

Task 21: Full-File Bug Detection and Refactoring

Prompting Strategy 1: Zero Shot Prompt

Prompt: Analyze the following utility script. Identify any potential logic issues or design flaws. Then, refactor it for better readability, correctness, and safety.

Model 1 (GPT-4) Output:

```
import csv

def read_csv(filepath):
    with open(filepath, 'r', encoding='utf-8') as f:
        return list(csv.reader(f))

def summarize_column(data, index):
    try:
        values = [float(row[index]) for row in data[1:] if row[index]]
    except (ValueError, IndexError):
        raise ValueError("Invalid data or column index.")

    if not values:
        raise ValueError("No valid numeric data found.")

    total = sum(values)
    avg = total / len(values)
    return total, avg

def main():
    filepath = 'data.csv'
    data = read_csv(filepath)
    try:
        total, avg = summarize_column(data, 1)
        print("Total:", total)
        print("Average:", avg)
    except Exception as e:
        print("Error:", e)

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

Model 2 (Gemini) Output

Analysis and Refactoring:

The original script had potential issues with error handling and flexibility. The `read_csv` function was improved to include checks for file existence and to catch `IOError`, `OSError`, and `csv.Error` exceptions, returning an empty list in case of failure. The `summarize_column` function was also enhanced to handle empty data, out-of-bounds column indices (returning `None`, `None`), and non-numeric values by skipping them and printing a warning to standard error. If a column contains no numeric data, it returns 0, 0.

The main function was modified to accept the file path and column index as command-line arguments, providing more flexibility. Default values are used if the arguments are not provided. The script now prints error messages to

standard error (sys.stderr) and exits with a non-zero status code using sys.exit(1) when appropriate, indicating an error. The code also uses f-strings for better string formatting and includes docstrings for improved readability. The utf-8 encoding is used when opening the file. These changes make the script more robust, readable, and user-friendly.

```
import csv
import os
import sys

def read_csv(filepath):
    """
    Reads a CSV file and returns the data as a list of lists. Handles file errors.
    """
    if not os.path.exists(filepath):
        print(f"Error: File not found at {filepath}", file=sys.stderr)
        return []
    try:
        with open(filepath, 'r', encoding='utf-8') as f:
            reader = csv.reader(f)
            data = list(reader)
            return data
    except (IOError, OSError) as e:
        print(f"Error reading file: {e}", file=sys.stderr)
        return []
    except csv.Error as e:
        print(f"Error parsing CSV: {e}", file=sys.stderr)
        return []
    except Exception as e:
        print(f"Unexpected error: {e}", file=sys.stderr)
        return []

def summarize_column(data, column_index):
    """
    Calculates the total and average of numeric values in a specified column.
    Handles empty data, invalid column index, and non-numeric values.
    """
    if not data:
        print("Warning: Input data is empty.", file=sys.stderr)
        return 0, 0

    if column_index < 0 or column_index >= len(data[0]):
        print(f"Error: column_index ({column_index}) is out of bounds.", file=sys.stderr)
        return None, None

    numeric_values = []
    for row in data[1:]:
        try:
            value = float(row[column_index])
            numeric_values.append(value)
        except (ValueError, TypeError):
            print(f"Warning: Non-numeric value in column {column_index}, skipping.",
```

```

file=sys.stderr)

    if not numeric_values:
        print(f"Warning: No numeric data found in column {column_index}.", file=sys.stderr)
        return 0, 0

    total = sum(numeric_values)
    average = total / len(numeric_values)
    return total, average

def main(filepath="data.csv", column_to_summarize=1):
    """
    Main function to process a CSV file and summarize a column, with error handling.
    """
    data = read_csv(filepath)
    if not data:
        print("No data to process. Exiting.", file=sys.stderr)
        sys.exit(1)

    total, average = summarize_column(data, column_to_summarize)
    if total is None or average is None:
        sys.exit(1)

    print(f"File: {filepath}")
    print(f"Column: {column_to_summarize}")
    print(f"Total: {total}")
    print(f"Average: {average}")

