CSCI 420 Assignment 3

Lynelle Chen
lschen01@wm.edu

Alex Zhou
ajzhou02@wm.edu

Reported are a few problems that utilized different prompt engineering strategies compared to a control strategy of zero-shot given to ChatGPT-4o and Gemini 2.0 Flash. The strategies here include prompt-chaining, few-shot, and role playing. For the sake of keeping the report concise, we have chosen two problems to list in detail and included individual analysis of the ChatGPT and Gemini's responses to both our control prompting strategy in zero-shot and other prompting strategies in prompt-chaining and role-playing, totaling to 8 responses. At the very bottom of our report, we have a table of the remaining evaluation results and some analysis for all the problems and outputs including both unlisted and listed.

Prompt-Chaining vs Zero-shot for ChatGPT-4o

Prompt-Chaining

Goal	Finding the error in a JavaScript function for if a number is even
Model	GPT-4o
Temperature: 0.7	**Token Limit:** 1024

Prompt	1. Give me just the error in this code, do not attempt to fix it: function isEven (n) { return n % 2; // Returns 1 or 0 , not true / false }
	2. What is the fix for this code

Output	1. The error is that the function returns `1` or `0` instead of a boolean `true` or `false`.
	2. The fix is to explicitly return a boolean value:
	function isEven(n) {
	return n % 2 === 0;
	}

Zero-shot

Goal	Finding the error in a JavaScript function for if a number is even
Model	GPT-4o
Temperature: 0.7	**Token Limit:** 1024
Prompt	What is the error in this code; function isEven (n) { return n % 2; // Returns 1 or 0 , not true / false }

Output	The main issue with the function is that it doesn't return a **boolean** value (`true` or `false`), even though the function name `isEven` suggests it should.
	
	Here's the original code:
	
	function isEven(n) {
	
	return n % 2; // Returns 1 or 0, not true / false
	
	}
	
	Explanation:
	
	• `n % 2` returns:
	  ○ `0` if `n` is even
	  ○ `1` if `n` is odd
	
	But:
	
	• `0` is **falsy** and `1` is **truthy**, so it *might* work in some conditional expressions.
	• However, it's misleading: `isEven(2)` returns `0`, not `true`, which can cause confusion or bugs in logic expecting a boolean.
	
	Fix:
	
	You should return a boolean explicitly:
	
	function isEven(n) {
	
	return n % 2 === 0;
	
	}
	
	Now:
	
	• `isEven(2)` returns `true`
	• `isEven(3)` returns `false`
	
	✅ Clear and semantically accurate.

Prompt-Chaining vs Zero-shot for Gemini

Prompt-Chaining

Goal	Finding the error in a JavaScript function for if a number is even
Model	Gemini 2.0 Flash
Token Limit	1024
Prompt	1. The error is that the function `isEven` returns the remainder of the division by 2 (which will be 0 or 1) instead of a boolean value (`true` or `false`). 2. What is the fix for this code

Output	1. The error is that the function `isEven` returns the remainder of the division by 2 (which will be 0 or 1) instead of a boolean value (`true` or `false`). 2. function isEven (n) { return n % 2 === 0; // Returns true if the remainder is 0, false otherwise } The fix is to compare the result of the modulo operation (`n % 2`) to `0`. If the remainder is `0`, then the number is even, and the expression evaluates to `true`. Otherwise, it evaluates to `false`.

Zero-shot

Goal	Finding the error in a JavaScript function for if a number is even
Model	Gemini 2.0 Flash
Token Limit	1024

Prompt	What is the error in this code;
	
	function isEven (n) {
	
	return n % 2; // Returns 1 or 0 , not true / false
	
	}

Output	Ah, you've spotted a common little pitfall in JavaScript! You're absolutely right. The error lies in the fact that the modulo operator (`%`) returns the **remainder** of a division, not a boolean `true` or `false`. Here's the breakdown: ● When `n` is even, `n % 2` will indeed evaluate to `0`. ● When `n` is odd, `n % 2` will evaluate to `1`. JavaScript treats `0` as a "falsy" value and `1` (or any non-zero number) as a "truthy" value in conditional contexts. While this might sometimes work unintentionally in `if` statements, it's not the correct way to explicitly return a boolean indicating evenness. **The fix is to explicitly compare the result of the modulo operation to `0`:** function isEven (n) { return n % 2 === 0; // Returns true if the remainder is 0, false otherwise } By using the strict equality operator (`===`), you ensure that the function returns a clear and unambiguous boolean value (`true` or `false`) based on whether the remainder of the division by 2 is exactly equal to 0. Good catch! This kind of attention to detail is key in programming. 😊

