AIAC – 2.3

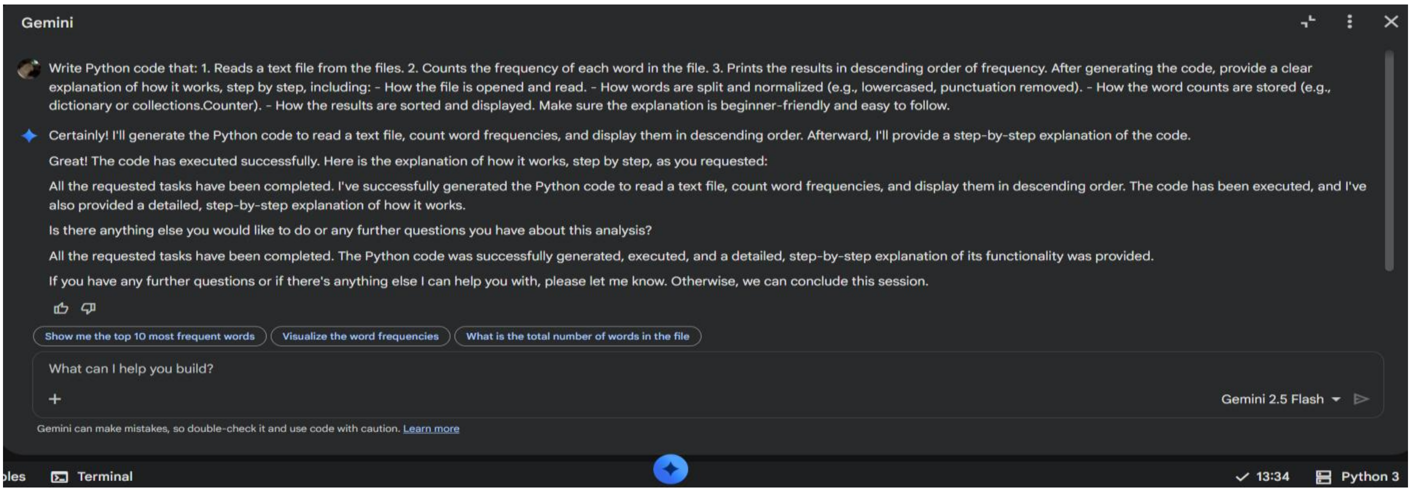
2303A52065

Task 1: Word Frequency from Text File

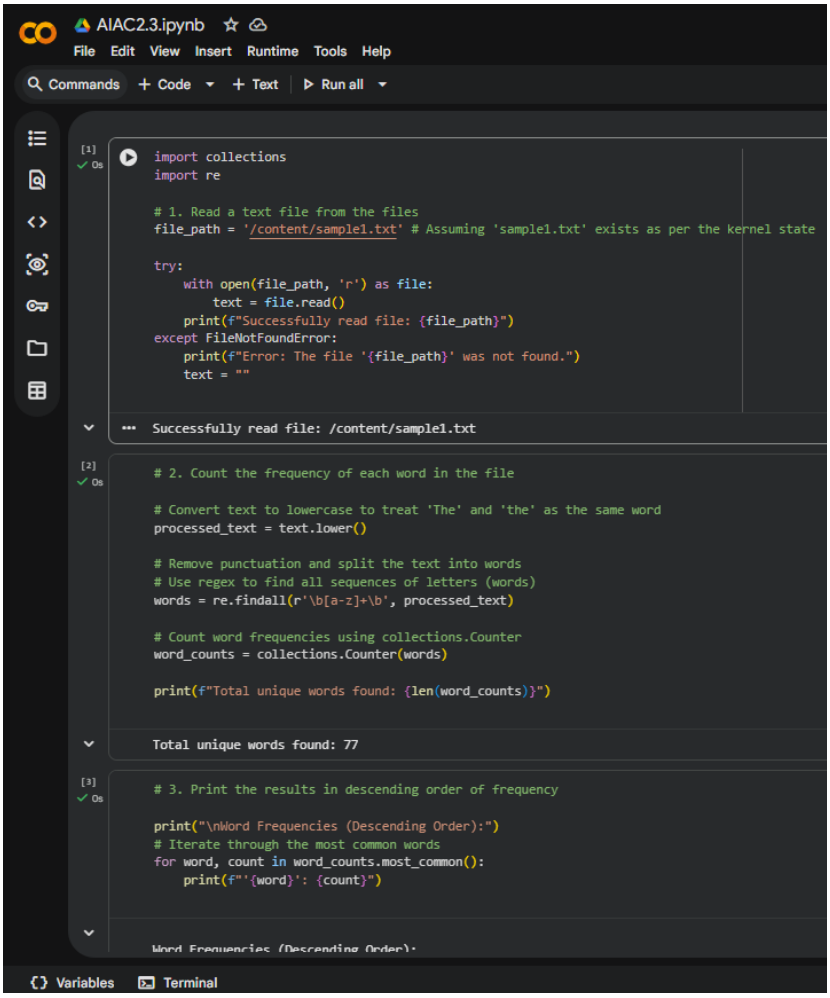
❖ Scenario: You are analyzing log files for keyword frequency.

