Milestone 2 - ML4SE - Group-B

Group Members


```
Ashwin Patil <anpatil2@illinois.edu>
Chinmay Saraf <csaraf2@illinois.edu>
Kedar Takwane <takwane2@illinois.edu>
Maahi Patel <maahidp2@illinois.edu>
```


Repository Link

- Main Repository: https://github.com/theashwin/ml4se
- Milestone-2 Code and Data: https://github.com/theashwin/ml4se/tree/main/milestone-2
- Main Dataset Used: https://github.com/github/CodeSearchNet

Introduction

In this project, we're evaluating ChatGPT's effectiveness to reason about a code snippet's execution, generate tests and finally generate a semantically equivalent function for a given function. We use Python and Java repositories from the CodeSearchNet corpus. We use OpenAI API to interact with ChatGPT and then we engineer prompts to elicit the required outputs from the model. We finally evaluate the generated reasoning, tests, and methods by setting up and running them. We present our observation and results overview in this report.

Methodology

1. **Data Collection** - We selected 6 repositories from CodeSearchNet : 3 each for Python and Java (repositories listed further down). From these 6 repositories, we selected 50 functions for Python and Java each. Limiting the source repositories to 6 helps in testing the generated test cases as the project setup becomes relatively easier.
2. **Data Categorization** - To help us understand the accuracy and effectiveness of ChatGPT to solve the 3 tasks we categorized the functions into three categories based on the type of inputs, outputs, and the body of the function by whether it's primitive or non-primitive. *We concede there might be better ways to categorize the dataset but these are quite good approximations.*
    a. Easy - If the used variables are of primitive datatypes.
    b. Medium - If the used variables contain both primitive and non-primitive datatypes.
    c. Hard - If the used variables are of non-primitive datatypes.
3. **Prompt Engineering** - Using **random** data points from the dataset we used multiple prompts to elicit the required response from the model. **The tried prompts along with the screenshots can be found in the folder** `milestone-2/observations` **For a better understanding the prompts are further categorized by tasks.**
4. **Automating Runs -** Once the dynamic prompts were finalized we ran these prompts for 50 functions each for Python and Java. Outputs can be found at `milestone-2/out`
    a. **Dynamic** - We've simulated back-and-forth conversations with ChatGPT by breaking down the task into parts which are known to help the model understand the task better and elicit better responses.
    b. **Markdown** - As the output given by ChatGPT is in markdown format (which was quite an interesting observation), we save the response from the model in `.md` format which is visually pleasing and a better way to present our work.
5. **Evaluated Method Execution Reasoning -** After the generation of the outputs we verify the evaluate the generated outputs and add our observations in the individual function files.

6. **Tested Generated Tests -** For evaluating this task we had to set up individual repositories for running the generated tests. Once set up we follow the following methodology.
    a. If the generated test is generated as a standalone test file we create a new file and try to run it for evaluation.
    b. If the generated test is a single function we try to find an adequate existing test file to append this function too and then evaluate.
    c. We do the following practical changes to ensure that the generated tests can be evaluated.
        i. Change the type of tabs to make it compatible with existing code (As Python is tab sensitive)
        ii. Other formatting changes without changing the semantics and syntax of the function to ensure a fair evaluation.
7. **Tested Generated Semantically Equivalent Methods -** Similar to the above task, we evaluate the semantically generated by appending the generated function in the existing code file by renaming the function name to make it work.
    a. Testing
        i. As the focus here is on the function generated and not on the tests generated. We use existing tests for this function if it exists.
        ii. Else we use the test generated in the previous task but in this case, we modify the to make it runnable and semantically correct.
8. **Report Generation -** With everything else out of the way we generate this report, to sum up, our observations and give statistics on what works and what does not and for what cases the model struggle, and in what part it excels.

