MIE451 LLM Prompt Engineering for Natural Language Tasks

Q1. Task Definition and Motivation

Summarization involves filtering information from a source into a condensed form. This can be achieved through extractive summarization, which selects key phrases or segments from the source text to create a summary, or abstractive summarization, which involves understanding the source text's meaning and generating entirely new text that conveys the same information succinctly.

In this project, I will explore the subtask of code summarization or code comment generation to enhance the readability and maintainability of the code.This task belongs to abstractive summarization, as it demands a deep understanding of the code's functionality and purpose. The input for this task will be code snippets in Java and Python, provided as strings without any formatting elements like indents and new lines and the objective of GPT is to generate a succinct, one-line summary that effectively communicates the function of each code snippet.

The metric I am going to use is METEOR (Metric for Evaluation of Translation with Explicit Ordering). It is originally developed for assessing machine translation quality, and compares the generated text to one or more reference summaries. It compares the words between the generated summary and reference summaries based on exact, stem, synonym, and paraphrase matches, providing a more comprehensive understanding of semantic similarity which is how I am going to evaluate the performance of ChatGPT. METEOR also incorporates a penalty for longer generated summaries that deviate from the reference length, promoting brevity and precision that are key aspects in code comments. Additionally, METEOR balances between precision and recall, making it an ideal metric for this task. [1][2]

The task of code summarization or code comment generation is an important component of a Decision Support System (DSS) due to its role in enhancing code readability and maintainability, which are crucial for effective decision-making in software development. Accurate and succinct code summaries assist developers in quickly understanding and evaluating the functionality of various code segments, thereby facilitating more informed decisions. For instance, in a large-scale software project, developers can easily comprehend and modify code written by others, reducing the time and effort required for code reviews and debugging. Similarly, in the field of automated code generation, where algorithms often produce complex code, such summarizations can help in assessing the algorithm's output, ensuring that the generated code aligns with the intended functionality and adheres to best practices. By integrating code summarization into a DSS, organizations can improve their software development lifecycle, leading to more efficient, reliable, and maintainable software solutions.

Q2. Data Curation

Table 1. First Half of Curated Data

Index	Input	Suggested Output
1	def rindex (lst , value) : lst . reverse ( ) i = lst . index (value) lst . reverse ( ) return len (lst) - i - 1	return the last index of the value in the list
2	Object function (String arg0 , Object arg1) { Object loc0 = this . valueMap . get (arg0) ; if (loc0 != null) { return loc0 ; } return arg1 ; }	returns the object associated with the specified name of the child 's node

3	matrix = np . zeros ((numUsers , numItems) , dtype = np . int8) for (index , userID , itemID , rating , timestamp) in dataset . itertuples ( ) : matrix [userID - 1 , itemID - 1] = rating	create a user-item matrix from the dataset
4	Integer function ( ) { return frameRate ; }	returns the frame rate value for the encoding process
5	void function (String arg0 , String arg1) { _parameters . put (arg0 , arg1) ; }	sets the value of a parameter

Table 1 above provides representative examples of the curated dataset. The entire dataset has been pre-shuffled and the second half is listed in the Appendix.

The dataset comprises two primary sources. Firstly, half of the data (5 out of 10 samples) was retrieved from the CodeXGLUE_CONCODE dataset and is in Java. The CONCODE dataset was collected from Java files containing method documentation from public Github repositories and it is the first dataset to condition on the environment of the output code. It has contained examples from several domains, ranging from LookupCommand to ImageSequenceWriter [3]. This makes the dataset ideal for code summarization while taking consideration of context.

The subset (5/10) was curated using a random selection process, where a page from the dataset was chosen randomly, followed by the selection of a sample code from that page. Some input from the sample code was used directly in my dataset, while the remaining samples were modified by removing function definitions and return values. This ensures that the dataset aligns with the task of code summarization, rather than function summarization. The output was also modified to include only the first sentence, providing concise explanations in line with our requirements. This was necessary as the original dataset's outputs were extensively complex.

The latter half of the dataset consists of Python code snippets, obtained from various projects I have previously worked on. To align with the first half's distribution, three of these snippets include function definitions, while two do not. In keeping with the format of the CodeXGLUE dataset, these snippets have been reformatted by removing newlines and indentations, and spaces have been added between words and symbols to improve readability and consistency. The annotations (outputs) for these snippets were created using Copilot, which considers the entire project folder it resides in. This enables Copilot to generate contextually relevant comments, overlapping with the in-context nature of the CONCODE dataset. Table 2 below outlines the evaluation criteria for summarization generated by CoPilor. For a summarization to be deemed acceptable, it must adhere to the criteria specified in Table 2.

