**Data Requirement:-**

Prior to commencing the capstone project, it is essential to load the necessary data for the Kafka implementation and Pyspark streaming. Given data for the project is stored in the Google Drive repository and after mounting the drive data is loaded in google colab.A computer code with red green and blue text

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

**Brief Kafka Architecture Explanation:**

Kafka is a distributed streaming platform that provides a robust and scalable architecture for building real-time data pipelines and streaming applications. The core components of Kafka include producers, brokers, topics, partitions, and consumers, which work together to create a highly efficient and fault-tolerant messaging system.

Here's an overview of the Kafka producer-consumer architecture:

* **Producers:** Producers are applications or components that generate and publish data to Kafka topics. They send records (messages) to Kafka brokers, which act as intermediaries between producers and consumers. Producers can be various data sources, such as web servers, sensors, log files, or any application that generates data.
* **Brokers:** Kafka brokers are servers responsible for handling incoming messages from producers and serving them to consumers. They are the core components of the Kafka cluster and maintain the publish-subscribe model. Brokers store the incoming records in persistent storage on disk.
* **Topics:** Topics are logical channels or categories to which records are published. They act as message queues where producers write data, and consumers read from them. Each topic can have multiple partitions to store the data, and they are replicated across different brokers for fault tolerance.
* **Partitions:** Topics are divided into partitions to distribute the load and provide parallelism. Each partition is an ordered and immutable sequence of records. Producers can specify a key while sending a record, which helps determine the partition to which the record will be written.
* **Consumers:** Consumers are applications or components that read and process data from Kafka topics. They subscribe to one or more topics and consume records from the partitions within those topics. Consumers can be part of a consumer group, which enables them to share the load and work in parallel.
* **Consumer Groups:** Consumer groups are logical sets of consumers that work together to consume data from Kafka topics. Each consumer group is assigned to one or more partitions of a topic. Each partition can be consumed by only one consumer from a consumer group at any given time, which allows for parallel processing of messages.
* **Offset Management:** Kafka keeps track of the position of each consumer in each partition using offsets. Offsets represent the position of the last record consumed by a consumer in a partition. It allows consumers to pick up from where they left off in case of failure or rebalancing.
* **ZooKeeper (Optional, for older versions of Kafka):** In older versions of Kafka, ZooKeeper was used for managing cluster state, but in newer versions, Kafka now uses its internal metadata management system.

The Kafka producer-consumer architecture offers several advantages, including high scalability, fault tolerance, and real-time data processing capabilities. It has become a popular choice for building data pipelines, event-driven applications, log aggregation systems, and other streaming data solutions.

**Explanation of Yelp Dataset :** Yelp review dataset contains below fields,

* "**review\_id**": This is a unique identifier for each review.
* "**user\_id**": This field represents the ID of the user who wrote the review.
* "**business\_id**": This field contains the ID of the business that was reviewed.
* "**stars**": This is the rating the user gave to the business, on a scale from 1 (worst) to 5 (best).
* "**useful**": This is a count of the number of people who found the review useful.
* "**funny**": This is a count of the number of people who found the review funny.
* "**cool**": This is a count of the number of people who found the review cool.
* "**text**": This field contains the actual text of the review.

**Kafka Installation and Set-up on Google Colab Notebook:**

**Step 1**: As a steps 1, we have installed the Kafka and Zookeeper using following sets of command as mentioned below,

A screenshot of a computer

Description automatically generated

**Step 2**: In this step, we have run the Kafka and Zookeeper in port 9092 as mentioned by professor in notebook.

A close up of a text

Description automatically generated

**Step 3**: In this step, we have created a topic named as ‘**yelp\_reviews’** as our given dataset deals with yelp review data.

A close up of a text

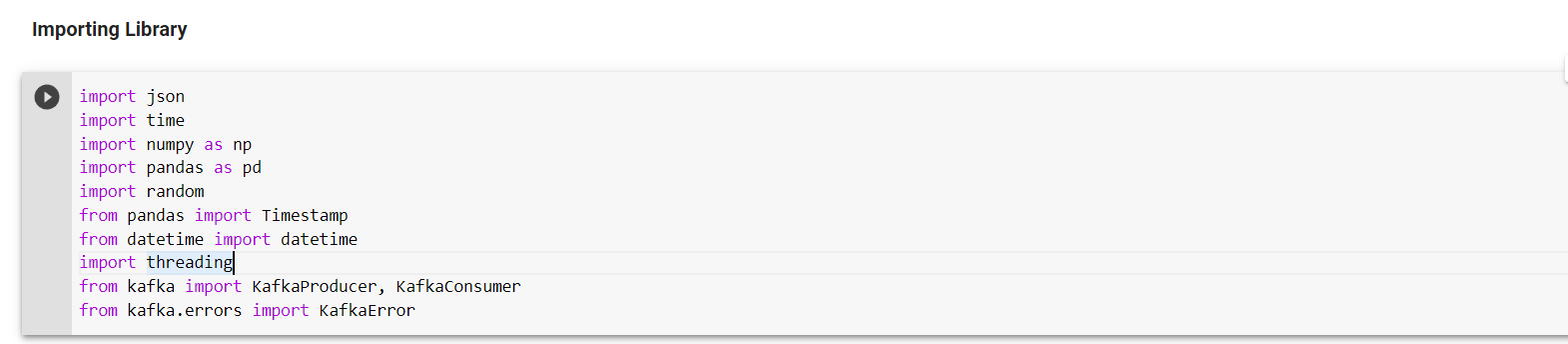
Description automatically generated

**Step 4**: In further steps we have open jdk and kafka python for performing the given task.



For performing all the task, we have made a logical segregation in the code for better maintainability and future reusability.

