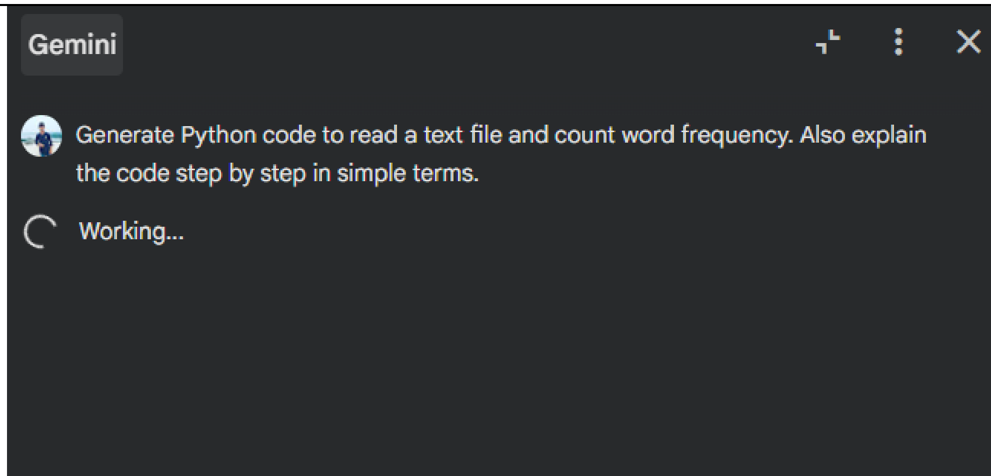
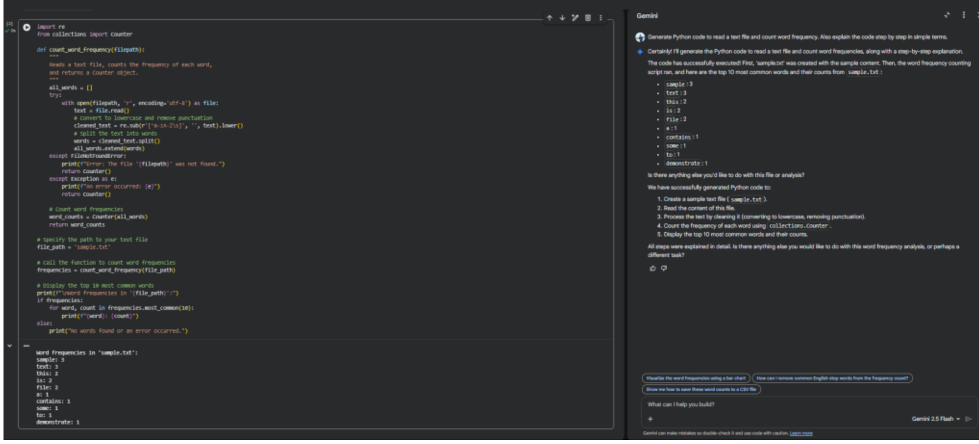
SCHOOL OF COMPUTER SCIENCE AND ARTIFICIAL INTELLIGENCE		DEPARTMENT OF COMPUTER SCIENCE ENGINEERING	
Program Name: B. Tech		**Assignment Type: Lab**	**Academic Year:**2025-2026
Course Coordinator Name		Dr. Rishabh Mittal	
Instructor(s) Name		Mr. S Naresh Kumar Ms. B. Swathi Dr. Sasanko Shekhar Gantayat Mr. Md Sallauddin Dr. Mathivanan Mr. Y Srikanth Ms. N Shilpa Dr. Rishabh Mittal (Coordinator) Dr. R. Prashant Kumar Mr. Ankushavali MD Mr. B Viswanath Ms. Sujitha Reddy Ms. A. Anitha Ms. M.Madhuri Ms. Katherashala Swetha Ms. Velpula sumalatha Mr. Bingi Raju	
CourseCode	23CS002PC304	**Course Title**	AI Assisted Coding
Year/Sem	III/II	**Regulation**	R23
Date and Day of Assignment	**Week1 – Wednesday**	**Time(s)**	23CSBTB01 To 23CSBTB52
Duration	2 Hours	**Applicable to Batches**	All batches
Assignment Number:1.3(Present assignment number)/**24**(Total number of assignments)			