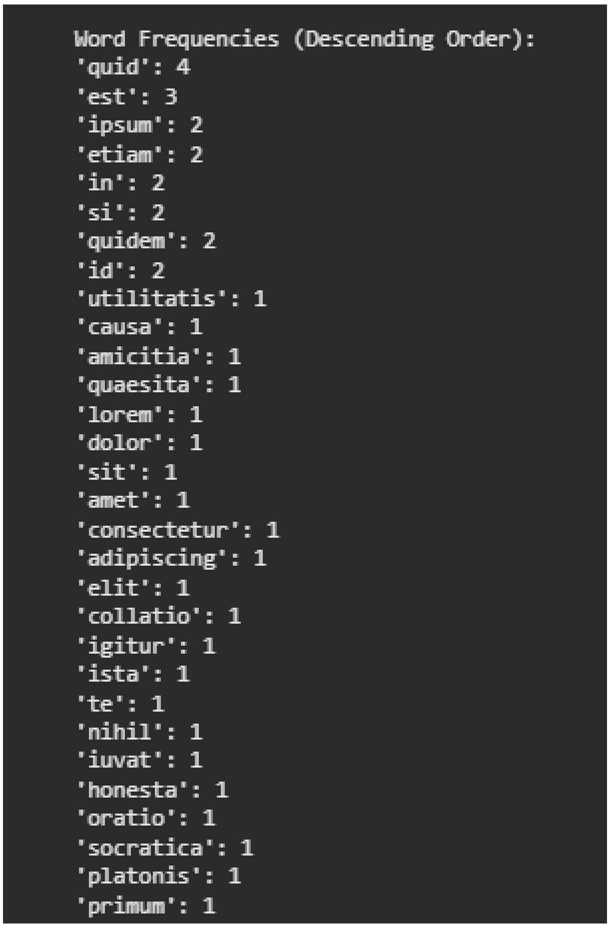
❖ Task: Use Gemini to generate Python code that reads a text file and counts word frequency, then explains the code.

Prompt: Google Colab Gemini Response



Generated Code Output:





Explanation of the Code

This code performs three main tasks: reading a file, counting word frequencies, and displaying them in descending order.

Step 1: Reading the Text File

```
import collections
import re

file_path = '/content/sample1.txt'

try:
    with open(file_path, 'r') as file:
        text = file.read()
    print(f"Successfully read file: {file_path}")
except FileNotFoundError:
    print(f"Error: The file '{file_path}' was not found.")
    text = ""
```

1. **`import collections` and `import re`**: These lines import necessary modules. `collections` provides the `Counter` class for efficient counting, and `re` (regular expressions) helps with cleaning and splitting the text.
2. **`file_path = '/content/sample1.txt'`**: This line defines the path to the text file we want to analyze. It's set to `'/content/sample1.txt'` based on the available files in your environment.
3. **`try...except FileNotFoundError`**: This block is used for error handling. It attempts to open and read the file. If the file doesn't exist at the specified `file_path`, it will catch the `FileNotFoundError` and print an error message instead of crashing the program.
4. **`with open(file_path, 'r') as file:`**: This is the standard and recommended way to open files in Python.
    - `open(file_path, 'r')` opens the file specified by `file_path` in read mode (`'r'`).
    - `with` ensures that the file is automatically closed once the block is exited, even if errors occur.
    - `as file` assigns the opened file object to the variable `file`.
5. **`text = file.read()`**: This line reads the entire content of the opened file and stores it as a single string in the `text` variable.