Important Links

1. Data - https://github.com/theashwin/ml4se/tree/main/milestone-2/data
2. Multi-Turn Prompts - https://github.com/theashwin/ml4se/tree/main/milestone-2/prompt/json
    a. Init = Initial Prompt
    b. 1…N = N-Turn Prompts
3. Observations (Used for Prompt Generation) - https://github.com/theashwin/ml4se/tree/main/milestone-2/observations
4. Final Output - https://github.com/theashwin/ml4se/tree/main/milestone-2/out
5. Automation Script Code - The main script which is executing all prompts is gpt.py. This script is calling:
    a. reasoning.py - This script executes the reasoning prompt for a code
    b. tests.py - This script executes the test generation prompt for a code
    c. refactor.py - This script executes the refactoring prompt for a code

Design Decisions

Markdown Output

We realized that the output that OpenAI API returns are in the markdown format so instead of storing the output in a textual format our output is in markdown format. **This part is also autogenerated so needs no manual effort.** This has many benefits like,

1. The output is displayed exactly like it was intended by ChatGPT
2. We can manipulate the text file to highlight the important parts or add additional information to the file to make the report per function.
3. It is much easier to read and understand. For instance, tables are actually shown in a tabular format instead of text with dashes.

Tasks are Independent

To avoid wasting up the context space which is already limited in transformer-based models. We decided to treat each one of the tasks as **independent.** That means that after each task for a particular function, the context is cleaned up. This allows the tasks to be run independently but on the downside, the context (for instance, the method under test, etc.) needs to be provided again to the model for each task.

Corner Case - No Return Statement

There were a few cases where there are no return statements at all. In this case, we ask for the behavior of the function itself and move away from the input-output type of checking which we do in other cases.

Corner Case - No Input Params

We've also handled cases where there are no input parameters in the method under test. In this case, we've added another field in the dataset to include variables of interest that will be passed to the model alongside the code itself. The intuition behind this is to give the model as much information as possible and the variables we've chosen are global variables that potentially can take up any value.

Installation

Requirements

The one and only requirement of this project is a pip package named `openai` which provides convenient access to the OpenAI API from applications written in the Python language. It includes a pre-defined set of classes for API resources that initialize themselves dynamically from API responses which makes it compatible with a wide range of versions of the OpenAI API.

```
pip install --upgrade openai
```

This would require an OpenAI API key which can be obtained from `https://platform.openai.com/account/api-keys`. Save the key in a new file named `config.json` in `milestone-2` folder in the format given below.

```
{
    "OPENAI_API_KEY": "<KEY>"
}
```

Execute

After you've added `config.json` with the required OpenAI API. To run the code you need to execute `gpt.py` with optional parameters `--lang=python/java` by default the language is considered as `Java` and `--n=50` or `--number=50` by default the number of functions is `5` (As it takes quite a long time, due to API limits). **Running this file will run all three tasks for n code snippets for that particular language Generated outputs can be found in `milestone-2/out/<lang>/#.md` Where # is the serial number of the function.**

```
gpt.py --lang=python --n=5 #For Python

gpt.py --lang=java --n=5 #For Java
```

Data

The dataset can be found in `milestone-2/prompt/json/java.json` or `milestone-2/prompt/json/python.json` Content of the JSON is as follows:

Java

	Java	Python
Easy	18	32
Medium	15	12

Hard	17	7
Total	**50**	**51**

Java Example

```
{
    "func_id": 1,
    "project_repo": "https://github.com/Esri/geometry-api-java",
    "location": "src/main/java/com/esri/core/geometry/Bufferer.java",
    "function_name": "progress_",
    "function": "private void progress_() {
                \n\t\tm_progress_counter++;\n\t\tif
                (m_progress_counter % 1024 == 0) {\n\t\t\tif ((m_progress_tracker
                != null)\n\t\t\t\t\t\t&& !(m_progress_tracker.progress(-1, -1)))
                \n\t\t\t\tthrow new RuntimeException(\"user_canceled\");\n\t\t}\n\t}",
    "non_primitive_data_objects": "",
    "primitive_data_objects": "",
    "input": "m_progress_counter",
    "expected_output": "",
    "flag": "hard"
  }
```