**Part 1: Importing Library:**



**Explanation and use of the imported library:**

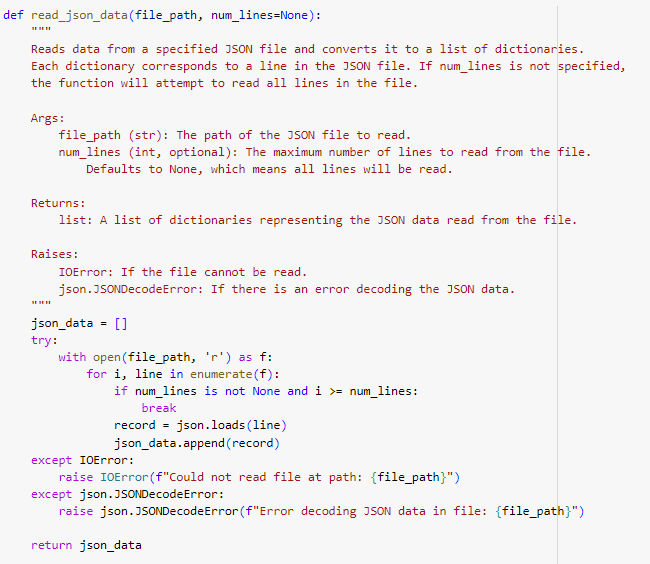
* **import json**: This imports the Python JSON module, which allows you to work with JSON data. It provides methods to encode Python objects into JSON strings and decode JSON strings into Python objects.
* **import time**: This imports the Python time module, which provides various time-related functions. It allows you to work with time values, sleep the program for a specific duration, and measure time intervals.
* **import numpy as np:** This imports the NumPy library and assigns it the alias "np." NumPy is a powerful library for numerical computing in Python. It provides support for large, multi-dimensional arrays and matrices, as well as a collection of mathematical functions to operate on these arrays efficiently.
* **import pandas as pd**: This imports the Pandas library and assigns it the alias "pd." Pandas is a widely used library for data manipulation and analysis. It offers data structures like DataFrame and Series, along with various functions for data cleaning, filtering, grouping, and merging.
* **import random**: This imports Python's built-in random module, which provides functions to generate random numbers, select random elements from a sequence, and shuffle elements in a list.
* **from pandas import Timestamp**: This imports the Timestamp class from the Pandas library. Timestamp is a specific data type in Pandas that represents a single timestamp, similar to a Python datetime object, but with additional functionalities for time series operations.
* **from datetime import datetime**: This imports the datetime class from the Python datetime module. The datetime class allows you to work with date and time values in Python, including formatting and arithmetic operations.
* **import threading**: This imports Python's threading module, which provides support for multi-threading in your programs. Threads allow you to execute multiple tasks concurrently, potentially improving performance and responsiveness.
* **from kafka import KafkaProducer, KafkaConsumer**: This imports the KafkaProducer and KafkaConsumer classes from the Kafka library. Kafka is a distributed streaming platform used for building real-time data pipelines and streaming applications.
* **from kafka.errors import KafkaError**: This imports the KafkaError class from the kafka.errors module. KafkaError represents common errors that can occur when working with Kafka, such as connection errors or message handling issues.

**Part 2: Common Variables:** Required common variables for storing the topic name used in Kafka and yelp academic review file path in the google drive link is stored in two variables which will be used in further part of the code

A close-up of a computer screen

Description automatically generated

**Sequential Kafka Producer and Consumer Call:**

**Part 3: Common Method:**

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

Explanation and the parameter details are mentioned clearly at the top of each common methods in details.

**Part 4: Kafka Operation:** This section has two parts, in first part we are creating **a Kafka Producer** and pushing the messages from the yelp dataset to Kafka and in next part we are creating **Kafka Consumer** for consuming those messages.



A computer code with black text

Description automatically generated

A screenshot of a computer program

Description automatically generated

A white background with black lines

Description automatically generated

As per problem statement ‘**10 records that are written to Kafka, separated by a sleep time of 10 seconds until 100 records are written**’ has given so we have taken randomly 100 messages from the yelp dataset and sending batch of 10.

Below are the screenshots of the batch wise operation,

A close-up of a newspaper

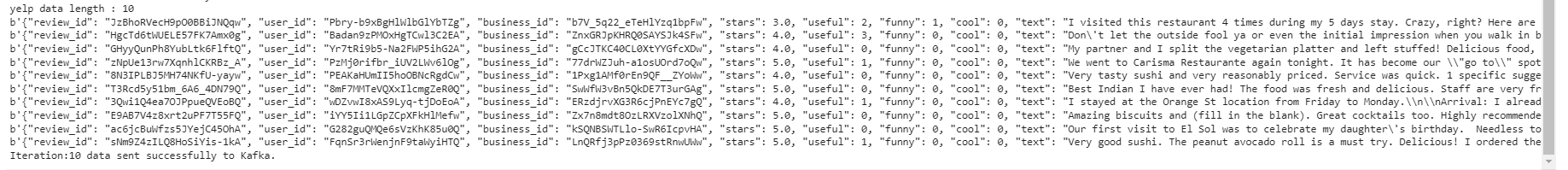
Description automatically generated

A close-up of a document

Description automatically generated

A close-up of a text

Description automatically generated



As per the problem statement after sending the details we are creating Kafka **“Kafka consumer will read in every 5 seconds from the producer”**



**Note**: I have used random selection of 100 record from the whole data which leads different data selection at each run for sending to Kafka producer.

**Parallelization of Kafka Producer and Consumer Call:** I will be trying to implement the parallelization call of both producer and consumer. In first approach will be trying to implement the same in Google Colab and in next step will do the same using only google colab in a single notebook.

**Approach 1:** GCP implementation,

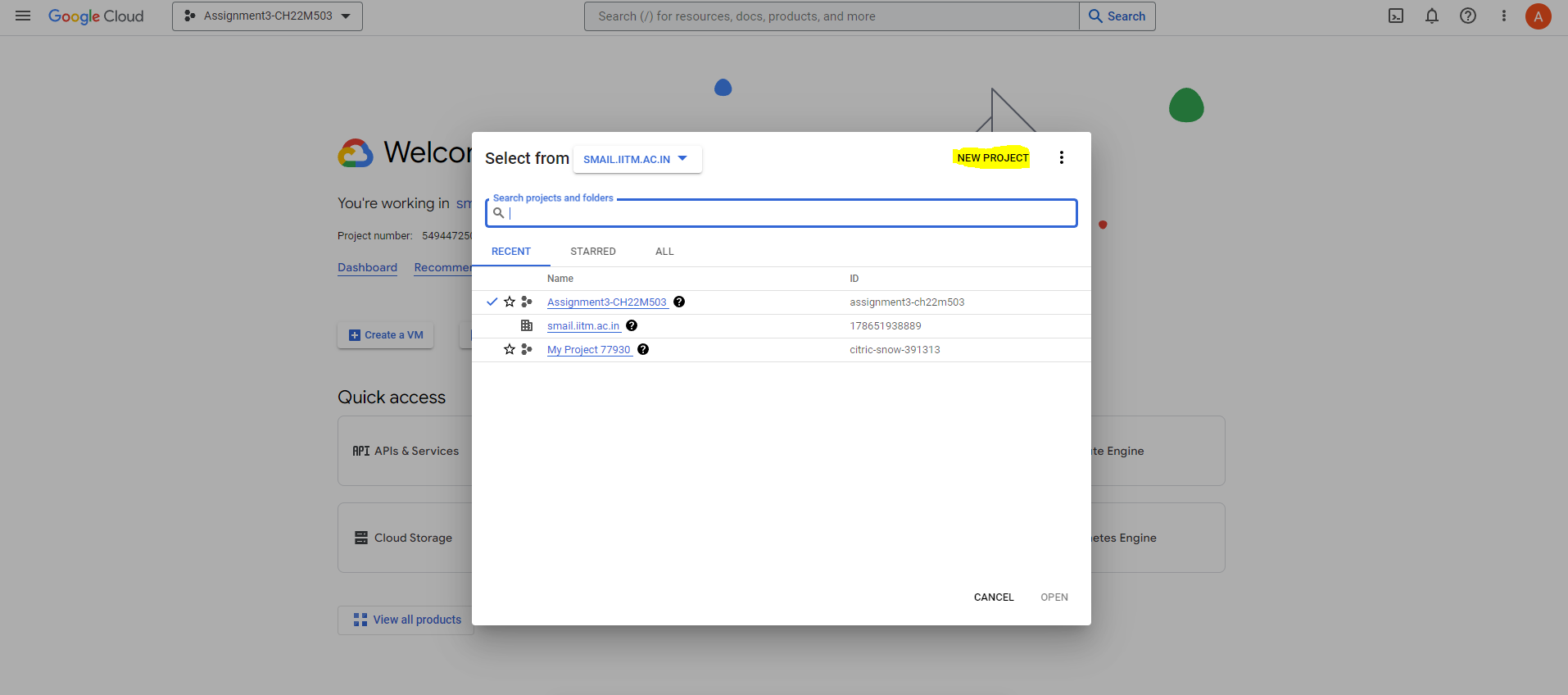
**Showing the credentials:**

A screenshot of a computer

Description automatically generated

Now I am trying to create the sequential call of the Kafka producer and consumer method which allow the parallelization in the entire process using GCP. Below are the steps followed,

