenAI Python SDK
      Home-page: https://github.com/googleapis/python-genai
      Author:
      Author-email: Google LLC <googleapis-packages@google.com>
      License: Apache-2.0
      Location: C:\Users\Md. Ehsan Shahmi\anaconda3\Lib\site-packages
      Requires: anyio, google-auth, httpx, pydantic, requests, tenacity
      Required-by:
```

Fig. 6: The version of the LLM used and tested the installation



```
[4]:  # Set your API key
      os.environ['GEMINI_API_KEY'] = '[redacted]'

      # Confirm the key was set (optional)
      print(os.environ['GEMINI_API_KEY'])

      # The client gets the API key from the environment variable `GEMINI_API_KEY`, which is set above.
      client = genai.Client()

      [redacted]
```

Fig. 7: The generated free API from the website

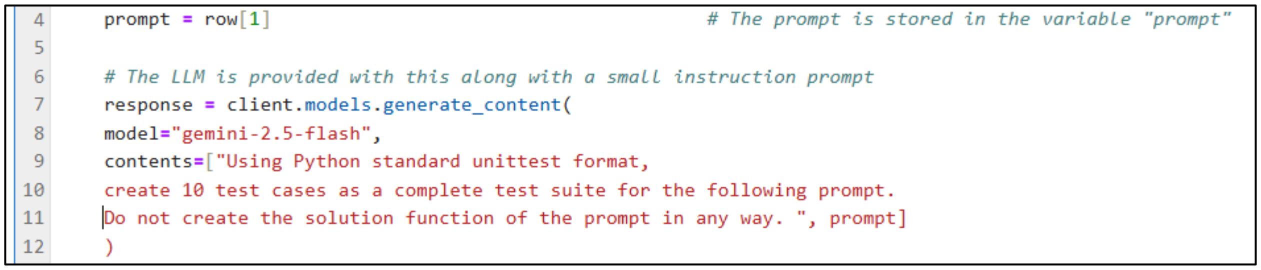
Additionally, we have also the LLMs operation against a simple provided prompt and see the generated result. This is displayed in the workspace directly.

❖ <u>Test Generation (Part 1):-</u>

The task was to instruct the LLM to generate 10 test cases for each problem of the dataset. We were instructed not to provide the ground truth solution of the dataset to the LLM. We created a simple prompt and fed it into the LLM. The provided prompt from us is as follows:

"Using Python standard unittest format, create 10 test cases as a complete test suite for the following prompt. Do not create the solution function of the prompt in any way."

Along with our provided prompt, we also supplied (as a concatenation) the LLM the prompt given in the dataset for each problem. This is displayed in Fig. 8.

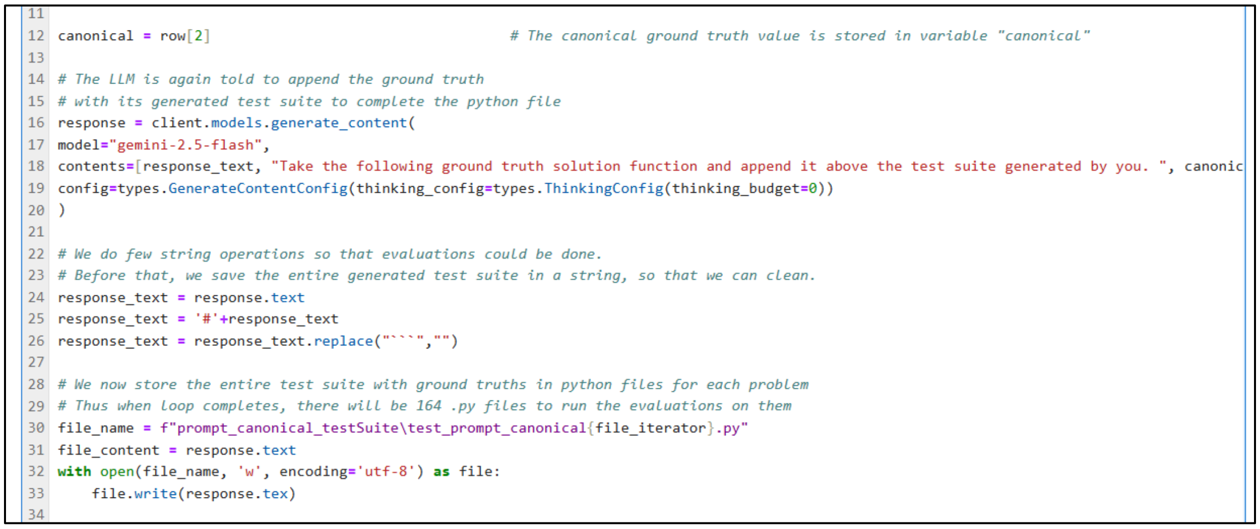