Python Example

```
{
  "func_id": 1,
  "project_repo": "https://github.com/limix/numpy-sugar/blob/4bdfa26913135c76ef3cd542a332f4e5861e948b/numpy_sugar/linalg",
  "location": "numpy-sugar/numpy_sugar/linalg/_kron.py",
  "function_name": "kron_dot",
  "function": "def kron_dot(A, B, C, out=None):\n    \n
              from numpy import dot, zeros, asarray\n\n    A = asarray(A)\n
              B = asarray(B)\n    C = asarray(C)\n\n    if out is None:\n
              out = zeros((B.shape[0], A.shape[0]))\n
              dot(B, dot(C, A.T), out=out)\n    return out",
  "non_primitive_data_objects": "",
  "primitive_data_objects": "",
  "input": "",
  "expected_output": "",
  "flag": "hard"
}
```

Projects Setup

We set up a total of 6 projects: 3 Python and 3 Java projects to test the execution of generated tests and semantically equivalent methods. The projects are listed below:

Java

1. `https://github.com/Esri/geometry-api-java`
2. `https://github.com/locationtech/jts`
3. `https://github.com/looly/hutool`

Python

1. `https://github.com/limix/numpy-sugar/`
2. `https://github.com/scipy/scipy`
3. `https://github.com/numpy/`

Observations

Prompt Engineering

Listed below are the part of prompts that we've included/excluded from our final prompt and our reasoning/intuition behind why they might or might not work and why there are important to the overall prompt. There are four sections below: three

task-specific sections and one general section which relates to the formatting of the responses and prompts.

Legend

🛠️ Final/Best Prompt Used

✅ Successful Prompt

⚠️ Moderately Successful Prompt

⛔ Unsuccessful Prompt

Task 1: Code Explanation

🛠️ **Prompt 1: For the <java/python> function with comments given in the variable 'code' below, write three inputs and outputs in a table and reason about the execution of the function. Follow the instructions carefully. Instructions: Assume the variable's contents as required, The generated response must have the following qualities: concise, stepwise and specific.**

code: <code>

Response 1: <response>

Prompt 2: Show calculations and reasons for each of the three input-output pairs.

1️⃣ **“For the java function with comments given in the variable 'code' below…”**

This part of the prompt defines the task at hand along with the programming language used. For our use case, the languages will be java/python. The reason behind adding “with comments” is that we think this helps the model consider the comments to write the explanation for any given input.

2️⃣ “**write three inputs and outputs in a table and reason about the execution of the function.”**

Instead of creating the input ourselves and asking the model for answers and then evaluating it’s reasoning, we get the input from GPT this allows us to check for edge cases where the function might break. This also is a good way for us to evaluate GPT’s effectiveness and understanding of the code.

One specific case we’ve noticed is that even if we specify “three” input-output pairs in the prompt sometimes the model generates two or even four such pairs. We know for a fact that GPT 3.5 i.e ChatGPT is not that great with large numbers but seeing it ignore smaller numbers in the prompt is an interesting observation.

3️⃣ “**Follow the instructions carefully. Instructions: Assume the variable's contents as required, …”**

This part of the prompt helps us list the things we **need** the response to follow in its entirety. Here, “Assume variable’s content as required” instructs the model to mock variables as required or instantiate it while coming up with an output for the model as class-level or system-level variables are not present at the function-level. Instead of coming up with manual mocks or stubs. We ask GPT to do it for us.

4️⃣ **The generated response must have the following qualities: concise, stepwise, and specific.**

This part of the prompt helps us tune the response to our required qualities like conciseness (instead of verbose), stepwise, and being as specific as possible (includes details). This is added to ensure the quality of the generated response.

5️⃣ **code: <code>**

This variable contains the method under test as it is from the repository. Without modifications.

5️⃣ **Show calculations and reason about its execution for each of the three input-output pairs.**

We’ve decided to send this prompt after we get a response from GPT giving us input-output pairs because according to the research we’ve seen during the course there were multiple instances where breaking down the prompt into smaller prompts helped the researchers to get the expected answer. This also helps the model as the model has a single deliverable after each major prompt. So, in the first case, it returns input-output pairs and after the second prompt, it reasons about its execution.