Q.No.	**Question**	***Expected Time to com***

		plet e
1	Lab 2: Exploring Additional AI Coding Tools beyond Copilot – Gemini (Colab) and Cursor AI **Lab Objectives:** ❖ To explore and evaluate the functionality of Google Gemini for AI-assisted coding within Google Colab. ❖ To understand and use Cursor AI for code generation, explanation, and refactoring. ❖ To compare outputs and usability between Gemini, GitHub Copilot, and Cursor AI. ❖ To perform code optimization and documentation using AI tools. **Lab Outcomes (LOs):** After completing this lab, students will be able to: ❖ Generate Python code using Google Gemini in Google Colab. ❖ Analyze the effectiveness of code explanations and suggestions by Gemini. ❖ Set up and use Cursor AI for AI-powered coding assistance. ❖ Evaluate and refactor code using Cursor AI features. ❖ Compare AI tool behavior and code quality across different platforms.	Week1 - Monday
	Task 1: Word Frequency from Text File ❖ **Scenario:** You are analyzing log files for keyword frequency. ❖ **Task:** Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code. ❖ **Expected Output:** ➢ Working code ➢ Explanation ➢ Screenshot Prompt Generate Python code to read a text file and count word frequency. Also explain the code step by step in simple terms.	





Description

In this task, Google Gemini is used within Google Colab to generate Python code that reads a text file and calculates the frequency of each word. This task simulates real-world log file analysis where keyword frequency is important. Gemini not only generates correct Python code but also provides a clear explanation of each step, helping users understand how file handling and text processing work in Python.

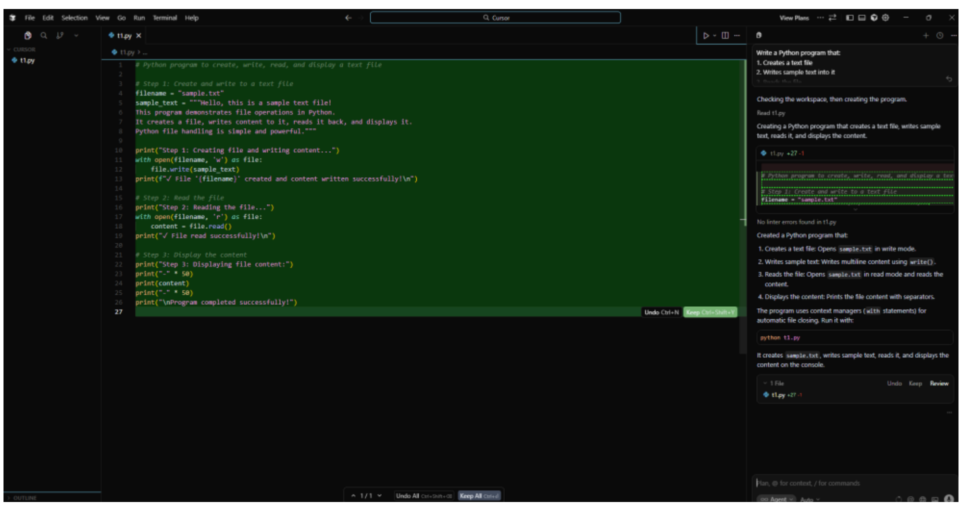
Task 2: File Operations Using Cursor AI

❖ **Scenario:**
You are automating basic file operations.

❖ **Task:**
Use Cursor AI to generate a program that:
- ➢ Creates a text file
- ➢ Writes sample text
- ➢ Reads and displays the content

❖ **Expected Output:**
- ➢ Functional code
- ➢ Cursor AI screenshots

Prompt
Write a Python program that:
1. Creates a text file
2. Writes sample text into it
3. Reads the file
4. Displays the content



Explanation

- File opened in write mode
- Sample text written
- File reopened in read mode
- Content printed

Description

This task demonstrates the use of Cursor AI to automate basic file operations in Python. Cursor AI is used to generate a program that creates a text file, writes sample content into it, reads the stored data, and displays the output. This task highlights how Cursor AI assists in quickly generating functional code and is useful for understanding file handling concepts in Python.

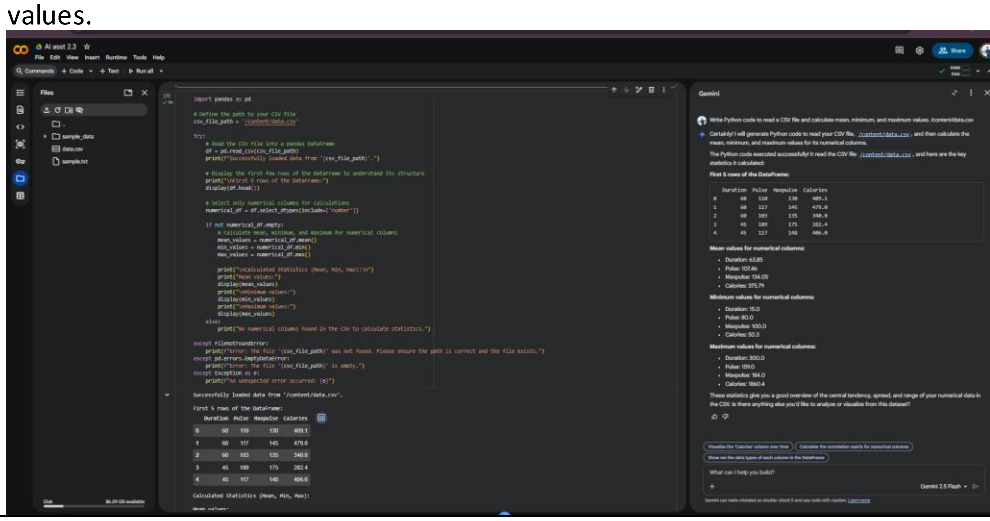
Task 3: CSV Data Analysis

❖ **Scenario:**
You are processing structured data from a CSV file.