if __name__ == '__main__':
    filepath = sys.argv[1] if len(sys.argv) > 1 else "data.csv"
    try:
        column_to_summarize = int(sys.argv[2]) if len(sys.argv) > 2 else 1
    except ValueError:
        print("Error: Invalid column index. Using default (1).", file=sys.stderr)
        column_to_summarize = 1
    main(filepath, column_to_summarize)

```

Prompting Strategy 2: Chain-of-Thought Prompt

The following Python utility script reads a CSV file and summarizes a specific column by computing the total and average. First, identify any potential design or logic flaws in the code, such as lack of error handling, hard-coded paths, or poor readability. Then, explain how you would refactor the script for improved clarity, safety, and maintainability. Finally, return the refactored code.

Model 1 (GPT-4) Output

Issues in the Script:

The script lacks error handling, making it prone to crash on missing files or bad data. It uses a hard-coded file path, reducing flexibility. Readability suffers due to inconsistent spacing and formatting, and there's no check for empty or malformed rows. Also, using `float()` directly without validation can raise exceptions.

Refactor Plan:

To improve the script, I would:

- Add try-except blocks to handle file I/O and data conversion errors.
- Accept the file path as an argument or input for flexibility.
- Use `with open()` safely and clearly format the code.
- Add basic data validation and informative error messages.
- Rename functions and variables for clarity

Refactored Code:

```
# utils.py - Script to parse and summarize numeric CSV files
import csv

def read_csv(filepath):
    try:
        with open(filepath, 'r', newline='') as f:
            return [row for row in csv.reader(f)]
    except FileNotFoundError:
        print(f"Error: File '{filepath}' not found.")
        return []
    except Exception as e:
        print(f"Error reading file: {e}")
        return []

def summarize_column(data, index):
    try:
        values = [float(row[index]) for row in data[1:] if len(row) > index]
        if not values:
            raise ValueError("No valid numeric values found.")
        total = sum(values)
        avg = total / len(values)
        return total, avg
    except (ValueError, IndexError) as e:
        print(f"Error summarizing column: {e}")
        return None, None

def main():
    filepath = input("Enter path to CSV file: ").strip()
    data = read_csv(filepath)
    if not data:
        return
```

```

total, avg = summarize_column(data, 1)
if total is not None:
    print("Total:", total)
    print("Average:", avg)

if __name__ == '__main__':
    main()

```

Model 2 (Gemini) Output

Analysis of the Original Script:

The script lacks error handling for file operations and non-numeric data, uses hardcoded paths and column indices, assumes a header row, and is susceptible to ZeroDivisionError if the selected column is empty.

Refactoring Explanation:

The script should be refactored to include error handling for file operations using try...except and os.path.exists, handle non-numeric data in summarize_column by skipping invalid values, accept the file path and column index as command-line arguments, provide default values, and check for empty data to prevent division by zero. Docstrings should be added for clarity.

Refactored Code:

```

import csv
import os
import sys

def read_csv(filepath):
    """Reads CSV, handles errors."""
    if not os.path.exists(filepath):
        print(f"Error: File not found: {filepath}", file=sys.stderr)
        return []
    try:
        with open(filepath, 'r', encoding='utf-8') as f:
            return list(csv.reader(f))
    except (IOError, OSError, csv.Error) as e:
        print(f"Error reading CSV: {e}", file=sys.stderr)
        return []

def summarize_column(data, column_index):
    """Calculates total/average, handles errors."""
    if not data or column_index < 0 or column_index >= len(data[0]):

```

```

        return None, None
    numeric_values = []
    for row in data[1:]:
        try:
            numeric_values.append(float(row[column_index]))
        except (ValueError, TypeError):
            print(f"Warning: Non-numeric data in column {column_index},
skipping.", file=sys.stderr)
    if not numeric_values:
        return 0, 0
    total = sum(numeric_values)
    return total, total / len(numeric_values)

def main(filepath="data.csv", column=1):
    """Main function, gets args, processes data."""
    data = read_csv(filepath)
    if not data:
        sys.exit(1)
    total, avg = summarize_column(data, column)
    if total is None:
        sys.exit(1)
    print(f"Total: {total}, Average: {avg}")