1. As a first step created two separate jupyter notebook named as below to have the code for kafka consumer and kafka producer separately which allows to run the producer and consumer sequentially.
   1. CH22M503\_Kafka\_Producer.ipynb
   2. CH22M503\_Kafka\_Consumer.ipynb
2. **Create project:** Click on the create project button it will take you to the project creation page.



1. **Project creation page** – Fill out the Project name and click on the create button.

A computer screen shot of a computer

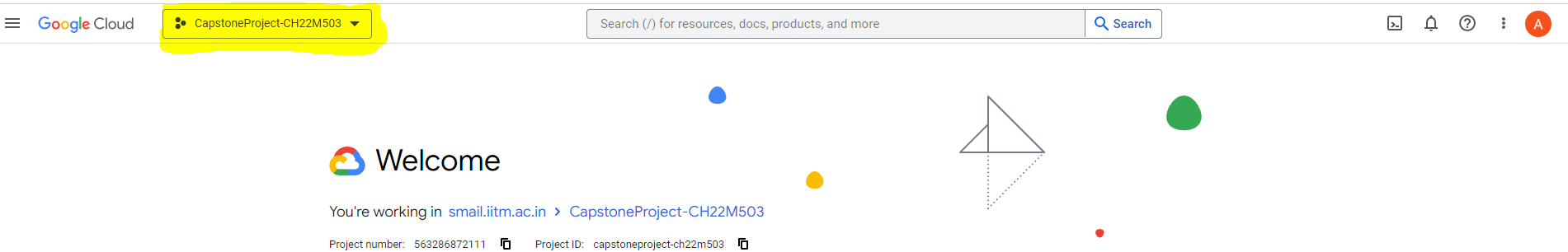
Description automatically generated

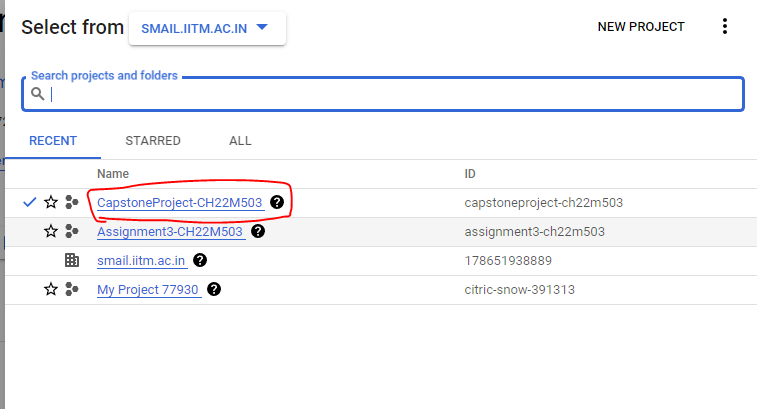
1. If the project gets created successfully then successful notification will appear at the top right corner in the notification section.

A screenshot of a computer

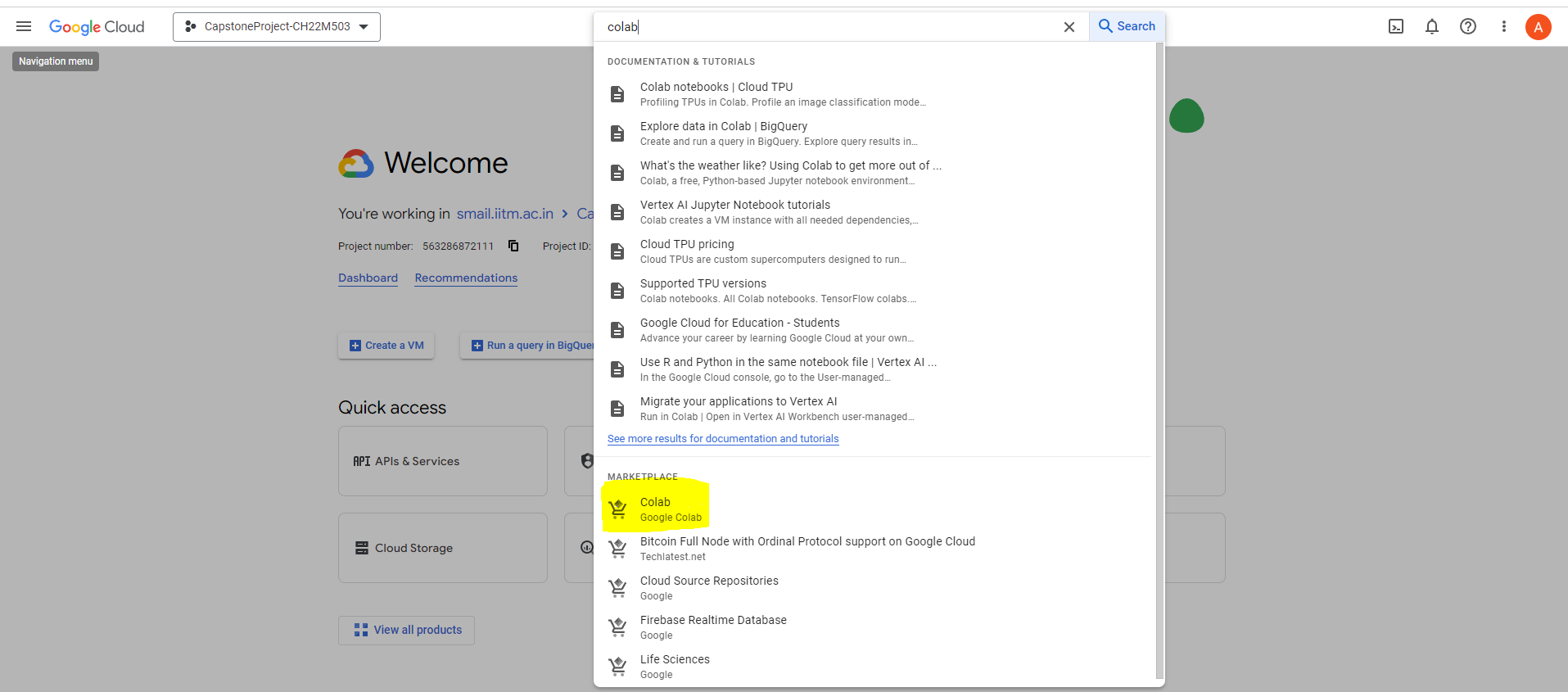
Description automatically generated

1. Choose the project from the top left dropdown and make sure your newly created project appears,





1. Create a Colab GCP VM –



1. Upon clicking, a dialogue box will appear, presenting a list of required APIs for enabling. Enter each API's name into the search bar individually to activate them sequentially. Keep in mind that attempting to access the APIs directly from the list won't lead you to the correct destination.