❖ **Task:**
Use Gemini in Colab to read a CSV file and calculate mean, min, and max.

❖ **Expected Output:**
- ➢ Correct output
- ➢ Screenshot

Prompt
Write Python code to read a CSV file and calculate mean, minimum, and maximum values.



Duration	63.846154
Pulse	107.461538
Maxpulse	134.047337
Calories	375.790244

dtype: float64

```
Minimum values:
               0
```

	0
Duration	15.0
Pulse	80.0
Maxpulse	100.0
Calories	50.3

dtype: float64

```
Maximum values:
               0
```

	0
Duration	300.0
Pulse	159.0
Maxpulse	184.0
Calories	1860.4

dtype: float64

Code Explanation:

1. `import pandas as pd` : Imports the pandas library, which is essential for working with DataFrames (table-like data structures).
2. `csv_file_path = '/content/data.csv'` : Defines the path to your CSV file. Make sure this path is correct.
3. `try...except` **block:** This block is used for robust error handling:
   - `df = pd.read_csv(csv_file_path)` : Attempts to read the CSV file into a pandas DataFrame named `df`.
   - `display(df.head())` : Shows the first 5 rows of the loaded DataFrame. This helps verify that the data was loaded correctly and gives an idea of its structure.
   - `numerical_df = df.select_dtypes(include=['number'])` : Filters the DataFrame to include only columns with numerical data types (integers and floats). Statistical calculations only make sense for numerical columns.
   - `mean_values = numerical_df.mean()` : Calculates the mean (average) for each numerical column.
   - `min_values = numerical_df.min()` : Calculates the minimum value for each numerical column.
   - `max_values = numerical_df.max()` : Calculates the maximum value for each numerical column.
   - `display(...)` : Prints the calculated mean, minimum, and maximum values for each respective numerical column.
   - **Error Handling:** Catches `FileNotFoundError` if the CSV doesn't exist, `EmptyDataError` if the file is empty, and a general `Exception` for any other issues, providing informative messages.

Explanation

- Pandas used for CSV handling
- mean(), min(), max() functions applied
- Displays statistical values

Description

In this task, Google Gemini is used in Google Colab to process structured data from a CSV file. The generated Python code reads the CSV file and calculates statistical values such as mean, minimum, and maximum. This task demonstrates how AI tools simplify data analysis tasks and help users apply Python libraries like Pandas for real-world data processing.

Task 4: Sorting Lists – Manual vs Built-in

❖ **Scenario:**
You are reviewing algorithm choices for efficiency.

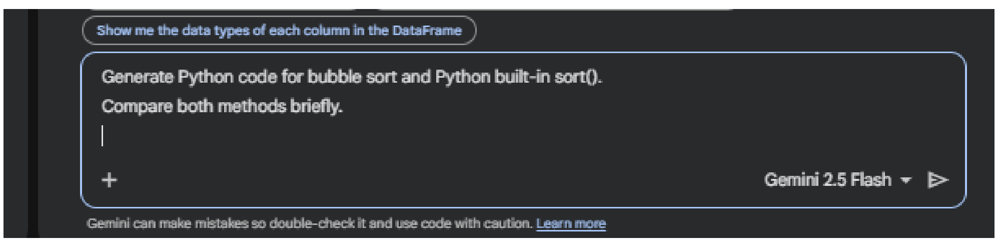
❖ **Task:**
Use **Gemini** to generate:

- Bubble sort
- Python's built-in sort()
- Compare both implementations.

❖ **Expected Output:**

- Two versions of code
- Short comparison

Note: Report should be submitted as a word document for all tasks in a single document with prompts, comments & code explanation, and output and if required, screenshots.



Prompt
Generate Python code for bubble sort and Python built-in sort().
Compare both methods briefly.