⚠️ **“Considering the code snippet in variable “code” and reasoning about the execution in the variable “explanation” given below as an example. code: <code> explanation: <explanation>. For the java function with comments are given in the variable “method” write three inputs and outputs in a table and reason about the execution of the function. Step by step. Be as specific as possible. Assume variable's contents as required.”**

The intuition behind this prompt was that as we do in few-shot learning, giving a few examples to the model helps the model understand the task at hand and the expected output format. As we’ve also seen in many cases the increase in accuracy due to giving an example to model increases by quite a large margin.

However, we did not notice a considerable increase in the quality of output or generated code and also noticed that in some cases the model confuses between the two code snippets and writes an explanation for example code snippet when it’s expected to write an explanation for the prompted method under test. Also, in some cases, the generated explanation was a combination of the two functions. Which was **not** the expected output.

Task 2: Test Generation

🛠️ **For the <LANG> function with comments given in the variable 'code' below, write a unit test for testing the function. Follow the instructions carefully. Instructions: Import all the required packages, Mock all the functions and classes required by the method.**

code: <code>

<response>

Refactor the unit test produced, make sure it's runnable and efficient.

<response>

1️⃣ **For the <LANG> function with comments given in the variable 'code' below, write unit test for testing the function.**

This part of the prompt is quite similar to the one we used for the first task. This prompt describes the task at hand to the model and also specifies the language of the code snippet. *The same reasoning follows for this prompt as in the first task.*

2️⃣ **Follow the instructions carefully. Instructions: Import all the required packages, Mock all the functions and classes required by the method.**

This part of the prompt helps us list the things we **need** the response to follow in its entirety. Here, "**Mock all the functions and classes required by the method."** instructs the model to create helper functions or initialize mock function calls or stub APIs as required to run the test method.

3️⃣ **Refactor the unit test produced, and make sure it's runnable and efficient.**

The intuition behind this is quite simple. irrespective of what the model returned we ask the model to refactor/clean up the generated code snippet. This allows us a couple of chances to test the method and also helps the model clean up any non-runnable artifacts from the produced code snippet.

⚠️ **Passing constructor of the class as context.**

This helped ChatGPT to understand the context to some extent. The generated tests were better than those without this context. But, in Java, some variables could be private and when the generated tests try to access these variables, it gives compile time error.

⛔ **This test will fail. The expected output is incorrect. Can you please regenerate the test with the same inputs?**

ChatGPT continues to generate the wrong expected result when a calculation is involved. Even if we point out the exact error, it still generates the wrong test.

⛔ **The generated test is incorrect. Can you please regenerate the test? Consider the following things while generating the test:**

For functions that call some other functions from the project, the initial prompt gives the wrong output. If we try to provide more context like the function's mock definition, the ChatGPT still fails to generate the test correctly. It even failed in passing the correct number of inputs to the intermediate function calls.

⛔ **The third test in <function_name> has the wrong expected output. Can you please rectify that?**

This prompt hasn't proved to be useful. If we want to ask ChatGPT to regenerate the test with a specific input combination, we'll have to specify the input combination in the prompt. Referencing the previously generated code's input combination doesn't help.

Task 3: Code Refactoring / Semantic Equivalent Code Generation

🛠️ **For the <java/python> function with comments given in the variable 'code' below write a semantically equivalent function. Follow the instructions carefully. Instructions: Method signature should remain identical, Return type of the function should also remain identical, Add comments to the method when required, Add <javadoc/docstring> to the method.**

code: <code>

<response>

Refactor the semantically equivalent function produced, make sure it's runnable and efficient.

<response>

1️⃣ **For the <java/python> function with comments given in the variable 'code' below write a semantically equivalent function.**

This part of the prompt is quite similar to the one we used for the first task. This prompt describes the task at hand to the model and also specifies the language of the code snippet. *The same reasoning follows for this prompt as in the first task.*

2️⃣ **Follow the instructions carefully. Instructions: Method signature should remain identical, Return type of the function should also remain identical, Add comments to the method when required, Add <javadoc/docstring> to the method.**

This part of the prompt helps us list the things we **need** the response to follow in its entirety.