Analysis of Prompt-Chaining vs Zero-shot

For this simple Python problem presented in the prompt, the code outputs for both prompt types for both models are the same. The explanation provided for zero-shot, however, is much more verbose than prompt-chaining as we requested the model to only do one task. When not cued to directly identify an error, the zero-shot prompt for both models results in a long explanation for finding the issue at hand. BLEU score: GPT - `0.1951;` Gemini - `1907`

Comparison between ChatGPT and Gemini

The Gemini model notably is set to automatically adjust its temperature to the type of prompt, while GPT is set to 0.7. For this problem, the most clear deviance is in the zero-shot response because of the volume of text available; Gemini highlights the use of the modulo operation and exhibits more time on behavioral type text not related to code explanation than ChatGPT. BLEU score: Zero-shot - `0.0914;` Chain - `0.0095`

Role-Playing vs Zero-shot for ChatGPT-4o

Role-playing

Goal	Code Completion and Robustness Enhancement of Python function
Model	GPT-4o
Temperature: 0.7	**Token Limit:** 1024
Prompt	You are an expert Python developer specializing in refactoring code, finishing code, and fixing errors. You have been assigned the following task: Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly # file_processor .py - Incomplete script for processing text files import string def load_file (filepath) : with open (filepath , 'r') as f : return f . readlines () def clean_line (line) : # TODO : Remove punctuation and make lowercase pass def count_words (lines) : word_counts = {} for line in lines: clean = clean_line (line) for word in clean.split () : word_counts [word] = word_counts . get (word , 0) + 1 return word_counts def main () : filepath = 'input .txt' lines = load_file (filepath) counts = count_words (lines) for word , count in sorted (counts . items ()) : print (f"{ word }: { count }") if __name__ == '__main__ ': main ()

Output	Here is the **completed and refactored version** of your Python script. It includes:
	• Proper indentation and spacing
	• A functional `clean_line()` implementation using `str.translate()` to remove punctuation
	• Corrected `main()` invocation and path formatting
	# file_processor.py - Complete script for processing text files
	import string
	
	def load_file(filepath):
	  with open(filepath, 'r') as f:
	  return f.readlines()
	
	def clean_line(line):
	  # Remove punctuation and convert to lowercase
	  return line.translate(str.maketrans('', '', string.punctuation)).lower()
	
	def count_words(lines):
	  word_counts = {}
	  for line in lines:
	  clean = clean_line(line)

	for word in clean.split():
	word_counts[word] = word_counts.get(word, 0) + 1
	return word_counts
	
	def main():
	filepath = 'input.txt'
	lines = load_file(filepath)
	counts = count_words(lines)
	for word, count in sorted(counts.items()):
	print(f"{word}: {count}")
	
	if __name__ == '__main__':
	main()
	Example Output:
	If `input.txt` contains:
	Hello, world! Hello Python.
	The output will be:
	hello: 2
	python: 1
	world: 1
	Let me know if you want to add stopword filtering or output to a file!

Zero-shot

Goal	Code Completion and Robustness Enhancement of Python function
Model	GPT-4o
Temperature: 0.7	**Token Limit:** 1024
Prompt	Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly # file_processor .py - Incomplete script for processing text files import string def load_file (filepath) : with open (filepath , 'r') as f : return f . readlines () def clean_line (line) : # TODO : Remove punctuation and make lowercase pass def count_words (lines) : word_counts = {} for line in lines: clean = clean_line (line) for word in clean.split () : word_counts [word] = word_counts . get (word , 0) + 1 return word_counts def main () : filepath = 'input .txt' lines = load_file (filepath) counts = count_words (lines) for word , count in sorted (counts . items ()) : print (f"{ word }: { count }") if __name__ == '__main__ ': main ()

Output	Here's a completed and corrected version of your `file_processor.py` script that:
	● Cleans each line by removing punctuation and converting to lowercase.
	● Properly counts word frequencies.
	● Fixes minor syntax/indentation issues.
	