```python
    prompt = row[1]                                         # The prompt is stored in the variable "prompt"

    # The LLM is provided with this along with a small instruction prompt
    response = client.models.generate_content(
    model="gemini-2.5-flash",
    contents=["Using Python standard unittest format,
    create 10 test cases as a complete test suite for the following prompt.
    Do not create the solution function of the prompt in any way. ", prompt]
    )
```

Fig. 8: Providing the LLM the dataset prompt and our own prompt, without giving the ground truth.

We have saved the generated result in a directory for generated test cases as suites for each problem. We run this above for all the problems of the dataset. Hence, we are created with a directory of 164 generated test suites.

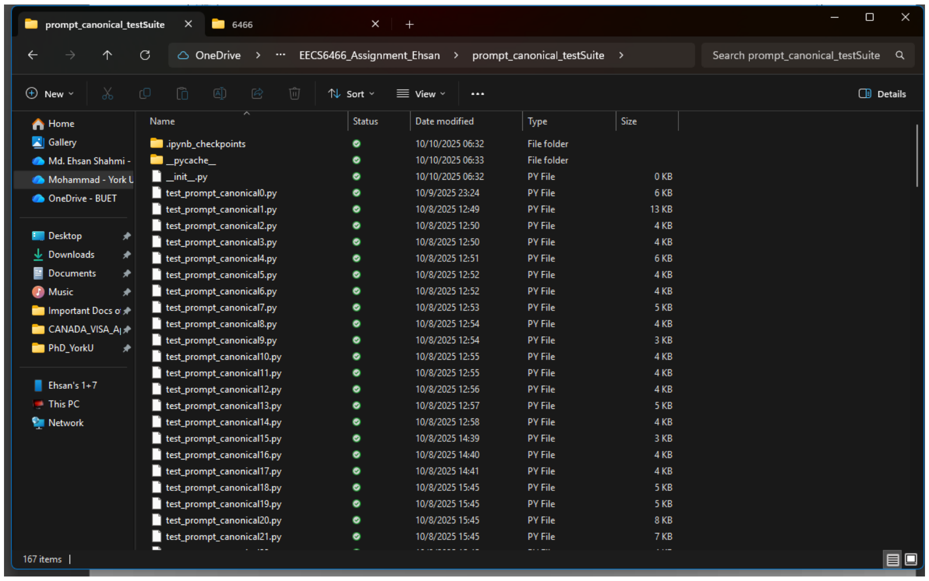
Additionally, for the next instruction for evaluation, we concatenate the ground truth provided solution in the dataset with the generated test suites by the LLM. This ground truth solution can be denoted as the **System under Test (SUT)** for Part 1 of this assignment. We did this concatenation with the help of LLM itself. Then we did some cleaning to make a runnable python file. These files are needed for the evaluation part of the assignment. The cleaned finalized appended files ready for evaluation are stored in the directory ***"prompt_canonical_testSuite\",*** where each file is named as "***test_prompt_canonical*.py***". Here the * denotes the serial of the problem number from the dataset.



```python

canonical = row[2]                                      # The canonical ground truth value is stored in variable "canonical"

# The LLM is again told to append the ground truth
# with its generated test suite to complete the python file
response = client.models.generate_content(
model="gemini-2.5-flash",
contents=[response_text, "Take the following ground truth solution function and append it above the test suite generated by you. ", canonic
config=types.GenerateContentConfig(thinking_config=types.ThinkingConfig(thinking_budget=0))
)

# We do few string operations so that evaluations could be done.
# Before that, we save the entire generated test suite in a string, so that we can clean.
response_text = response.text
response_text = '#'+response_text
response_text = response_text.replace("```","")