Table 2. Rubrics on Evaluating the Comment Outputs by CoPilot	
Accuracy	Does the summary accurately describe the function and purpose of the code?
Completeness	Does it cover all essential aspects of the code without omitting crucial details?
Relevance	Are the comments relevant to the code's functionality?
Clarity	Is the language clear and understandable?

The evaluation of CoPilot's code outputs was conducted by two classmates, as detailed in Table 3. It's important to note that the calculation of Kappa, a statistical measure of inter-rater agreement, is not feasible in scenarios where there is unanimous agreement among the assessors regarding the accuracy of the outputs. This is because Kappa is designed to measure the extent of agreement beyond what would be expected by chance, and complete consensus leaves no room for evaluating this aspect.

Table 3. Interjudge Agreement of Whether the Outputs of CoPilot are Acceptable		
Code snippet index	Student 1	Student 2
1	Yes	Yes
3	Yes	Yes
6	Yes	Yes
7	Yes	Yes
9	Yes	Yes

Q3. Prompt Styles

Prompt 1 ---- Zero-Shot prompt:

"""
For each code snippet below, write a one line succinct summary of what they do.
Code Snippets:
####
Code snippet 1: [formatted code for snippet 1]
####
Code snippet 2: [formatted code for snippet 2]
####
And more
"""

Description:

Zero-shot prompting is a technique in machine learning where a model is presented with a task or question without any prior tailored examples or training for that specific task. In this context, the model depends entirely on its existing knowledge and comprehension to formulate a response. When it comes to interpreting code snippets, utilizing a zero-shot approach entails asking ChatGPT to summarize these snippets without having received specific training on these exact examples or comparable tasks. ChatGPT must leverage its pre-trained knowledge base, encompassing an understanding of various programming languages, code structures, and algorithm concepts, to analyze and elucidate these snippets. In the given prompt, the structure is arranged that it started with clear instructions followed by the distinct components, each separated by the delimiter '####', ensuring clarity and coherence in the task.

Prompt 2 ---- Few-Shot prompt:

"""
For each code snippet below, write a one line succinct summary of what they do.

Examples:
""
####
Sample Snippet 1: Object function ( ) { String loc0 = null ; if (this . hasNext ( )) { loc0 = this . segment ; this . segment = null ; } if (loc0 == null) { throw new NoSuchElementException ( ) ; } return loc0 ; }
####
Sample Snippet 2: n = pts_des . shape [1] J = np . empty ((0 , 6)) for i in range (n) : tmp_J = ibvs_jacobian (K , pts_obs [: , i] . reshape (- 1 , 1) , zs [i]) J = np . vstack ((J , tmp_J))
####

Output:
Snippet 1: retrieves the next available sql segment as a string
Snippet 2: Computes the Jacobian matrix for each observed point and stack them vertically in J
""
Code Snippets:
####
Code snippet 1: [formatted code for snippet 1]
####
Code snippet 2: [formatted code for snippet 2]
####
And more
""""

<u>Description:</u>
Few-shot prompting is a method where the model is given a small number of examples to help it understand and perform a specific task. This approach is particularly effective for models that need to adapt to tasks for which they were not explicitly trained.

In this context of code summarization, initially, the prompt instructs the model to write a one-line summary for each code snippet. This is followed by a series of examples, each separated with '####'. These examples include a sample code snippet along with its concise summary, providing the model with a clear template of what is expected in the output. After the model is acquainted with the task through the instruction and examples, it encounters new code snippets. Here, the model is expected to apply the pattern it learned from the examples to generate similar summaries for these new snippets. This structured approach enables the model to efficiently grasp and execute a specific task with minimal prior exposure.

<u>Prompt 3 ---- Chain of Thoughts Prompt:</u>
""""
Examine the following code snippet. First, identify the main components and their roles in the code. Next, determine what the overall function of the code is based on these components. Finally, summarize the purpose of this code in one concise sentence.