First, let's implement the **Bubble Sort** algorithm. It's a simple comparison-based sorting algorithm.

```
def bubble_sort(arr):
    n = len(arr)
    # Traverse through all array elements
    for i in range(n - 1):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Traverse the array from 0 to n-i-1
            # Swap if the element found is greater
            # than the next element
            if arr[j] > arr[j + 1]:
                arr[j], arr[j + 1] = arr[j + 1], arr[j]
    return arr

# Test Bubble Sort
my_list = [64, 34, 25, 12, 22, 11, 90]
print(f"Original list: {my_list}")
sorted_list_bubble = bubble_sort(list(my_list)) # Use a copy to avoid modifying original
print(f"Sorted using Bubble Sort: {sorted_list_bubble}")
```

Next, let's use Python's **built-in `sort()` method** (for lists) or `sorted()` function (for any iterable). These are highly optimized and generally preferred in practice.

```
# Test Python's built-in sort()
my_list_2 = [64, 34, 25, 12, 22, 11, 90]
print(f"Original list: {my_list_2}")
my_list_2.sort() # Sorts the list in-place
print(f"Sorted using built-in sort() method: {my_list_2}")

# Test Python's built-in sorted() function
my_tuple = (64, 34, 25, 12, 22, 11, 90)
print(f"Original tuple: {my_tuple}")
sorted_tuple = sorted(my_tuple) # Returns a new sorted list from an iterable
print(f"Sorted using built-in sorted() function: {sorted_tuple}")
```

✦ Gemini

Comparison of Bubble Sort vs. Python's Built-in Sort

Feature	Bubble Sort	Python's Built-in `sort()` / `sorted()`
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and i
Time Complexity	**Worst/Average:** O(n^2) (very inefficient for large datasets). **Best:** O(n) (if already sorted).	**Worst/Average/Best:** O(n log n) (very efficient for large datasets).
Space Complexity	O(1) (in-place sort).	O(n) in worst case (for Timsort, due to temporary storage).
Stability	Stable (maintains the relative order of equal elements).	Stable.
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its

In summary: While Bubble Sort is easy to understand, Python's built-in `sort()` method (and `sorted()` function) are significantly more efficient, robust, and should be used for practical applications.

Sure! I'll generate the Python code for both bubble sort and the built-in `sort()` function, and then provide a brief comparison of these two sorting methods.

Great! Both the Bubble Sort implementation and the demonstrations of Python's built-in `sort()` method and `sorted()` function have executed successfully.

Bubble Sort:

- Original list: `[64, 34, 25, 12, 22, 11, 90]`
- Sorted using Bubble Sort: `[11, 12, 22, 25, 34, 64, 90]`

Python's Built-in Sort:

- Original list (`my_list_2`): `[64, 34, 25, 12, 22, 11, 90]`
- Sorted using `sort()` method: `[11, 12, 22, 25, 34, 64, 90]`
- Original tuple (`my_tuple`): `(64, 34, 25, 12, 22, 11, 90)`
- Sorted using `sorted()` function: `[11, 12, 22, 25, 34, 64, 90]`

As you can see, both methods correctly sorted the provided lists/tuples. Now, let's briefly compare them:

Comparison of Bubble Sort vs. Python's Built-in Sort

Feature	Bubble Sort	Python's Built-in `sort()` / `sorted()`
Algorithm	Simple comparison-based, repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.	Timsort (a hybrid stable sorting algorithm, derived from merge sort and insertion sort). Highly optimized.
Time Complexity	**Worst/Average:** O(n^2) (very inefficient for large datasets). **Best:** O(n) (if already sorted).	**Worst/Average/Best:** O(n log n) (very efficient for large datasets).
Space Complexity	O(1) (in-place sort).	O(n) in worst case (for Timsort, due to temporary storage).
Stability	Stable (maintains the relative order of equal elements).	Stable.
Ease of Use	Requires manual implementation.	Very easy to use; built directly into Python.
When to Use	Primarily for educational purposes to understand basic sorting concepts. Not recommended for production.	Almost always preferred for general-purpose sorting in Python due to its efficiency and reliability.

In summary: While Bubble Sort is easy to understand, Python's built-in `sort()` method (and `sorted()` function) are significantly more efficient, robust, and should be used for practical applications.

We've successfully generated Python code for both the Bubble Sort algorithm and Python's highly optimized built-in `sort()` method and `sorted()` function. We also executed them to demonstrate their functionality, and provided a detailed comparison highlighting their differences in terms of algorithm, time/space complexity, stability, and ease of use.

Is there anything else I can assist you with regarding sorting, or perhaps a different programming task?

Final Comparison Section

Tool	Strengths	Weakness
Gemini	Good explanations, works well in Colab	Needs internet

Cursor AI	Fast code generation & refactoring	Editor based
GitHub Copilot	Inline suggestions	Limited explanation

Description

This task compares two sorting approaches generated by Google Gemini: a manual sorting algorithm (Bubble Sort) and Python's built-in `sort()` method. The purpose of this task is to analyze the efficiency and readability of different sorting techniques. Gemini helps in generating both implementations and provides a brief comparison, showing why built-in methods are more efficient for practical applications.