# We now store the entire test suite with ground truths in python files for each problem
# Thus when loop completes, there will be 164 .py files to run the evaluations on them
file_name = f"prompt_canonical_testSuite\test_prompt_canonical{file_iterator}.py"
file_content = response.text
with open(file_name, 'w', encoding='utf-8') as file:
    file.write(response.tex)

```

Fig. 9: Appending the ground truth and the test suite generated by LLM and saving

We run the above steps for each problem of the dataset, with the help of utilizing a variable "**file_iterator**". All the steps are shown in the provided workspace.

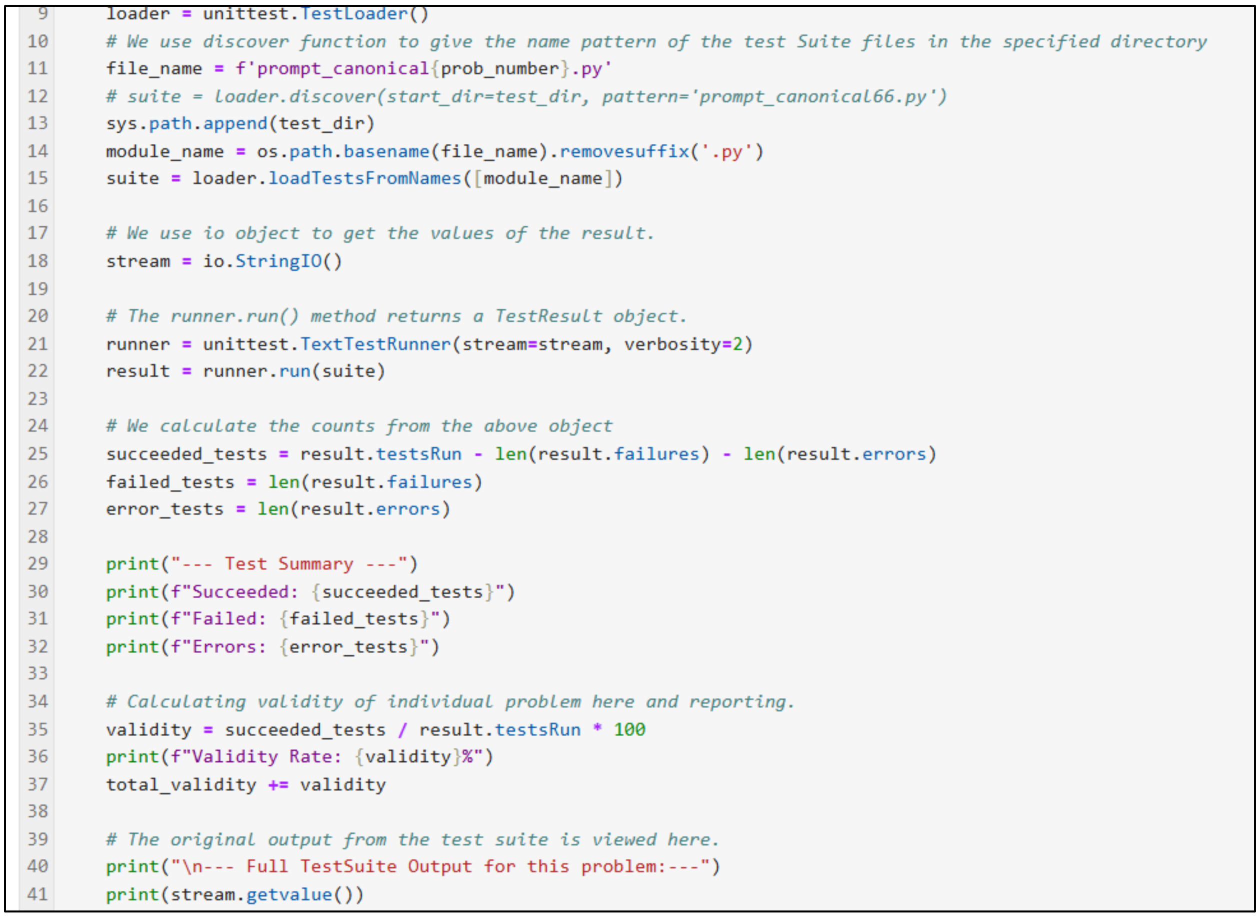


Name	Status	Date modified	Type	Size
.ipynb_checkpoints	✓	10/10/2025 06:32	File folder	
__pycache__	✓	10/10/2025 06:33	File folder	
__init__.py	✓	10/10/2025 06:32	PY File	0 KB
test_prompt_canonical0.py	✓	10/9/2025 23:24	PY File	6 KB
test_prompt_canonical1.py	✓	10/8/2025 12:49	PY File	13 KB
test_prompt_canonical2.py	✓	10/8/2025 12:50	PY File	4 KB
test_prompt_canonical3.py	✓	10/8/2025 12:50	PY File	4 KB
test_prompt_canonical4.py	✓	10/8/2025 12:51	PY File	6 KB
test_prompt_canonical5.py	✓	10/8/2025 12:52	PY File	4 KB
test_prompt_canonical6.py	✓	10/8/2025 12:52	PY File	4 KB
test_prompt_canonical7.py	✓	10/8/2025 12:53	PY File	5 KB
test_prompt_canonical8.py	✓	10/8/2025 12:54	PY File	4 KB
test_prompt_canonical9.py	✓	10/8/2025 12:54	PY File	3 KB