Code Snippets:

####
Code snippet 1: [formatted code for snippet 1]
####
Code snippet 2: [formatted code for snippet 2]
####
And more
"""

Description:
The Chain of Thought prompt is a method designed to elicit a detailed reasoning process from the model. This technique guides the model to not only produce an answer but also to articulate the step-by-step thought process leading to that conclusion. In the above prompt, the model is directed first to identify the key components and their roles within each snippet, then to ascertain the overall function of the code based on these elements, and finally, to concisely summarize the code's purpose. By laying out all the instructions at the beginning, the model gains a clear, comprehensive understanding of the task's scope, allowing for a more efficient, logically coherent processing of the information. This structure reduces context switching and ensures a cohesive flow of thought, making the model's analysis and the resulting explanations more thorough and interpretable than a multi-stage chain-of-thought prompting.

Q4. Results and Analysis

Table 4. METEOR Scores for Code Snippets using Different Prompt Style			
Index	Zero Shot Score	Few Shot Score	Chain-of-Thought Score
1	0.5350	0.6626	0.9308
2	0.0735	0.0730	0.1786
3	0.2564	0.2632	0.6395
4	0.4133	0.4043	0.5437
5	0.3024	0.3872	0.3125
6	0.4482	0.4845	0.7295
7	0.3209	0.3855	0.1358
8	0.0766	0.1481	0.0755
9	0.3807	0.6657	0.4523
10	0.2041	0.2041	0.1961
Average	0.3011	0.3678	0.4194

*The above scores are being calculated using the METEOR metric mentioned in section 1. The scores are generated by comparing the summary returned by ChatGPT and the suggested summary generated by copilot.

**In the referenced paper [1], the highest correlation with human judgment at the sentence level was 0.403. Given this finding, setting a threshold score of 0.5, or potentially even 0.4, is justifiable. Therefore, scores above 0.5 indicate relevance, while those below 0.5 signify a lack of relevance.

The table above illustrates that the average METEOR scores increased gradually as the prompting style shifted from zero-shot to few-shot, and then to chain-of-thought. This upward trend is evident not only overall but also in the scores of some specific individual code snippets. However, it's noteworthy that some snippets, initially scoring lower, did not exhibit score improvements corresponding with the changes in prompt style such as snippet 2, 5, or 10. In contrast, snippets 3, 6, 8, and 9 displayed intriguing trends in their scoring patterns.

Table 5. The Outputs of Three Successful and Failed Results in Chain-of-Thought Style			
Three successful results			
Index	Suggested Output	ChatGPT Output	Score
1	return the last index of the value in the list	The code's purpose is to return the last index of the given value in the list.	0.9308
3	create a user-item matrix from the dataset	The purpose is to create a user-item rating matrix from a dataset	0.6395
6	get the topic name from the file path	The purpose is to obtain the name of the parent directory from the file path.	0.7295
Three failed results			
5	sets the value of a parameter	The purpose is to set key value pairs in parameters.	0.3125
8	tests if the content length set in the stream equals the bytes read from the stream, if any exception is thrown, then the test fails.	The purpose appears to be related to managing input streams and handling exceptions	0.0893
9	generate rotation matrix from roll, pitch, yaw Euler angles	The purpose is to calculate a direction cosine matrix from roll pitch yaw angles	0.4523

The success of Snippet 1 in code summarization is largely due to its straightforward written format. As a utility function, it clearly defines its variables and maintains a simple structure, which facilitates easier interpretation and summarization by GPT.

In contrast, the success of Snippet 3 appears to stem from the chain of thought approach, especially given its lower scores in the previous two styles. The chain of thought prompting aids in breaking down complex code into more manageable segments, enhancing contextual understanding and simplifying the overall comprehension process.

Snippet 6 achieved a score of 0.7, partly because it is a simple function that retrieves the topic name given a file path. Additionally, GPT used the word ‘obtain’, a synonym of the original verb ‘get’, in its chain of thoughts. This contrasts with its previous attempts in zero-shot or few-shot learning, where it didn't use words from the synonym set of ‘get’ in the NLTK module.

The failure of Snippet 5 might be attributed to inherent ambiguity in the code and the Java language. Such ambiguity can lead to different interpretations by ChatGPT, resulting in vague summaries.

The consistent failure of Snippet 8 across all three prompt styles is significant. This failure is primarily due to the inherent complexity of the code, which is designed to check the length of bytes in the input stream. This complexity poses a challenge for ChatGPT, making it difficult for the model to fully grasp the essence of the code. Unlike CoPilot, ChatGPT operates without any additional context. As a result, it can only provide a superficial explanation of the code, lacking a deeper understanding.