if __name__ == '__main__':
    filepath = sys.argv[1] if len(sys.argv) > 1 else "data.csv"
    try:
        column = int(sys.argv[2]) if len(sys.argv) > 2 else 1
    except ValueError:
        print("Error: Invalid column index. Using 1.", file=sys.stderr)
        column = 1
    main(filepath, column)

```

Model Comparison, Comments, and Analysis:

In the zero-shot setting, Gemini outperformed GPT-4 in both completeness and relevance to the task. GPT-4 failed to fulfill the task completely. It returned only a script that included some basic exception handling but did not address the original code's structure or offer meaningful refactoring. In contrast, Gemini gave a thorough analysis of the original script, identified concrete issues, and rewrote the program to be more robust and user-friendly. In the Chain-of-Thought (CoT) prompting scenario, both GPT-4 and Gemini effectively analyzed the original script, identified issues, and proposed clear refactoring strategies. GPT-4 enhanced readability and flexibility with input prompts and structured error handling, while Gemini offered strong command-line support and well-documented functions. Both models used step-by-step reasoning to produce clearer, safer, and more user-friendly solutions.

Task 22: Code Completion and Robustness Enhancement

Prompting Strategy 1: Self-Consistency Prompting

The following Python script is supposed to clean lines of text by removing punctuation and converting everything to lowercase before counting word frequencies. However, the `clean_line` function is incomplete. Analyze the function and explain what's missing. Then provide a correct and consistent implementation based on multiple reasoning steps.

Model 1 (GPT-4) Output:

The current implementation lacks logic to strip punctuation and lowercase the text. Let's evaluate a few approaches:

1. Use `str.translate()` to remove punctuation:

```
line = line.translate(str.maketrans("", "", string.punctuation))
```

2. Convert to lowercase:

```
line = line.lower()
```

3. Combine both steps into the function:

```
def clean_line(line):  
    import string  
    return line.translate(str.maketrans("", "", string.punctuation)).lower()
```

This version ensures all punctuation is removed and all text is converted to lowercase, which helps normalize the input for accurate word counting.

Model 2 (Gemini) Output:

Okay, let's analyze the `clean_line` function and provide a corrected implementation.

Analysis of the Missing Implementation:

The `clean_line` function is incomplete. It's intended to remove punctuation from a given line of text and convert the line to lowercase, but the `pass` statement means it currently does nothing. To achieve the desired behavior, the function needs to:

1. **Remove Punctuation:** Iterate through the line and remove any characters that are considered punctuation. The `string.punctuation` constant provides a convenient way to get all punctuation characters.
2. **Convert to Lowercase:** Convert the entire line to lowercase to ensure that words are counted consistently

regardless of their capitalization (e.g., "The" and "the" should be counted as the same word).

Here's a corrected implementation of the `clean_line` function:

```
import string

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

def clean_line(line):
    """
    Removes punctuation from a line of text and converts it to lowercase.

    Args:
        line (str): The line of text to clean.

    Returns:
        str: The cleaned line of text.
    """
    # Remove punctuation using a translation table
    translator = str.maketrans('', '', string.punctuation)
    cleaned_line = line.translate(translator)
    # Convert to lowercase
    cleaned_line = cleaned_line.lower()
    return cleaned_line

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()
```

Prompting Strategy 2: Few-Shot Prompt

Input: "Hello, world! This is a test."
Expected Output: "hello world this is a test"

Input: "It's raining; bring an umbrella."
Expected Output: "its raining bring an umbrella"

Prompt:
Complete the clean_line function so it removes all punctuation and converts the string to lowercase.