A white background with black text

Description automatically generated

1. Click ‘**Launch’**

A screenshot of a computer

Description automatically generated

1. Once the required API’s are enabled it will take you through ‘**New Colab Deployment**’ page.

A screenshot of a computer

Description automatically generated

1. Generate a fresh Colab-accessible virtual machine, making certain to complete all the steps marked for attention. Regarding the GPU, kindly omit that section, as GPU usage is unnecessary

A screenshot of a computer

Description automatically generated

1. Click on ‘Deploy’ button at the bottom of the page.

A screenshot of a computer

Description automatically generated

1. Patiently await the completion of your deployment. Jot down the "Instance Zone" and "Instance Name" displayed on the current screen.

A screenshot of a computer

Description automatically generated

1. Open your Colab notebook for the Producer module and adhere to the provided instructions for establishing a connection between the notebook and the recently generated virtual machine. Click the "**Connect**" button and subsequently opt for the "**Connect to a custom GCE VM**" choice.

A screenshot of a computer

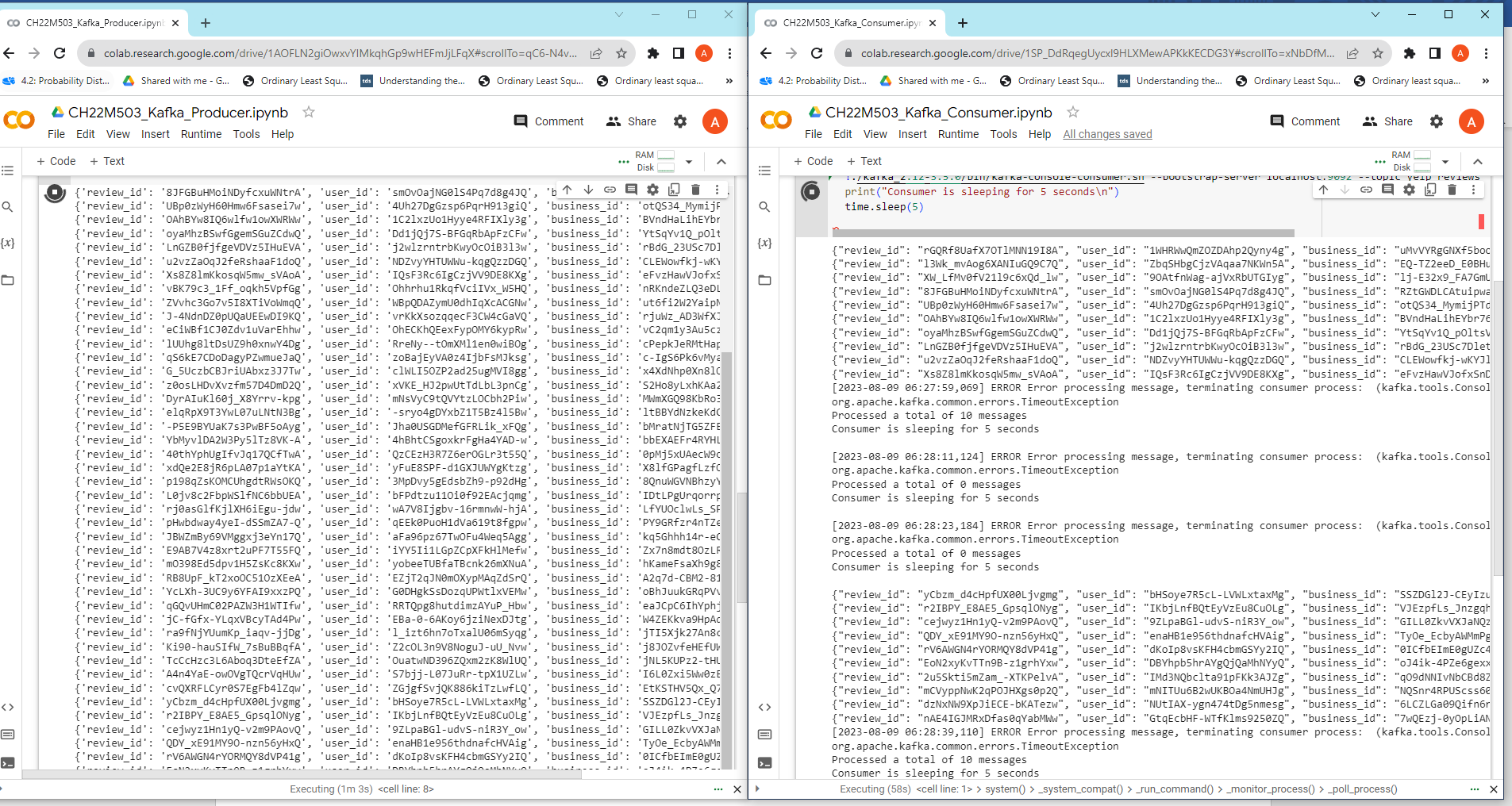
Description automatically generated

A screenshot of a computer

Description automatically generated

Note: Follow the same steps for Consumer file also.