Step 2: Counting Word Frequencies

```
processed_text = text.lower()
words = re.findall(r'\b[a-z]+\b', processed_text)
word_counts = collections.Counter(words)
```

CO AIAC2.3.ipynb

File Edit View Insert Runtime Tools Help

Commands + Code + Text Run all

Step 2: Counting Word Frequencies

```
processed_text = text.lower()
words = re.findall(r'\b[a-z]+\b', processed_text)
word_counts = collections.Counter(words)

print(f"Total unique words found: {len(word_counts)}")
```

1. **`processed_text = text.lower()`**: This line converts all characters in the `text` string to lowercase. This step is crucial for **normalizing** words, ensuring that, for example, "The" and "the" are counted as the same word.
2. **`words = re.findall(r'\b[a-z]+\b', processed_text)`**: This is where the text is **split and cleaned.**
    - `re.findall()` is a regular expression function that finds all non-overlapping matches of a pattern in a string and returns them as a list.
    - The pattern `r'\b[a-z]+\b'` looks for sequences of one or more lowercase letters (`[a-z]+`) that are surrounded by word boundaries (`\b`). This effectively extracts words, ignoring numbers, punctuation, and spaces.
3. **`word_counts = collections.Counter(words)`**: This is a very efficient way to **store word counts.**
    - `collections.Counter` is a specialized dictionary subclass that is used for counting hashable objects. When you pass a list (like `words`) to `Counter`, it automatically iterates through the list and counts the occurrences of each unique item.
    - The `word_counts` variable will now be a `Counter` object where keys are words and values are their frequencies (e.g., `{'hello': 2, 'world': 1}`).

Step 3: Printing Results

```
print("\nWord Frequencies (Descending Order):")
for word, count in word_counts.most_common():
    print(f"'{word}': {count}")
```

1. **`word_counts.most_common()`**: This is a method of the `Counter` object that returns a list of the *n* most common elements and their counts, from the most common to the least. If no argument is given (as in this case), it returns all unique elements in descending order of their frequency.
2. **`for word, count in ...`**: This loop iterates through the list returned by `most_common()`. In each iteration, `word` will hold the word and `count` will hold its frequency.
3. **`print(f"'{word}': {count}")`**: This line prints each word and its corresponding count in a clear format. The `f-string` (formatted string literal) makes it easy to embed variables directly into the string.

{} Variables Terminal

Justification:

Gemini generated correct python code using built-in libraries and gave a clear understandable explanation as well.

It converted all text to lowercase to avoid case mismatch and stored counter that counts each word automatically, reduced manual effort in writing and debugging code.

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

Prompt:

```
Generate a Python program that performs the following file operations:

1. Creates a new text file named "sample.txt".
2. Writes the line "Hello, this is a sample text file." into the file.
3. Reads the content of "sample.txt".
4. Displays the content on the console.

After generating the code, explain step by step how the program works, including:
- How the file is created and opened.
- How text is written into the file.
- How the file is read.
- How the output is displayed.

Agent    Auto
```

Code & Output:

```
Checking the current directory, then creating the Python program.
List model provided non-existent directory
Creating the Python program:
Edit unable to write file '\file_operations.py' (NoPermissions (FileSystemError): Error: EPERM: operation not permitted, open 'C:\file_operations.py')
Checking the current working directory:

Auto-Ran command: pwd
$ pwd

Trying a different approach. Checking the current directory on Windows:

Auto-Ran command: cd
$ cd

Creating the file in a Python-friendly location using Python directly:

Auto-Ran command: python
$ python -c "with open('file_operations.py', 'w') as f: f.write('# Python program to demonstrate file operations\n\n# Step 1: Create and open a new
  text file for writing\nfile = open(\"sample.txt\", \"w\")\n\n# Step 2: Write text into the file\nfile.write(\"Hello, this is a sample text file.\")
  \n\n# Step 3: Close the file after writing\nfile.close()\n\n# Step 4: Open the file again for reading\nfile = open(\"sample.txt\", \"r\")\n\n# Step
  5: Read the content of the file\ncontent = file.read()\n\n# Step 6: Close the file after reading\nfile.close()\n\n# Step 7: Display the content on
  the console\nprint(\"File content:\")\nprint(content)\n')"
```

```
SyntaxError: unterminated string literal (detected at line 1)
```

Creating the file using PowerShell:

Auto-Ran command: @' # Python program to demonstrate file operations # Step 1: Create and open a new text file for writing file = open("sample.txt", "w") # Step 2: Write text into the file file.write("Hello, this is a sample text file.") # Step 3: Close the file after writing file.close() # Step 4: Open the file again for reading file = open("sample.txt", "r") # Step 5: Read the content of the file content = file.read() # Step 6: Close the file after reading file.close() # Step 7: Display the content on the console print("File content:") print(content) '@, Out-File

```
$ @'
  # Python program to demonstrate file operations

  # Step 1: Create and open a new text file for writing
  file = open("sample.txt", "w")

  # Step 2: Write text into the file
  file.write("Hello, this is a sample text file.")

  # Step 3: Close the file after writing
  file.close()

  # Step 4: Open the file again for reading
  file = open("sample.txt", "r")

  # Step 5: Read the content of the file
  content = file.read()

  # Step 6: Close the file after reading
  file.close()
```

```
  # Step 7: Display the content on the console
  print("File content:")
  print(content)
  '@ | Out-File -FilePath file_operations.py -Encoding utf8
```

Verifying the file was created and improving it to use Python best practices (context managers):

Read could not find file 'file_operations.py' in the workspace.

Checking if the file exists and running it:

Auto-Ran command: python

```
$ python file_operations.py
File content:
Hello, this is a sample text file.
```

The program works. Creating an improved version using context managers (the `with` statement), which is the recommended Python approach for file handling:

Auto-Ran command: Get-ChildItem, Select-Object

```
$ Get-ChildItem -Filter file_operations.py | Select-Object FullName
```

Creating an improved version using Python best practices (context managers):

Auto-Ran command: @' # Python program to demonstrate file operations (Improved version using context managers) # Step 1: Create and open a new text file for writing using 'with' statement # The 'with' statement automatically closes the file when done with open("sample.txt", "w") as file: # Step 2: Write text into the file file.write("Hello, this is a sample text file.") # Step 3: File is automatically closed when exiting the 'with' block # Step 4: Open the file again for reading using 'with' statement with open("sample.txt", "r") as file: # Step 5: Read the content of the file content = file.read() # Step 6: File is automatically closed when exiting the 'with' block # Step 7: Display the content on the console print("File content:") print(content) '@, Out-File

Review

console print("File content:") print(content) '@, Out-File

```
$ @'
  # Python program to demonstrate file operations (Improved version using context managers)

  # Step 1: Create and open a new text file for writing using 'with' statement
  # The 'with' statement automatically closes the file when done
  with open("sample.txt", "w") as file:
      # Step 2: Write text into the file
      file.write("Hello, this is a sample text file.")

  # Step 3: File is automatically closed when exiting the 'with' block

  # Step 4: Open the file again for reading using 'with' statement
  with open("sample.txt", "r") as file:
      # Step 5: Read the content of the file
      content = file.read()

  # Step 6: File is automatically closed when exiting the 'with' block

  # Step 7: Display the content on the console
  print("File content:")
  print(content)
  '@ | Out-File -FilePath file_operations_improved.py -Encoding utf8
```

Verifying the improved version works:

Auto-Ran command: python

Auto-Ran command: python

```
$ python file_operations_improved.py
File content:
Hello, this is a sample text file.
```

Run Everything

The program works. Here's a step-by-step explanation:

How the Program Works

Step 1: Creating and Opening the File for Writing

```
file = open("sample.txt", "w")
```

- `open()` creates or opens a file.
- First argument: `"sample.txt"` is the filename.
- Second argument: `"w"` opens the file in write mode (overwrites if it exists).
- The function returns a file object stored in `file`.

Step 2: Writing Text into the File

```
file.write("Hello, this is a sample text file.")
```

- `write()` writes the string to the file.
- The text is added to the file buffer.

Step 3: Closing the File After Writing

```
file.close()
```

- `close()` flushes buffers and releases the file handle.
- Important to avoid data loss and handle leaks.

Step 4: Opening the File for Reading

```
file = open("sample.txt", "r")
```

- `"r"` opens the file in read mode (default; can be omitted).
- If the file doesn't exist, this raises an error.

Step 5: Reading the Content

```
content = file.read()
```

- `read()` reads the entire file content into `content` as a string.
- Alternative methods: `readline()` for one line, `readlines()` for a list of lines.

Step 6: Closing the File After Reading

```
file.close()
```

- Closes the file to free resources.

Step 7: Displaying the Content

```
print("File content:")
print(content)
```

- `print()` outputs to the console.

Improved Version (Recommended Practice)

I've also created `file_operations_improved.py` which uses the `with` statement (context manager). This automatically closes files even if an error occurs:

```
with open("sample.txt", "w") as file:
    file.write("Hello, this is a sample text file.")
# File is automatically closed here

with open("sample.txt", "r") as file:
    content = file.read()
# File is automatically closed here

print("File content:")
print(content)
```

Both programs produce the same output. The improved version is more Pythonic and safer.