	``
	# file_processor.py - Script for processing text files
	
	import string
	
	def load_file(filepath):
	    with open(filepath, 'r') as f:
	        return f.readlines()
	
	def clean_line(line):
	    # Remove punctuation and convert to lowercase
	    translator = str.maketrans('', '', string.punctuation)
	    return line.translate(translator).lower()
	
	def count_words(lines):
	    word_counts = {}
	    for line in lines:
	        clean = clean_line(line)
	        for word in clean.split():
	            word_counts[word] = word_counts.get(word, 0) + 1
	    return word_counts
	
	def main():
	    filepath = 'input.txt'
	    lines = load_file(filepath)
	    counts = count_words(lines)
	    for word, count in sorted(counts.items()):
	        print(f"{word}: {count}")
	``

	if __name__ == '__main__': main() **Changes made:** • Fixed indentation errors (especially in `count_words` and `main()`). • Replaced `input .txt` with `input.txt` (removed extra spaces). • Implemented the `clean_line` function to remove punctuation and make everything lowercase. • Removed unnecessary spacing around punctuation and in function calls. Let me know if you want to enhance this further (e.g., ignore stopwords, output to file, etc.)!

Role-Playing vs Zero-shot for Gemini

Role-Playing

Goal	Code Completion and Robustness Enhancement of Python function
Model	Gemini 2.0 Flash
Token Limit	1024
Prompt	You are an expert Python developer specializing in refactoring code, finishing code, and fixing errors. You have been assigned the following task: Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly # file_processor .py - Incomplete script for processing text files import string def load_file (filepath) : with open (filepath , 'r') as f : return f . readlines () def clean_line (line) : # TODO : Remove punctuation and make lowercase pass def count_words (lines) : word_counts = {} for line in lines: clean = clean_line (line) for word in clean.split () : word_counts [word] = word_counts . get (word , 0) + 1 return word_counts def main () : filepath = 'input .txt' lines = load_file (filepath) counts = count_words (lines) for word , count in sorted (counts . items ()) : print (f"{ word }: { count }") if __name__ == '__main__ ': main ()

Output	Alright, let's get this file processing script working smoothly! You're on the right track. We just need to fill in the clean_line function. Here's the completed script:
	
	Python
	
	# file_processor.py - Incomplete script for processing text files
	
	import string
	
	def load_file(filepath):
	
	with open(filepath, 'r') as f:
	
	return f.readlines()
	
	def clean_line(line):

```
    # Remove punctuation
    line = line.translate(str.maketrans('', '', string.punctuation))
    # Make lowercase and remove leading/trailing whitespace
    return line.lower().strip()

def count_words(lines):
    word_counts = {}
    for line in lines:
        clean = clean_line(line)
        for word in clean.split():
            word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts

def main():
    filepath = 'input.txt'
    lines = load_file(filepath)
    counts = count_words(lines)
    for word, count in sorted(counts.items()):
        print(f"{word}: {count}")

if __name__ == '__main__':
    main()
```

Here's what I did to complete the `clean_line` function:

1. **Removed Punctuation:** I used the `string.punctuation` constant, which contains a string of all punctuation marks. Then, `str.maketrans('', '', string.punctuation)` creates a translation table that maps each punctuation character to `None` (effectively deleting it). Finally, `line.translate()` applies this table to the input `line`.
2. **Made Lowercase:** I used the `.lower()` method to convert the entire line to lowercase. This ensures that words like "The" and "the" are counted as the same word.
3. **Removed Whitespace:** I added `.strip()` to remove any leading or trailing whitespace from the line after converting it to lowercase. This prevents empty strings or words with extra spaces from being counted.

Now, when you run this script with an `input.txt` file, it will read the lines, clean them by removing punctuation and converting them to lowercase, and then accurately count the frequency of each word.

For example, if your `input.txt` file contains:

	The quick brown fox, jumps over the lazy dog. The dog barked loudly! The output of the script will be: barked: 1 brown: 1 dog: 2 fox: 1 jumps: 1 lazy: 1 loudly: 1 over: 1 quick: 1 the: 2 Let me know if you have any other code you'd like me to work on!