1. Run the Producer and Consumer simultaneously and observe the results

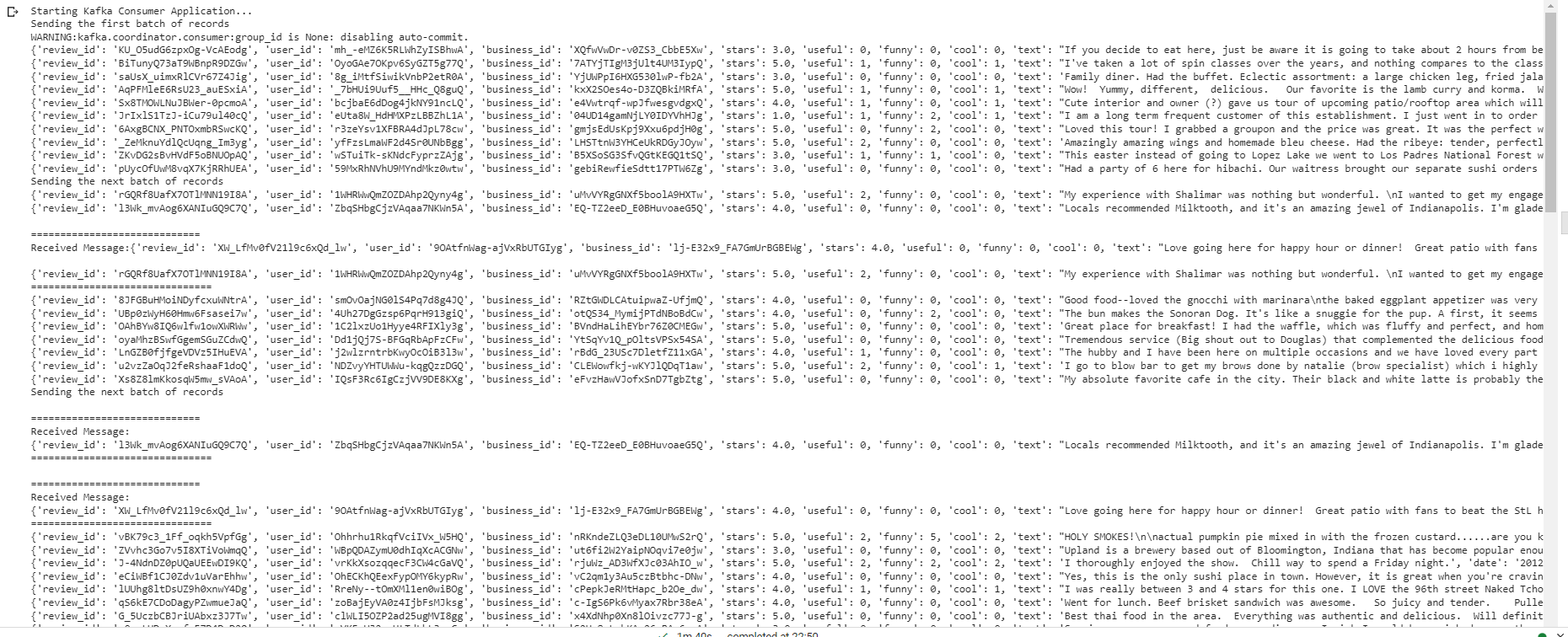


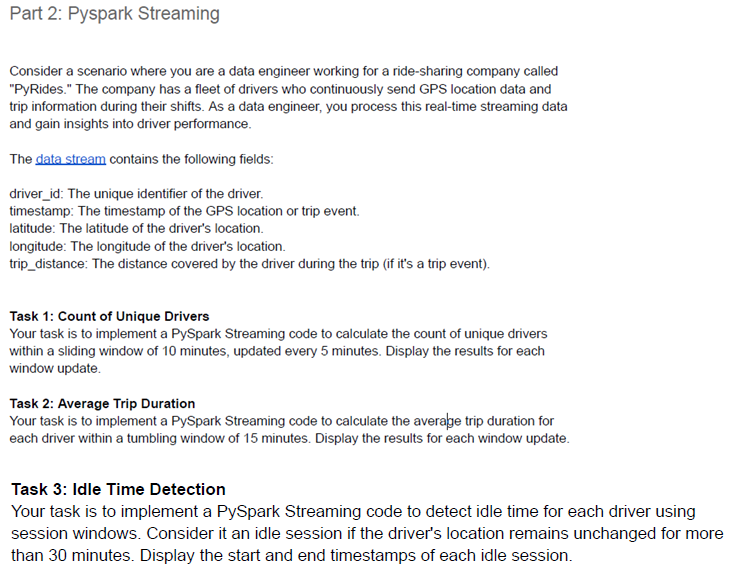
**Approach 2:** Google colab notebook implementation,

**Step 1**: Create two different function for implementing the parallalization of Kafka producer call and Kafka consumer call.

**Step 2**: Then using multiprocessing approach in python execute both the kafka producer and kafka consumer parallely as per the problem statement.

**Output of the parallal call:**





**PySpark Streaming** is a component of the Apache Spark ecosystem that enables processing and analyzing real-time streaming data. It provides an abstraction for handling continuous data streams, allowing developers to work with streaming data using familiar Spark APIs.

* **Pros of PySpark Streaming:**
  + **Unified API**: PySpark Streaming integrates seamlessly with the Spark ecosystem, allowing users to leverage their existing knowledge of Spark's RDD (Resilient Distributed Dataset) transformations and actions for real-time data processing.
  + **High-level Abstractions**: PySpark Streaming provides high-level abstractions like DStreams (Discretized Streams), which abstract away the complexities of managing streaming data and allow developers to focus on the business logic.
  + **Fault Tolerance**: Like other Spark components, PySpark Streaming offers fault tolerance through lineage information. It can recover lost data due to node failures.
  + **Scalability**: PySpark Streaming can easily scale to handle large volumes of streaming data by taking advantage of Spark's distributed computing capabilities.
  + **Support for Windowing**: It supports various windowing operations (sliding and tumbling windows) that enable processing data within specific time intervals.
  + **Broad Data Source Support**: PySpark Streaming can ingest data from various sources such as Kafka, Flume, HDFS, S3, and more, making it versatile for integrating with different data pipelines.
  + **Integration with Batch Processing**: PySpark Streaming seamlessly integrates with batch processing in Spark, allowing users to combine real-time and batch processing for comprehensive data analysis.
* **Cons of PySpark Streaming**:
  + **Micro-Batching Model**: PySpark Streaming operates using a micro-batching model, where data is processed in small batches. This introduces some latency as data is collected over intervals.
  + **Latency**: Due to the micro-batching approach, the latency is higher compared to true real-time processing systems that operate on event-triggered updates.
  + **Resource Management Overhead**: Managing a streaming application involves maintaining resources for both the streaming application itself and the Spark cluster, which can lead to increased operational complexity.
  + **Complexity for Low-latency Use Cases**: For use cases requiring very low-latency processing, where the delay between data arrival and processing needs to be minimal, PySpark Streaming might not be the ideal choice.
  + **Learning Curve**: Although PySpark Streaming abstracts away some of the complexities of real-time data processing, users still need to learn its concepts and APIs, which might require some initial effort.
  + **Resource Consumption**: Since Spark is resource-intensive, setting up and maintaining a Spark cluster for streaming can require substantial resources and infrastructure.

**Explanation of Pyride data:**

This is a JSON data structure that contains data related to a trip or multiple trips made by a driver. Each object (enclosed by {}) in the dataset corresponds to a single trip event. Let's look at the fields for each trip:

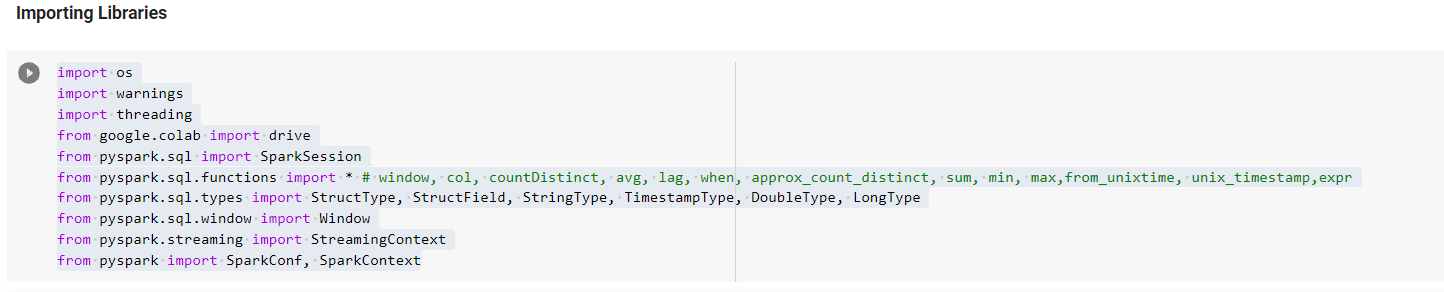
* "**driver\_id**": This is a unique identifier for a driver. In this case, both trips were completed by driver "D001".
* "**timestamp**": This field represents the time when the trip event was logged. It appears to be in Unix time format, which is the number of seconds that have passed since 00:00:00 Thursday, 1 January 1970, Coordinated Universal Time (UTC), minus leap seconds.
* "**latitude**" and "**longitude**": These fields represent the geographic coordinates of the driver at the time the event was logged. They give the position of the driver on the globe.
* "**trip\_distance**": This represents the distance of the trip in some unit, probably miles or kilometers.
* "**event\_type**": This field represents the type of event. In this case, it is "Trip", which might suggest that the data logged is related to a trip the driver has taken and “GPS” indicating that the driver is idle or on no trip.