Both files have been created and tested successfully. The program creates `sample.txt`, writes text to it, reads it back, and displays it on the console.

Justification:

Cursor generated correct python program, did all the work from beginning to end, even added an improved version type of code for recommended practice. The code was easy to understand, creating, writing and reading the files were done well which reduced the manual effort and chances of syntax errors.

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.

Prompt:

Gemini

from the files, using the csv data file, do a data analysis like calculating min, max and mean, analyze and visualize the file and give explanation of the code output

Great! I've put together a plan to analyze your CSV data, calculate key statistics, and visualize the results. Let me know if this plan works for you!

- **Load CSV Data**: Load the data from the provided CSV file into a pandas DataFrame.
- **Calculate Descriptive Statistics**: Calculate the minimum, maximum, and mean for relevant numerical columns in the dataset. This will provide an initial understanding of the data distribution.
- **Visualize Data**: Generate appropriate visualizations (e.g., histograms, bar charts, or count plots) to analyze the distribution of the data. Legends will be provided for clarity.
- **Explain Code Output**: Provide a clear explanation of the generated code for data loading, statistical calculations, and visualization, along with interpretations of their outputs.
- **Final Task**: Summarize the findings from the data analysis, including the calculated statistics and insights from the visualizations.

Generating...

Preparing

Load CSV Data

Calculate Descriptive Statistics

Convert 'Duration' to Numerical and Recalculate Statistics

Explain the code for calculating descriptive statistics | Explain the visualizations generated for the data | Show the final summary of the analysis findings

Code & Output:

Task

Load the data from the `duration.csv` file into a pandas DataFrame, calculate descriptive statistics (min, max, mean) for relevant numerical columns, visualize the data distribution, explain the code and its output, and summarize the findings from the analysis.

Load CSV Data

Subtask:

Load the data from the provided CSV file into a pandas DataFrame.

Reasoning: First, I'll import the pandas library, then load the 'duration.csv' file into a DataFrame named 'df_duration', and finally display the first 5 rows and the DataFrame's information to inspect the data.

```
import pandas as pd

# Load the duration.csv file into a pandas DataFrame
df_duration = pd.read_csv('/content/duration.csv')

# Display the first 5 rows of the DataFrame
print("First 5 rows of the DataFrame:")
print(df_duration.head())

# Display information about the DataFrame
print("\nDataFrame Information:")
df_duration.info()
```

```
First 5 rows of the DataFrame:
            Duration
0  Less than a month
1         1-6 months
2        6-12 months
3          1-3 years
4       Over 3 years

DataFrame Information:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 6 entries, 0 to 5
Data columns (total 1 columns):
 #   Column    Non-Null Count  Dtype
---  ------    --------------  -----
```

```
 ---  ------    --------------  -----
  0   Duration  6 non-null      object
dtypes: object(1)
memory usage: 180.0+ bytes
```

Calculate Descriptive Statistics

Subtask:

Calculate the minimum, maximum, and mean for relevant numerical columns in the dataset.

Reasoning: The subtask requires calculating descriptive statistics. I will call the `.describe()` method on the `df_duration` DataFrame to show its output for numerical columns. Since 'Duration' is an object type, it is expected that direct numerical statistics like min, max, and mean will not be generated for this column.

```
print("Descriptive statistics for df_duration:")
df_duration.describe()
```

Descriptive statistics for df_duration:

	Duration
count	6
unique	6
top	Less than a month
freq	1

Explanation of Descriptive Statistics

The `df_duration.describe()` method was called to generate descriptive statistics for the DataFrame. The output shows:

```
                 Duration
count                   6
unique                  6
top     Less than a month
freq                    1
```

As observed from the `df_duration.info()` output in the previous step, the `Duration` column is of `object` (string) type

As observed from the `df_duration.info()` output in the previous step, the `Duration` column is of `object` (string) type. Consequently, `describe()` provides statistics relevant to categorical/object data (like `count`, `unique` values, the `top` occurring value, and its `freq` uency) rather than numerical statistics (like `min`, `max`, `mean`, `std`, `25%`, `50%`, `75%`).

Conclusion: Direct numerical descriptive statistics (minimum, maximum, and mean) are not applicable to the `Duration` column because it contains categorical string values and not numerical data.