test_prompt_canonical10.py	✓	10/8/2025 12:55	PY File	4 KB
test_prompt_canonical11.py	✓	10/8/2025 12:55	PY File	4 KB
test_prompt_canonical12.py	✓	10/8/2025 12:56	PY File	4 KB
test_prompt_canonical13.py	✓	10/8/2025 12:57	PY File	5 KB
test_prompt_canonical14.py	✓	10/8/2025 12:58	PY File	4 KB
test_prompt_canonical15.py	✓	10/8/2025 14:39	PY File	3 KB
test_prompt_canonical16.py	✓	10/8/2025 14:40	PY File	4 KB
test_prompt_canonical17.py	✓	10/8/2025 14:41	PY File	4 KB
test_prompt_canonical18.py	✓	10/8/2025 15:45	PY File	5 KB
test_prompt_canonical19.py	✓	10/8/2025 15:45	PY File	5 KB
test_prompt_canonical20.py	✓	10/8/2025 15:45	PY File	8 KB
test_prompt_canonical21.py	✓	10/8/2025 15:45	PY File	7 KB

Fig. 10: The directory containing all test suites appended with the ground truth solutions

❖ <u>**Test Evaluation (Part 1):-**</u>

Firstly, we test the validity rate of the test suite. This is done by just running the above 164 python files. For each problem, we loaded the file name and run it using the ***unittest.TestRunner method.*** We note down the number of tests run for each problem, as well as the number of failures, and number of successes. We also note down these numbers and report on the validity of each problem. Finally, we also provide the entire output of the test suite to see in detail which test has passed, and which has failed. These outputs can be displayed in the output portion of the workplace.