For performing all the task, we have made a logical segregation in the code for better maintainability and future reusability.

**Step 1** : **Install Pyspark** : As a pre-requisite package we have installed pyspark before solving the problem statement.



**Step 2 : Importing Library:**



**Import necessary modules:**

* **os**: Provides a way to interact with the operating system, e.g., managing files and directories.
* **warnings:** Allows control over issuing warnings.
* **threading:** Enables multithreading support for running multiple threads simultaneously.
* **google.colab.drive**: Used for accessing and mounting Google Drive in a Colab environment.
* **pyspark.sql.SparkSession:** Used for creating and managing a Spark session.
* **pyspark.sql.functions.\*:** Imports all functions from the pyspark.sql.functions module, providing a wide range of SQL functions for DataFrame manipulation.
* **pyspark.sql.types**: Provides classes for defining custom data types in Spark.
* **pyspark.sql.window.Window:** Enables the creation of window specifications for window **functions.**
* **pyspark.streaming.StreamingContext**: Represents the entry point for streaming functionality in Spark.
* **pyspark.SparkConf, pyspark.SparkContext:** Used for configuring and creating a Spark context.

**Step 3: Common Variables**

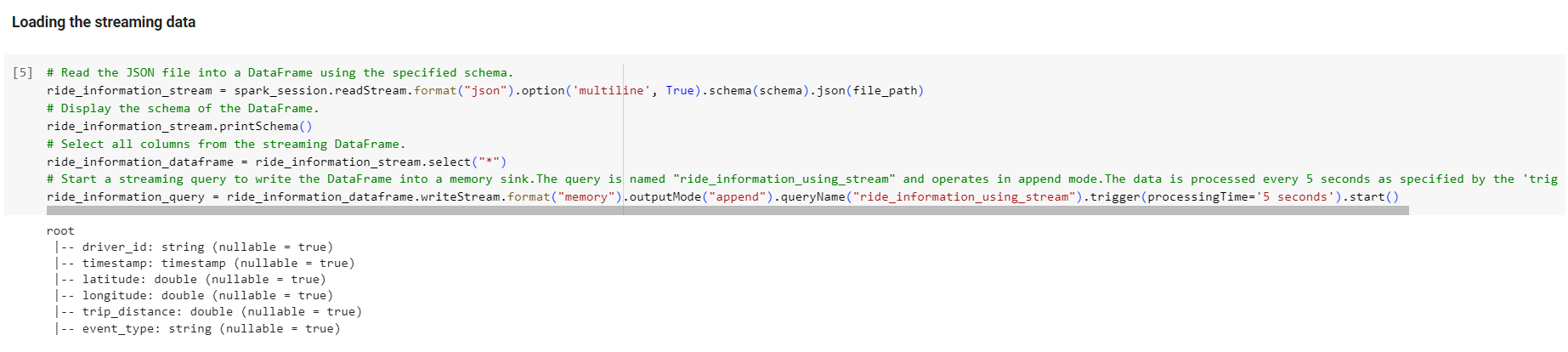
A screen shot of a computer code

Description automatically generated

* **Creating a SparkSession:**
  + A SparkSession is being created using the SparkSession.builder.
  + master("local"): This specifies that Spark should run in local mode.
  + appName("PyRidesDriverPerformance"): Assigns a name to the Spark application.
  + config('spark.ui.port', '4050'): Configures the port for the Spark UI.
  + getOrCreate(): Attempts to reuse an existing SparkSession or create a new one if none exists.
* **File Path and Data Directory:**
  + file\_path: Specifies the directory path where the JSON data is located. This should be a directory on the Google Drive.
* **Time Durations:**
  + window\_duration\_part1 and slide\_duration\_part1: Define the duration of a sliding window and the slide duration for its movement. In this case, a sliding window of 10 minutes slides every 5 minutes.
  + window\_duration\_part2: Specifies a tumbling window duration of 15 minutes.
  + session\_gap\_duration: Defines the session window duration, set at 30 minutes.
* **Idle Detection Threshold:**
  + idle\_threshold\_seconds: Specifies the threshold for detecting idle sessions, set to 1800 seconds (30 minutes).
* **Schema Definition:**
  + schema: Defines the schema for the streaming data using StructType and StructField. It includes fields like driver\_id, timestamp, latitude, longitude, trip\_distance, and event\_type

**Step 4: Loading Data**

* As per the problem statement data should be read as a stream and for doing the same, we are reading the data as a stream.





**Steps for loading the streaming data**:

* Read the data from the given google drive path using read stream method using predefine schema and allowing the multiline readability.
* Once the data is captured as a stream data convert that into pyspart dataframe and start the writestream process with a specific query name.
* Once the process is completed use that query name in next cell to read and view the data.

As per the problem statement there are three tasks need to be performed,

1. **Count the Unique Driver**
2. **Calculate the Average Trip Duration**
3. **Driver’s Idle Time Detection**

**Task 1 :**

**Approach of the Task 1:** Based on my comprehension, the initial step entails gathering information pertaining to all drivers operating within the stipulated timestamp range. Subsequently, it becomes imperative to identify the distinct driver entities among this cohort. My comprehensive evaluation, executed within an Excel framework, has also revealed the same result which I have received in programming.

**Screenshots of Task 1 code:**

**Step 1:** Sorting, Grouping and Counting the unique driver’s id for a given window

A screenshot of a computer program

Description automatically generated

**Explanation of Task 1 code:**

* **Sorting the DataFrame:**
  + ride\_information DataFrame is being sorted in ascending order based on two columns: 'driver\_id' and 'timestamp'.
  + This step ensures that the data is organized by driver and timestamp for further processing.
* **Grouping and Counting Unique Drivers:**
  + ride\_information DataFrame is grouped by time windows defined by the window\_duration\_part1 and slide\_duration\_part1 parameters.
  + Within each time window, the code calculates the count of distinct (unique) drivers using the countDistinct function. The result is aliased as "unique\_drivers\_count".
* **Printing a Descriptive Message:**
  + A message indicating the current task being performed is printed to the console.
  + In this case, the message is 'Task 1: Count of Unique Drivers :'.
* **Ordering the DataFrame:**
  + The DataFrame unique\_drivers\_count is ordered based on the 'window' column. This is done to prepare the data for display in chronological order.
* **Displaying the Ordered DataFrame:**
  + The ordered DataFrame ordered\_unique\_drivers\_data is displayed using the show() function.
  + The first argument of show() specifies the number of rows to display.
  + The second argument, False, specifies that the column values should not be truncated for display.
* **Overall, the code performs the following tasks:**
  + Sorts the ride information DataFrame.
  + Groups the data into time windows and counts the number of unique drivers in each window.
  + Prints a descriptive message.
  + Orders the result by time window.
  + Displays the ordered result, showing the count of unique drivers within each window.