Here, "**Method signature should remain identical"** instructs the model to make sure that the function should remain identical.

The second instruction "**Return type of the function should also remain identical**" asks the model to keep the return type the same as well to ensure semantic equivalence.

The third instruction "**Add comments to the method when required"** ensures that the model also generates comments for the written code so that it's easier to evaluate and understand.

The final instruction "**Add <javadoc/docstring> to the method."** asks the model to create a function level Javadoc or docstring depending on the language to help with understanding the generated code.

3️⃣ **Refactor the semantically equivalent function produced, make sure it's runnable and efficient.**

The intuition behind this is quite simple. irrespective of what the model returned we ask the model to refactor/clean up the generated code snippet. This allows us a couple of chances to test the method and also helps the model clean up any non-runnable artifacts from the produced code snippet.

⚠️ **Sometimes ChatGPT changes the name of the method function**

Sometimes it is observed that ChatGPT can generate a semantically equivalent code but it tends to change the method/function name. This is interesting because when it is asked to do this task is well defined and refactoring generally applies to the method/function body and not the name. Changing the name will break the code. Then again, we don't really know on which data the model behind ChatGPT is trained, or even the fact that it is trained on any code data at all. This observation hints to the fact that it could be trained on code data because in most cases it is able to generate a refactored code for different methods/functions of varying difficulty.

⚠️ **Tends to understand the use of a variable from it's name**

In certain cases, ChatGPT adds some additional conditions to handle boundary cases. For example, one of the java methods takes a CharSequence parameter with the name `creditCode`, and from the code, it is inferred that the length of this sequence has to be `17`. But the original code snippet had no such condition. One guess as to why it adds this if statement is that it tries to derive the meaning from the name `creditCode` and if it has seen this earlier in the training data where it learned that this code is of a specific length, then from the method body it extracts the `17` from the loop and uses this information to create that `if` condition.

Another case where we saw such behavior is when it dealt with DateTime formats. In one of the methods there were multiple custom DateTime formats and ChatGPT was able to correctly guess the format solely by its name and no other context. This is another interesting observation because we can see that ChatGPT tries to make sense of a variable using only its name.

Formatting

🪜 **“Step by Step” / “Stepwise”**

When you provide a prompt to GPT and ask it to generate text "step by step," the model will use its language generation capabilities to provide a sequence of steps or instructions related to the prompt you provided. This ensures that the generated text is formatted nicely in form of bullet points or numbered lists which makes it easier to read and concise. Instead of a verbose explanation in huge paragraphs. Our intuition is that it also forces the model to think about the solution in a stepwise manner.

📄 **“In a tabular manner”**

Similar to the prompt above, this part asks GPT to generate the output and format it into a markdown-compatible table format so that it’s easier to read and concise. For instance, we use this part of the prompt when we ask the model to generate three varied inputs for the method under test.

⚠️ **“display the results using comma separated values”**

This prompt works sometimes and sometimes the results are not well formatted, so we choose to not use this prompt to format the response from GPT.

Results

Interesting Observations

1️⃣ **ChatGPT Hallucinates**

This is one of the most interesting observations we’ve made. We’ve observed that ChatGPT hallucinates. For instance, if we ask the model to consider a link that contains the definition of the class and input, the model understands what we want it to do but then goes on the use the information that is not given in the provided link.

This in our opinion is worse than not knowing as it shows that ChatGPT has no idea if the data it is using is from the given location or not. So, we think ChatGPT is not able to visit links but instead tokenizes the link and finds something similar to the given link but not the actual info.

2️⃣ **Claiming given input as its own work**

In some cases when asked to improve the generated code or reasoning by giving the model extra information. The model responds by claiming that it found the information on its own, which is quite funny! Also, puts into context that GPT might not be aware of what info we give and what it generates on its own.