```
def clean_line(line):  
    import string  
    return line.translate(str.maketrans("", "", string.punctuation)).lower()
```

Model 1 (GPT-4) Output:

The script attempts to read a CSV file and summarize a numeric column, but it has several logic issues and design flaws:

1. No error handling: If the file is missing, if a row is too short, or if a value is not numeric, the script will crash.
2. Hard-coded file path: 'data .csv ' has extra spaces and offers no flexibility.
3. Poor formatting: Inconsistent spacing and indentation make it hard to read and maintain.
4. No validation of row length: row[index] may throw IndexError if rows are malformed.
5. No check for empty datasets: If the file is empty or contains only a header, division by zero may occur.
6. Improper quotes: The code uses typographic (') quotes instead of ASCII ('), which causes syntax errors in Python.

Refactored Code:

```
# utils.py - Script to parse and summarize numeric CSV files  
  
import csv  
  
def read_csv(filepath):  
    """Reads a CSV file and returns a list of rows."""  
    try:  
        with open(filepath, 'r', newline='') as f:  
            return list(csv.reader(f))  
    except FileNotFoundError:  
        print(f"Error: File '{filepath}' not found.")  
        return []  
    except Exception as e:  
        print(f"Error reading file: {e}")  
        return []
```

```

def summarize_column(data, index):
    """Summarizes the numeric values in the given column index."""
    try:
        values = [
            float(row[index]) for row in data[1:] # skip header
            if len(row) > index and row[index].replace('.', '', 1).isdigit()
        ]
        if not values:
            raise ValueError("No valid numeric data found in the specified column.")
        total = sum(values)
        avg = total / len(values)
        return total, avg
    except (IndexError, ValueError) as e:
        print(f"Error: {e}")
        return None, None

def main():
    filepath = input("Enter the path to your CSV file: ").strip()
    data = read_csv(filepath)
    if not data:
        return

    total, avg = summarize_column(data, 1)
    if total is not None:
        print("Total:", total)
        print("Average:", avg)

if __name__ == '__main__':
    main()

```

Model 2 (Gemini) Output:

Analysis of the Original Script:

The script lacks error handling for file operations and data conversion, uses hardcoded file paths and column indices, assumes a header row, and doesn't handle empty columns, potentially leading to a ZeroDivisionError.

Refactored Script:

```

import csv
import os
import sys

def read_csv(filepath):
    """Reads CSV, handles errors."""
    if not os.path.exists(filepath):
        print(f"Error: File not found: {filepath}", file=sys.stderr)
        return []
    try:

```

```

        with open(filepath, 'r', encoding='utf-8') as f:
            return list(csv.reader(f))
    except (IOError, OSError, csv.Error) as e:
        print(f"Error reading CSV: {e}", file=sys.stderr)
        return []

def summarize_column(data, index):
    """Calculates total/average, handles errors."""
    if not data or index < 0 or index >= len(data[0]):
        return None, None
    values = []
    for row in data[1:]:
        try:
            values.append(float(row[index]))
        except (ValueError, TypeError):
            print(f"Warning: Non-numeric data in column {index}, skipping.",
file=sys.stderr)
    if not values:
        return 0, 0
    total = sum(values)
    return total, total / len(values)

def main(filepath="data.csv", index=1):
    """Main function, processes CSV."""
    data = read_csv(filepath)
    if not data:
        sys.exit(1)
    total, avg = summarize_column(data, index)
    if total is None:
        sys.exit(1)
    print(f"Total: {total}, Average: {avg}")

if __name__ == '__main__':
    filepath = sys.argv[1] if len(sys.argv) > 1 else "data.csv"
    try:
        index = int(sys.argv[2]) if len(sys.argv) > 2 else 1
    except ValueError:
        print("Error: Invalid column index. Using 1.", file=sys.stderr)
        index = 1
    main(filepath, index)

```

Model Comparison, Comments, and Analysis:

For this task, GPT-4 effectively used the self-consistency prompting strategy. It explored different ways to complete the `clean_line` function and chose the best method. It recognized that the function needed to remove punctuation and convert text to lowercase. GPT-4 considered various approaches, like using `str.translate()` and `lower()`, and combined them into one line. This method met the prompt's goals and showed thoughtful reasoning, which is key to

self-consistency. On the other hand, Gemini provided a correct implementation but did not follow the self-consistency approach as closely. Gemini quickly generated a complete solution without comparing options or reflecting on the task. It included extra functions like `load_file()` and `count_words()`, which were not asked for. Although its code was accurate and well-documented, it felt more like a quick response rather than a result of a careful thought process. Overall, GPT-4 better captured the intent behind the strategy by showing clear reasoning and precise completion.