```
    loader = unittest.TestLoader()
    # We use discover function to give the name pattern of the test Suite files in the specified directory
    file_name = f'prompt_canonical{prob_number}.py'
    # suite = loader.discover(start_dir=test_dir, pattern='prompt_canonical66.py')
    sys.path.append(test_dir)
    module_name = os.path.basename(file_name).removesuffix('.py')
    suite = loader.loadTestsFromNames([module_name])

    # We use io object to get the values of the result.
    stream = io.StringIO()

    # The runner.run() method returns a TestResult object.
    runner = unittest.TextTestRunner(stream=stream, verbosity=2)
    result = runner.run(suite)

    # We calculate the counts from the above object
    succeeded_tests = result.testsRun - len(result.failures) - len(result.errors)
    failed_tests = len(result.failures)
    error_tests = len(result.errors)

    print("--- Test Summary ---")
    print(f"Succeeded: {succeeded_tests}")
    print(f"Failed: {failed_tests}")
    print(f"Errors: {error_tests}")

    # Calculating validity of individual problem here and reporting.
    validity = succeeded_tests / result.testsRun * 100
    print(f"Validity Rate: {validity}%")
    total_validity += validity

    # The original output from the test suite is viewed here.
    print("\n--- Full TestSuite Output for this problem:---")
    print(stream.getvalue())
```

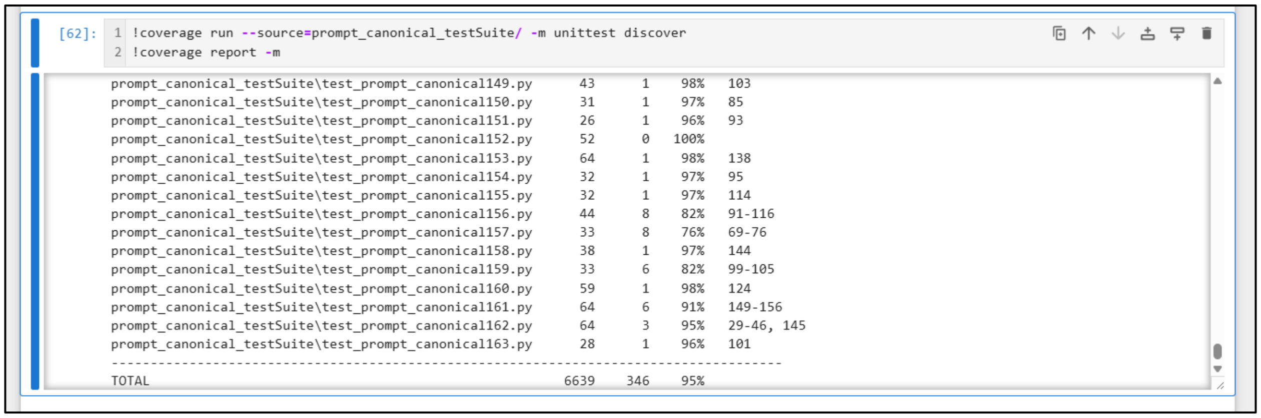
Fig. 11: Evaluation of individual test Suite and measuring the validity of the test suite

We did these steps on all the test suites over the dataset and finally calculate the average validity rate of the generated LLM-test suites as a whole.

Our average validity rate of the entire dataset is 97.01%.

Secondly, we test our test suites for coverage. For this no need of the coding was required, since **coverage library** have a good number of useful commands. We run two such commands. The first (***coverage run***) command is to run the test suites against the ground truth solutions and find the coverage percentages. This command does this operation as a whole in a single directory, so we provided our directory, ***prompt_canonica_testSuite*** on the command as its argument.

Additionally, we generate a report to see the coverage rates of each suite on each file. The second command (***coverage report***) directly notifies of this. A detailed report is generated if the argument ***-m*** is given with the command. Here the report shows all the test suites' coverages in percentages in the third column, whereas the fourth column displays the missing lines if the test cases for that particular problem. At the bottom of the report, we observe the final average coverage percentage of the entire dataset.



```
[62]: !coverage run --source=prompt_canonical_testSuite/ -m unittest discover
      !coverage report -m
```