Unlike Snippet 6, which succeeded in using the 'right' word, Snippet 9's failure in the chain of thought approach, following its success in few-shot learning, could be attributed to using 'calculate' as a less precise synonym for 'generate', compared to 'computes'. Additionally, the summary for Snippet 9 was a full sentence instead of a brief phrase, which may have affected its evaluation, as both length and format are important factors in METEOR scoring.

Q5. Limitations

Understanding of Code Context or Integration of Context:
When deployed for project-wide code summarization, LLMs may struggle to understand the interconnected roles and dependencies of different code segments. For instance, a function's role might be clear within its file, but its relevance to the broader project could make it hard for the model. This often leads to technically accurate but contextually incomplete summaries. A hierarchical summarization method, derived from this [4], can effectively address this issue. This approach involves the model summarizing individual components like functions and classes initially, before integrating these into a comprehensive summary of larger structures.

Accurate Understanding of Code without Context:
When summarizing isolated code snippets, LLMs might produce summaries that are either inaccurate or too general, especially in contexts where broader context is unavailable due to data security concerns in large corporations. This challenge primarily stems from the lack of external references or an understanding of the snippet's role in a larger codebase. To mitigate this, training the model with a diverse array of self-contained code examples demonstrating various functionalities is crucial. Additionally, incorporating interactive user feedback allows users to provide necessary context or clarify the code's intended use. Employing natural language processing techniques that specialize in extracting information from limited data can further refine the model's performance. [5]

References

[1] Satanjeev Banerjee and Alon Lavie. 2005. METEOR: An Automatic Metric for MT Evaluation with Improved Correlation with Human Judgments. In Proceedings of the ACL Workshop on Intrinsic and Extrinsic Evaluation Measures for Machine Translation and/or Summarization, pages 65–72, Ann Arbor, Michigan. Association for Computational Linguistics.

[2] “Meteor - A hugging face space by evaluate-metric,” METEOR - a Hugging Face Space by evaluate-metric, https://huggingface.co/spaces/evaluate-metric/meteor.

[3] S. Iyer, I. Konstas, A. Cheung, and L. Zettlemoyer, “Mapping language to code in programmatic context,” Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing, 2018. doi:10.18653/v1/d18-1192

[4] J. Christensen, S. Soderland, G. Bansal, and Mausam, “Hierarchical summarization: Scaling up multi-document summarization,” Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), 2014. doi:10.3115/v1/p14-1085

[5] A. Intelligence, “How can you use transformers for summarizing text?,” How to Use Transformers for Text Summarization, https://www.linkedin.com/advice/0/how-can-you-use-transformers-summarizing-l933e#what-are-the-challenges-of-transformers-for-summarizing-text?.

Appendix

Table 6. Second Half of Curated Data		
Index	Input	Suggested Output
6	def get_topic_name (file_path) : return file_path . parent . name	get the topic name from the file path
7	if not self . cursor . rowcount : return [ ] self . cursor . scroll (0 , 'absolute') return self . cursor . fetchall ( )	return all of the rows that are in the result set if cursor has rows else return empty list
8	InputStream loc0 = new ByteArrayInputStream (sampleData . getBytes ( )) ; LengthCheckInputStream loc1 = new LengthCheckInputStream (loc0 , sampleData . getBytes ( ) . length , INCLUDE_SKIPPED_BYTES) ; try { StreamUtils . consumeInputStream (loc1) ; } catch (Exception loc2) { fail ( ) ; } loc1 . close ( ) ;	tests if the content length set in the stream equals the bytes read from the stream, if any exception is thrown, then the test fails.
9	def dcm_from_rpy (rpy) : cr = np . cos (rpy [0]) . item ( ) sr = np . sin (rpy [0]) . item ( ) cp = np . cos (rpy [1]) . item ( ) sp = np . sin (rpy [1]) . item ( ) cy = np . cos (rpy [2]) . item ( ) sy = np . sin (rpy [2]) . item ( ) return np . array ([[cy * cp , cy * sp * sr - sy * cr , cy * sp * cr + sy * sr] , [sy * cp , sy * sp * sr + cy * cr , sy * sp * cr - cy * sr] , [- sp , cp * sr , cp * cr]])	generate rotation matrix from roll, pitch, yaw Euler angles
10	server = new ServerSocket (0) ; startPython ( ) ;	starts the python script