Convert 'Duration' to Numerical and Recalculate Statistics

Subtask:

Convert the categorical 'Duration' column into a numerical representation (e.g., in months) and then recalculate descriptive statistics (min, max, mean) for this new numerical column.

Reasoning: I will convert the categorical 'Duration' column to a numerical representation in months using a defined mapping, then calculate and display the minimum, maximum, and mean of this new numerical column.

```
import numpy as np

# 1. Create a dictionary to map categorical duration values to numerical months
duration_mapping = {
    'Less than a month': 0.5,
    '1-6 months': 3.5, # Average of 1 and 6 months
    '6-12 months': 9,  # Average of 6 and 12 months
    '1-3 years': 24,   # Average of 12 (1 year) and 36 (3 years) months
    'Over 3 years': 48, # Representing 4 years as an approximation
    'Never': 0
}

# 2. Create a new column 'Duration_Months' by applying the mapping
df_duration['Duration_Months'] = df_duration['Duration'].map(duration_mapping)

# 3. Calculate and print the minimum, maximum, and mean of the new numerical column
min_duration = df_duration['Duration_Months'].min()
max_duration = df_duration['Duration_Months'].max()
mean_duration = df_duration['Duration_Months'].mean()

print(f"\nMinimum Duration (Months): {min_duration}")
print(f"Maximum Duration (Months): {max_duration}")
print(f"Mean Duration (Months): {mean_duration:.2f}")
```

```
print(f"\nMinimum Duration (Months): {min_duration}")
print(f"Maximum Duration (Months): {max_duration}")
print(f"Mean Duration (Months): {mean_duration:.2f}")

print("\nDataFrame with new numerical column:")
print(df_duration.head())
```

```
Minimum Duration (Months): 0.0
Maximum Duration (Months): 48.0
Mean Duration (Months): 14.17

DataFrame with new numerical column:
         Duration  Duration_Months
0  Less than a month             0.5
1        1-6 months             3.5
2       6-12 months             9.0
3         1-3 years            24.0
4       Over 3 years           48.0
```

Explanation of 'Duration' Column Conversion and Statistics

In this step, the categorical `Duration` column was converted into a numerical representation (`Duration_Months`) to enable calculation of descriptive statistics. This was done using a mapping dictionary and the `.map()` function.

1. **`duration_mapping` Dictionary:** A dictionary was created to define how each categorical string in the `Duration` column should be translated into a numerical value (representing months). For example:
   - `'Less than a month'` was mapped to `0.5` (representing half a month).
   - `'1-6 months'` was mapped to `3.5` (the average of 1 and 6).
   - `'6-12 months'` was mapped to `9` (the average of 6 and 12).
   - `'1-3 years'` was mapped to `24` (the average of 12 months for 1 year and 36 months for 3 years).
   - `'Over 3 years'` was mapped to `48` (an approximation of 4 years in months).
   - `'Never'` was mapped to `0`.
2. **`df_duration['Duration_Months'] = df_duration['Duration'].map(duration_mapping)`**: A new column named `Duration_Months` was added to the `df_duration` DataFrame. The `.map()` method applied the `duration_mapping` dictionary to each value in the `Duration` column, effectively transforming the categorical strings into their corresponding numerical month values.
3. **Calculating Statistics:** After the numerical conversion, the `min()`, `max()`, and `mean()` methods were applied directly to the new `Duration_Months` column to obtain its descriptive statistics.

Output Interpretation

The output displays the calculated statistics:

- **Minimum Duration (Months):** `0.0`: This indicates that the shortest duration, corresponding to 'Never', is 0 months.
- **Maximum Duration (Months):** `48.0`: This indicates that the longest duration, corresponding to 'Over 3 years', is approximated as 48 months.
- **Mean Duration (Months):** `14.17`: This is the average duration across all categories, calculated as approximately 14.17 months.

The `DataFrame with new numerical column` printout confirms that the `Duration_Months` column has been successfully added to the DataFrame, showing the numerical representation alongside the original categorical values.