Zero-shot

Goal	Code Completion and Robustness Enhancement of Python function
Model	Gemini 2.0 Flash
Token Limit	1024
Prompt	Complete the following file-processing script. The goal is to clean each line, remove punctuation, and count word frequencies correctly # file_processor .py - Incomplete script for processing text files import string def load_file (filepath) : with open (filepath , 'r') as f : return f . readlines () def clean_line (line) : # TODO : Remove punctuation and make lowercase pass def count_words (lines) : word_counts = {} for line in lines: clean = clean_line (line) for word in clean.split () : word_counts [word] = word_counts . get (word , 0) + 1 return word_counts def main () : filepath = 'input .txt' lines = load_file (filepath) counts = count_words (lines) for word , count in sorted (counts . items ()) : print (f"{ word }: { count }") if __name__ == '__main__ ': main ()
Output	Here's a completed and corrected version of your `file_processor.py` script that: ● Cleans each line by removing punctuation and converting to lowercase. ● Properly counts word frequencies. ● Fixes minor syntax/indentation issues. # file_processor.py - Script for processing text files import string

```
def load_file(filepath):
    with open(filepath, 'r') as f:
        return f.readlines()


def clean_line(line):
    # Remove punctuation and convert to lowercase
    translator = str.maketrans('', '', string.punctuation)
    return line.translate(translator).lower()


def count_words(lines):
    word_counts = {}
    for line in lines:
        clean = clean_line(line)
        for word in clean.split():
            word_counts[word] = word_counts.get(word, 0) + 1
    return word_counts


def main():
    filepath = 'input.txt'
    lines = load_file(filepath)
    counts = count_words(lines)
    for word, count in sorted(counts.items()):
        print(f"{word}: {count}")


if __name__ == '__main__':
    main()
```

Changes made:

- Fixed indentation errors (especially in `count_words` and `main()`).

	• Replaced `input .txt` with `input.txt` (removed extra spaces). • Implemented the `clean_line` function to remove punctuation and make everything lowercase. • Removed unnecessary spacing around punctuation and in function calls. Let me know if you want to enhance this further (e.g., ignore stopwords, output to file, etc.)!

Analysis of Role-Playing vs Zero-shot

For both models when assigned the role of being an expert Python developer, they included a key difference in their responses for the zero-shot prompt by adding in an actual example runthrough of an input.txt file and the script's output. There is no significant difference between the code outputs between prompt strategy and model. The prompting BLEU score is: GPT - `0.1783;` Gemini - `0.0441` and the model BLEU score(s) are:`0.0174; 0.1597` respectively.

BLEU Scores per Task between Models:

Task	2	4	5	6	8	9	11	12
Zero-shot	0.0174	0.0149	0.0889	0.0535	0.025	0.0256	0.1857	0.0881
Few-shot	0.1597	0.0442	0	0.1064	0.0115	0.0272	0.016	0.1111
Task	13	14	15	16	17	19	20	21 (role)
Zero-shot	0.1691	0.0507	0.051	0.0663	0.0771	0.0501	0.0914	0.1286
Few-shot	0.1411	0.0708	0.0822	0.0781	0.0906	0.1492	0.0095	0.15

BLEU Scores per Task between Prompt Strategies:

Task	2	4	5	6	8	9	11	12
GPT	0.1783	0.2151	0.2901	0.2714	0.1551	0.3142	0.0862	0.2747
Gemini	0.0441	0.1069	0	0.0465	0.2019	0.2134	0.2151	0.1265
Task	13	14	15	16	17	19	20	21 (role)
GPT	0.4869	0.2954	0.2765	0.4489	0.2963	0.1624	0.2538	0.4838
Gemini	0.331	0.257	0.0031	1.0	0.2102	0.1292	0.1907	0.2696

Analysis of Trends for Applicable Problems (code generation / bug-fixing)

For the strategies employed (mainly few-shot in the table) such as few-shot, chain prompting, and role playing, the BLEU scores between models on average scored higher than what we considered the “control” in zero-shot prompting. We believe that due to added context with these prompting strategies the model has more confidence in its outputs and therefore the generated response is more similar and has a higher BLEU score.

Another general trend we noticed is between the two models we looked at, GPT and Gemini. GPT regardless of the different types of prompting strategies used has a higher BLEU score between strategies than Gemini. Upon individual review of the problems and their outputs, it seems that Gemini places more resources on human interaction; we could not control for temperature with the Gemini 2.0 Flash model used, our research into its temperature showed that Gemini would automatically adjust to the type of prompt given to it.

While most of the code evaluated looked almost identical, we believe that few-shot prompting, namely maintaining the same type of few-shot prompt with the exact same examples, is expected to have a higher BLEU score. Even with a small sample size, the BLEU scores across the two models for role-playing prompting increases as well, although this is surprisingly not the case for chain prompting.