Top of Form

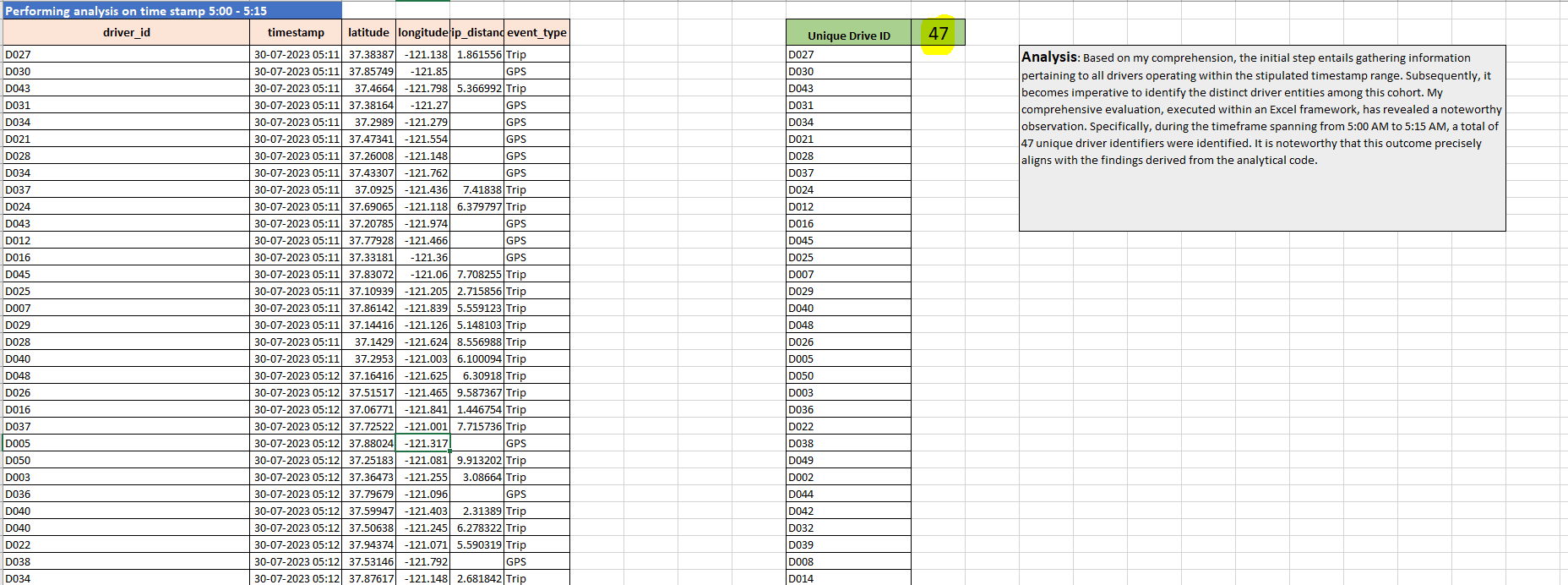
**Output of Task 1:**

A screenshot of a computer

Description automatically generated

Above table shows for each time duration how many unique drivers are available.

Same analysis has been done in the excel file for a single driver(**D001**) data from the py-ride dataset. Below is the screenshot for the same,



Specifically, during the timeframe spanning from 5:00 AM to 5:15 AM, a total of 47 unique driver identifiers were identified. This outcome precisely aligns with the findings derived from the programming code.

Note: Excel will attach in the document as well as the zip file

**Task 2:**

**Approach of Task 2:**

* In accordance with the outlined problem statement, my initial approach involved the comprehensive consideration of event types, specifically "Trip" and "GPS," to accurately compute trip durations. This selection was motivated by the need to establish consecutive timestamps, crucial for discerning between trips and idle periods.
* Upon successfully calculating the time durations for various events, my subsequent focus shifted to isolating data entries associated with the "**Trip**" event type. This enabled me to establish a clear relationship between each entry and its corresponding trip duration.
* The introduction of the **tumbling window** mechanism further enriched my analysis. By employing a 15-minute tumbling window approach, I methodically organized the trip data into meaningful groupings. This pivotal step paved the way for a straightforward averaging operation over these distinct windows, leading to the calculation of average trip durations.
* Through this methodical approach, I am not only maintained the integrity of consecutive timestamps but also achieved an accurate representation of actual trip durations. Consequently, the computed average trip duration now offers valuable insights into the overall trend of trips taken by drivers.

**Screenshots of Task 2 code:**

**Step 1:** Cleaning the data and calculate the lag timestamp and followed by calculate each event duration.



A computer code on a white background

Description automatically generated

**Step 2:** Filter only the “Trip” information, followed by grouping of data based on window and create a window wise average to get the average trip duration.