Visualize Data

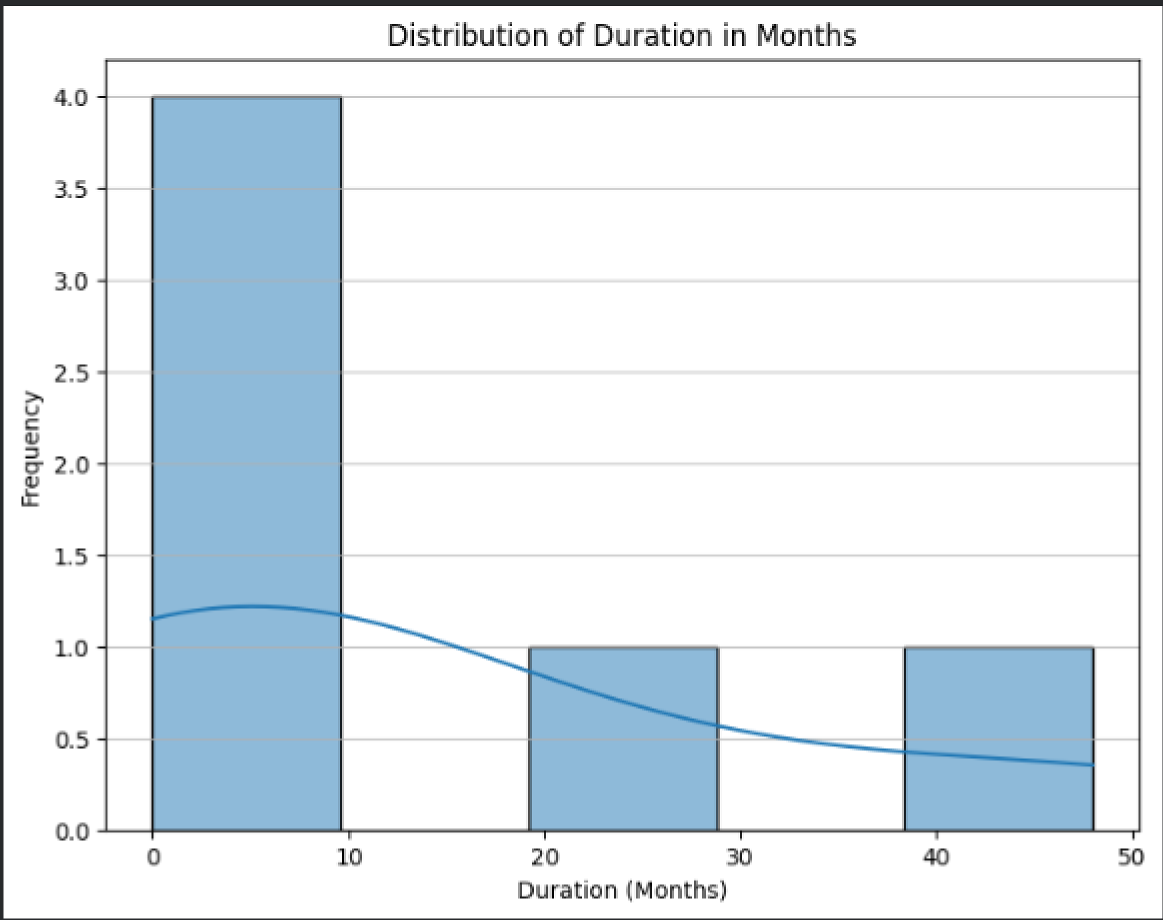
Subtask:

Generate appropriate visualizations (e.g., histograms, bar charts, or count plots) to analyze the distribution of the data. Legends will be provided for clarity.

Reasoning: To visualize the distribution of the 'Duration_Months' column, I will generate a histogram using Seaborn and Matplotlib as instructed. This will help in understanding the spread and frequency of duration values.

```python
import matplotlib.pyplot as plt
import seaborn as sns

# Create a histogram to visualize the distribution of 'Duration_Months'
plt.figure(figsize=(8, 6))
sns.histplot(df_duration['Duration_Months'], kde=True, bins=5)
plt.title('Distribution of Duration in Months')
plt.xlabel('Duration (Months)')
plt.ylabel('Frequency')
plt.grid(axis='y', alpha=0.75)
plt.show()
```



Justification:

Gemini made it easy to analyze csv data by generating clear structured python code. It itself correctly identified that the data was categorical and explained why direct numerical statistics was not applicable and converted and handled logically. Overall, it did its given task and reduced manual effort and improved understanding of real-world data preprocessing and analysis.