3️⃣ **Determining the function access type by its name**

The ChatGPT assumed the access type of the functions by their naming conventions. For instance, if the variable is named “_variable_” (single underscore) then it assumes that the variable is protected, and if the variable is named “__variable__” (double underscore) it assumes that the variable is private. This might be true for languages like Python which do not have access specifiers but it is incorrect for languages like Java which have explicit access specifiers.

⚠️ **ChatGPT fails to fetch files from GitHub repositories**

This goes hand in hand with the first observation, in most cases, ChatGPT claims to have found information from the link provided but rarely is the information coherent with what's present in that given link. However, in some cases, it can extract keywords from the links to understand the broad context of the prompt.

5️⃣ **ChatGPT gave preference to NL over PL of the code while generating semantically equivalent code**

One of the code snippets has a wrong mapping of comments and the actual implementation. This caused ChatGPT to give more emphasis to NL part of the data and generated the wrong code.

6️⃣ **Trying two functions in a prompt for all tasks**

We tried giving two functions to the same prompt and interestingly it generated the correct output for both functions for all tasks

7️⃣ **Inserting values in mathematical expression instead of determining their value**

We came across one more interesting observation while generating the tests for the functions using ChatGPT. As we already mentioned that ChatGPT struggles to calculate the correct output when it comes to mathematical calculation, ChatGPT tried to overcome this issue by not evaluating the expression instead leaving it as it is for the compiler. Here is the code snippet from one of the generated tests using ChatGPT:

```
// Call the method and verify the result
double expected = ((3.3 - 1.1)*(7.7 - 5.5)) + ((4.4 - 2.2)*(8.8 - 6.6));
double actual = myClass.scal(pt1, pt2, pt3, pt4);
assertEquals(expected, actual, 0.001);
```

Java

Task 1: Reasoning

	Easy	Medium	Hard
Correct	11	8	3
Partially Correct	3	3	9
Incorrect	4	4	5
Total	**18**	**15**	**17**

Task 2: Test Generation

	Easy	Medium	Hard
Correct	11	2	3
Partially Correct	4	8	9
Incorrect	3	5	5
Total	**18**	**15**	**17**

Task 3: Refactoring

	Easy	Medium	Hard
Correct	13	7	6
Partially Correct	0	3	4
Incorrect	5	5	7
Total	**18**	**15**	**17**

Python

Task 1: Reasoning

	Easy	Medium	Hard
Correct	15	3	0
Partially Correct	13	7	5
Incorrect	4	2	2
Total	**32**	**12**	**7**

Task 2: Test Generation

	Easy	Medium	Hard
Correct	6	2	0
Partially Correct	24	6	2
Incorrect	2	4	5
Total	**32**	**12**	**7**

Task 3: Refactoring

	Easy	Medium	Hard
Correct	27	9	4
Partially Correct	4	2	2
Incorrect	1	1	1
Total	**32**	**12**	**7**

Conclusion

This milestone aimed to perform Prompt Engineering using ChatGPT on code-related tasks such as code reasoning, test generation, and semantically equivalent code generation. It used Python and Java functions from the CodeSearchNet dataset, totaling around 50 data points for each language. The difficulty level of each data point was labeled based on its input parameters, i.e., primitive or non-primitive, and call to other methods.

Prompt engineering was performed on the data points to identify the most effective prompts for ChatGPT. Observations were noted and used to finalize the prompts. The finalized prompts were then executed on 100 data points, and the output was captured in markdown files in the respective language folders. The generated outputs were evaluated based on their reasoning and input-output combination, running the generated tests in the actual project environment, and running the tests on semantically equivalent generated code.

The results showed promising outcomes for all three tasks in both Java and Python. However, as the code's complexity increased, the results showed a declining curve. Code with mathematical calculations had worse results in code reasoning and test generation compared to code without mathematical calculations. Nevertheless, ChatGPT generated semantically equivalent code for all difficulty levels, and for code with a hard difficulty level, where non-primitive data and dependencies on other functions were present, the performance of code reasoning and test generation declined.

Overall, the results suggest that developers can use ChatGPT for various code-related tasks, but manual verification and changes are necessary before using the generated output. The project highlights the need for improvement in the dataset with mathematical calculations.