```
prompt_canonical_testSuite\test_prompt_canonical149.py      43      1    98%   103
prompt_canonical_testSuite\test_prompt_canonical150.py      31      1    97%   85
prompt_canonical_testSuite\test_prompt_canonical151.py      26      1    96%   93
prompt_canonical_testSuite\test_prompt_canonical152.py      52      0   100%
prompt_canonical_testSuite\test_prompt_canonical153.py      64      1    98%   138
prompt_canonical_testSuite\test_prompt_canonical154.py      32      1    97%   95
prompt_canonical_testSuite\test_prompt_canonical155.py      32      1    97%   114
prompt_canonical_testSuite\test_prompt_canonical156.py      44      8    82%   91-116
prompt_canonical_testSuite\test_prompt_canonical157.py      33      8    76%   69-76
prompt_canonical_testSuite\test_prompt_canonical158.py      38      1    97%   144
prompt_canonical_testSuite\test_prompt_canonical159.py      33      6    82%   99-105
prompt_canonical_testSuite\test_prompt_canonical160.py      59      1    98%   124
prompt_canonical_testSuite\test_prompt_canonical161.py      64      6    91%   149-156
prompt_canonical_testSuite\test_prompt_canonical162.py      64      3    95%   29-46, 145
prompt_canonical_testSuite\test_prompt_canonical163.py      28      1    96%   101
--------------------------------------------------------------------------------------
TOTAL                                                     6639    346    95%
```

Fig. 12: The coverage percentages and the average coverage of the dataset

Our average coverage of the entire dataset is 95.0%.

Finally, we evaluate the dataset and the LLM-generated Test suites against mutation score. Similar to the coverage library, mutation library also has some command line instructions to operate mutations on the source ground truth values of the problems. The command (***mutmut run –paths-to-mutate==`directory`***) takes the source ground truth solution functions and mutate them. Then the command introduces mutates and then runs the existing test suite on this mutated code. Mutation score is calculated on the percentage of the ratio of killed mutants to the total number of mutants. Then we get the average score by using all the mutation scores to find the average mutation score for our generated test suites of the entire dataset.

Our average mutation score of the entire dataset is 88.23%.

❖ Discussion and analysis (Part 1):-

LLMs are tasked to generate test cases for a provided prompt. In this assignment, using a prompt we used gemini-2.5-flash LLM to create 10 test cases, creating a single test suite, for a given problem. This is tasked over 164 problems of the entire Human-eval dataset. The generated test suites are evaluated using some **adequacy metrics**. The three-adequacy metrics to measure the quality of the generated test suites are the validity rate, coverage and mutation score.

We see from the above reports of this assignment, the validity rate is the largest value while the mutation score is the lowest. So according to our assignment,

average validity score > average coverage > average mutation score

Validity score is the measurement of the ratio between passed tests to the total number of tests. From that simple relationship, the validity should be the highest as usual. This is because the 10 test cases generated by the LLM are too correct.

On the other hand, coverage measures the line coverage of the test cases provided on a given ground truth function. This means that any function containing if-else case has many branches and lines in it, all of which are being tested with those generated test cases. This is separate from the simple validity metric. As a result, many test cases fail to address all the line or branch coverages, where those tests pass simply on the run in validation measurement. That's why coverage gives a lower result than validity score.

Similarly, mutation scores are stricter than the other two. This is because mutants are generated using the library package by operating on the mutant operators of the entire function. The test cases are then tried against these mutants of the function. Furthermore, test cases fail on this newly created mutated function than in case of coverage success. As a result, mutation score becomes the lowest metric value compared to coverage and validity rate.

To conclude, **the reason behind this is that the Google gemini-2.5-flash LLM is trained on the Human-Eval dataset itself**. For this reason, when we were tasked to generate test cases, LLM created test cases which have almost no way to get failed. This is proven by the three metrics mentioned above. However, because of the entire design mechanism of the metrics, mutation score is the best metric to evaluate correctly test suites. That's why the mutation score is the lowest on the dataset.

[Please note that more comments are made in the Jupyter notebook file connected with this report, submitted in the same repository.]

❖ <u>Iterative Enhancement Process (Part 2):-</u>

In this part of the assignment, we separate the test suite for each problem from the ground truth solutions using simple string operations. Afterwards, we provide the LLM again, with this test suite and the given prompt from the Human-eval dataset. We prompt the LLM to generate a possible solution of the function that is specified in the dataset-provided prompt. We use the following prompt to task this to the LLM:

"Using the provided 10 test cases and the function specification, create a possible solution code function for the problem in a runnable python format. The entire test suite must contain the 10 test cases below your possible solution function."

We store the new files separately. As before in part 1, we run the old test cases against this new **System Under Test (SUT).** The tests after being run, some of them were failing more than the Part 1. For each failed test, for each problem, we try to capture the error message of its failure. Then this error message along with the entire new SUT and the 10-test case test suite is provided to the LLM again to get the failed test to be fixed. For this step we used 3 iterations. This is because, much more iterations exhaust our API to prompt the LLM for the day, as our limit was around 270~300 requests. In these 3 iterations, we captured error message of the failed test case. Then, we send the whole SUT and test suite along with this error message to LLM and prompt to fix it. Then we capture what the LLM returned. Then we run the new suite again on the SUT. We see if the failed test fails again. If yes, then we iterate 2 more times. This entire step is done for all the problems of the dataset.

Finally, we try to measure the newly fixed test cases' new coverage value. At the start of the previous we store the coverage percentage. We try to compare between the two coverages of that problem. Unfortunately, we were not able to complete these steps of the assignment.

--- the end ---