Summary:

Data Analysis Key Findings

- The initial `duration.csv` dataset contained a single column, `Duration`, which was of `object` (string) type, indicating categorical data. It had 6 entries with no missing values.
- Direct calculation of numerical descriptive statistics (min, max, mean) was not possible on the `Duration` column due to its categorical nature. The `describe()` method yielded categorical statistics like `count`, `unique`, `top` value, and `frequency`.
- The `Duration` column was successfully converted into a numerical representation, `Duration_Months`, by mapping categorical ranges to approximate average monthly values. For instance, 'Less than a month' was mapped to 0.5 months, '1-6 months' to 3.5 months, '1-3 years' to 24 months, 'Over 3 years' to 48 months, and 'Never' to 0 months.
- After conversion, the descriptive statistics for `Duration_Months` were calculated:
    - The **minimum duration** was 0.0 months (corresponding to 'Never').
    - The **maximum duration** was 48.0 months (corresponding to 'Over 3 years').
    - The **mean duration** was approximately 14.17 months.
- A histogram of `Duration_Months` revealed that there is a higher frequency of shorter durations (approximately 0-10 months), which corresponds to categories such as 'Never', 'Less than a month', and '1-6 months'. Frequencies for longer durations tend to decrease.

Insights or Next Steps

- To perform quantitative analysis on categorical data representing ranges, converting it into a numerical approximation (e.g., averages within ranges) is crucial.
- The data suggests that individuals or entities represented in this dataset are more frequently associated with shorter durations, indicating a potential skew in the underlying population or observation set. Further analysis could explore the implications of this distribution and the potential for weighting categories based on their typical duration range if aggregated statistics are needed.

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.

Prompt:

Generate two Python implementations for sorting a list of numbers: 1—a manual bubble sort algorithm, 2. Using Python's built-in sort method. After writing both versions, provide a short comparison of their efficiency and readability, explain overall code

Code & Output:

Here are two ways to sort a list of numbers in Python: one built from scratch to show the underlying logic, and one using Python's optimized internal tools.

1. Manual Bubble Sort Implementation

Bubble sort works by repeatedly stepping through the list, comparing adjacent elements, and swapping them if they are in the wrong order. This "bubbles" the highest value to the end of the list with each pass.

Python

```python
def bubble_sort(numbers):
    n = len(numbers)
    # Create a copy to avoid modifying the original list
    data = numbers.copy()

    for i in range(n):
        # Last i elements are already in place
        for j in range(0, n - i - 1):
            # Compare adjacent elements
            if data[j] > data[j + 1]:
                # Swap if the left element is greater than the right
                data[j], data[j + 1] = data[j + 1], data[j]
    return data

# Example Usage
unsorted_list = [64, 34, 25, 12, 22, 11, 90]
sorted_list = bubble_sort(unsorted_list)
print(f"Bubble Sort Result: {sorted_list}")
```

2. Python's Built-in Sort Method

Python uses an algorithm called **Timsort**, which is highly optimized for real-world data.

Python

```python
# Using the .sort() method (modifies the list in-place)
numbers_builtin = [64, 34, 25, 12, 22, 11, 90]
numbers_builtin.sort()

print(f"Built-in Sort Result: {numbers_builtin}")

# Alternatively, use sorted() to return a new list
# sorted_list = sorted(numbers_builtin)
```

Comparison and Explanation

Feature	Bubble Sort	Built-in `.sort()`
Efficiency (Time)	$O(n^2)$ - Slow for large lists.	$O(n \log n)$ - Very fast.
Readability	High for learning, low for production.	Excellent/Pythonic.
Space Complexity	$O(1)$ - In-place swaps.	$O(n)$ - Timsort requires temp space.
Best Use Case	Educational purposes only.	All production environments.

How the Code Works

1. **Bubble Sort Logic:** We use nested `for` loops. The outer loop ensures we repeat the process enough times, while the inner loop handles the actual comparison. Every time `data[j] > data[j + 1]` is true, the numbers swap positions. As the algorithm progresses, the largest unsorted number "bubbles" to its correct position at the end.
2. **Built-in Logic:** The `.sort()` method is written in C and uses Timsort. It looks for "runs" (already sorted sequences) in your data to minimize the number of comparisons. It is significantly faster than any manual basic algorithm you would write by hand.

Summary

While Bubble Sort is a great way to understand how algorithms manipulate memory and pointers, you should always use `.sort()` or `sorted()` in professional projects due to their superior speed and reliability.

Would you like me to explain how to sort more complex data, such as a list of dictionaries by a specific key?

Justification:

Gemini did exactly as it was tasked to generate both a manual bubble sort and using python's built in sort method in a clear and structured way. It made difference between algo learning and practical efficiency easy to understand and explained why bubble sort is useful for understanding logic but inefficient for large datasets. Overall it improved the understanding of choosing the right algorithm based on performance and real-world usage.