A white background with text

Description automatically generated

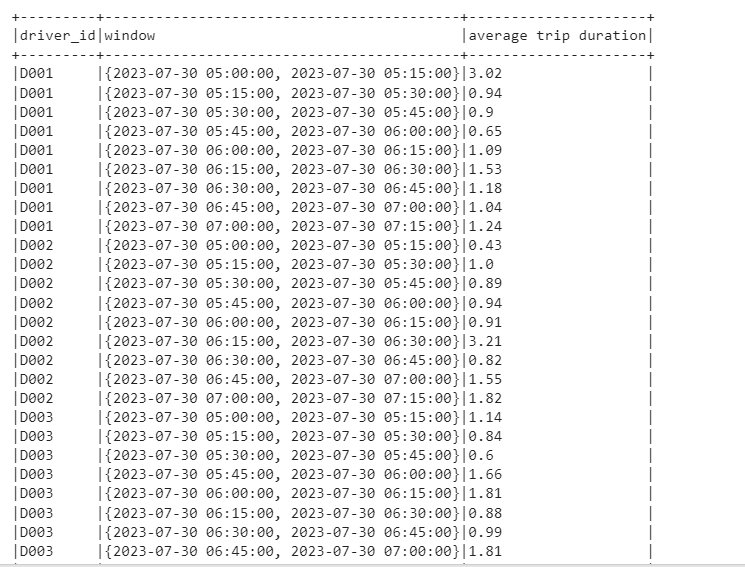
A close-up of a computer code

Description automatically generated

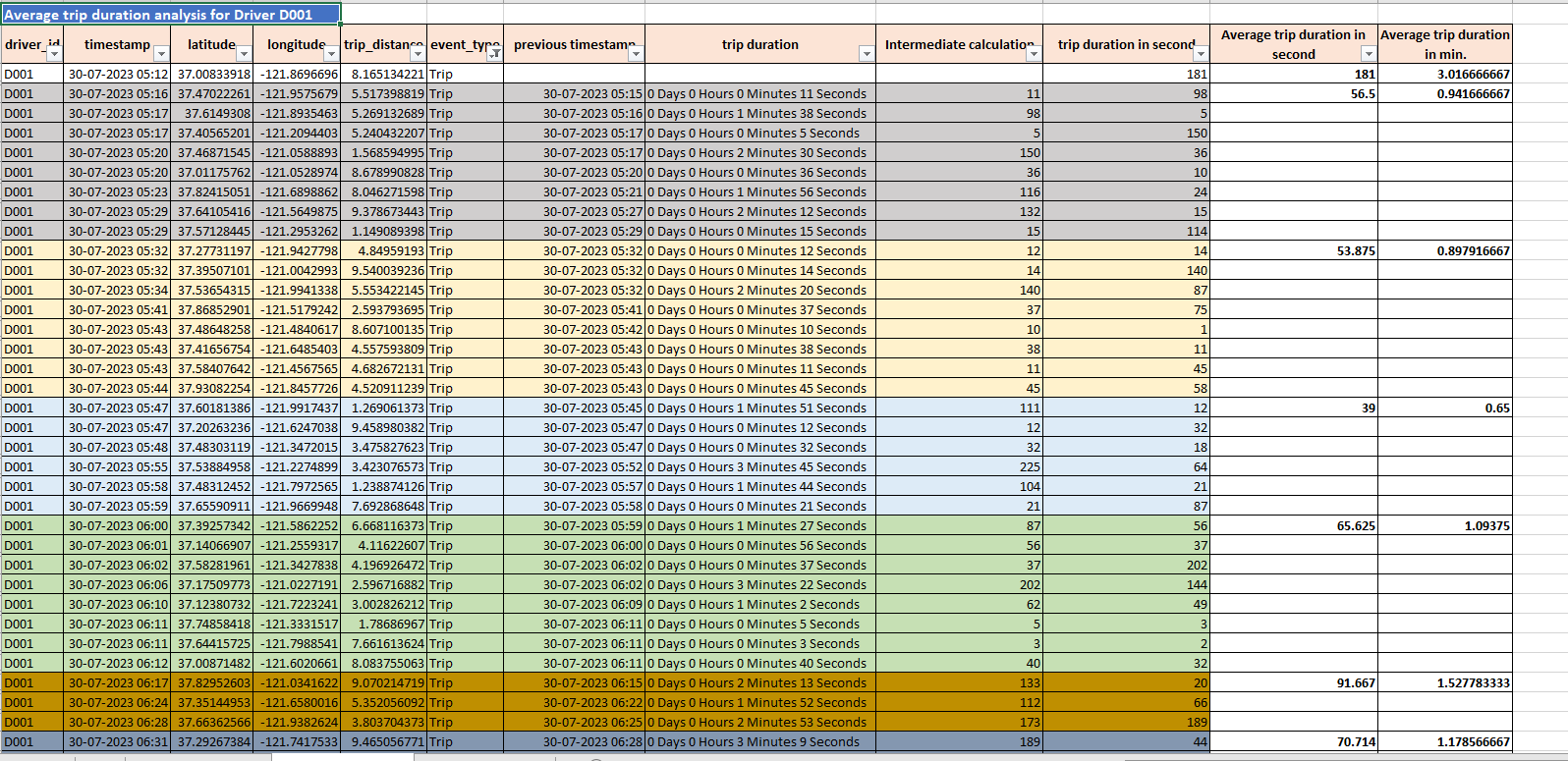
**Explanation of Task 2 code:**

* **Data Preparation:**
  + The code selects the relevant columns ("driver\_id", "timestamp", "event\_type") from the ride\_information DataFrame and stores it in the eventwise\_ride DataFrame.
  + It drops rows where all columns have null values using the dropna() function.
* **Windowing and Duration Calculation:**
  + A window specification named windows\_specs is defined for ordering data by driver and timestamp.
  + A new column named "prev\_timestamp" is added to the DataFrame using the lag() function to capture the previous event's timestamp for the same driver.
  + The duration between the current event's timestamp and the previous event's timestamp is calculated and stored in the "event\_duration" column.
* **Display of Data Within Window:**
  + The DataFrame eventwise\_data\_within\_window is displayed, showing the columns related to the calculated event duration.
* **Actual Event Duration Calculation:**
  + A window specification named window\_spec\_idle\_duration is defined for ordering by driver.
  + The "event\_duration\_actual" column is calculated by lagging the "event\_duration" column by -1 (previous row) using the lag() function.
* **Display of Data with Actual Event Duration:**
  + The DataFrame eventwise\_data\_within\_window is displayed again, this time showing the calculated actual event duration.
* **Filtering and Displaying "Trip" Events:**
  + The DataFrame only\_trip\_data is created by filtering out rows where the "event\_type" column is "Trip".
  + The resulting DataFrame is displayed, showing only the "Trip" event type data.
* **Calculating Average Trip Duration:**
  + The code calculates the average trip duration within the specified window for each driver.
  + The result is stored in the average\_trip\_duration DataFrame, with the average trip duration rounded to minutes.
* **Filtering and Displaying Calculated Average Trip Duration:**
  + Rows where the calculated average trip duration is not null are filtered.
  + The resulting DataFrame is displayed, showing the calculated average trip duration along with the count of rows.

**Output of Task 2: Snapshot of the output screenshot has been given,**



Same analysis has been done in the excel file for a sample driver data from the py-ride dataset. Below is the screenshot for the same**,**



Average trip duration for the excel calculated approach yield same result with programming.

Note: Excel will attach in the document as well as the zip file

**Task 3 :**

**Approach of Task 3:**

* In accordance with the outlined problem statement, the provided dataset is to be interpreted as driver-related data, detailing their transitions between non-trip states, categorized under the "GPS" event type, and trip states, categorized under the "Trip" event type.
* Subsequently, the "GPS" event type in the given data signifies periods of driver inactivity or idle time. To address this, a comprehensive analysis was conducted to compute the duration of each event. Following this, a meticulous assessment was carried out to determine whether any "GPS" event type instances exceeded the designated threshold of 30 minutes. This evaluation enabled the identification of periods marked as idle times, while events falling below the threshold were excluded from this classification.

**Screenshots of Task 3 code:**

**Step 1:** Cleaning the data and calculate the lag timestamp and followed by calculate each event duration.

A close-up of a computer screen

Description automatically generated

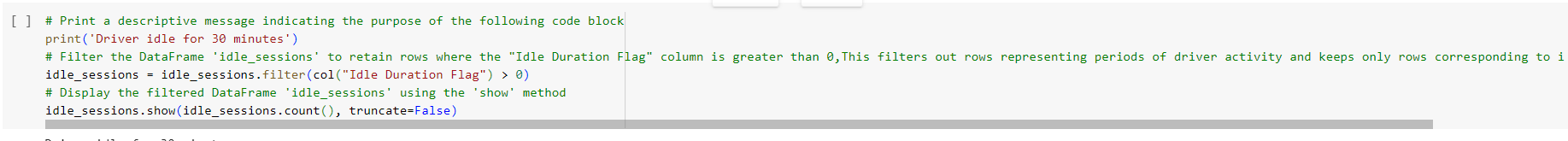
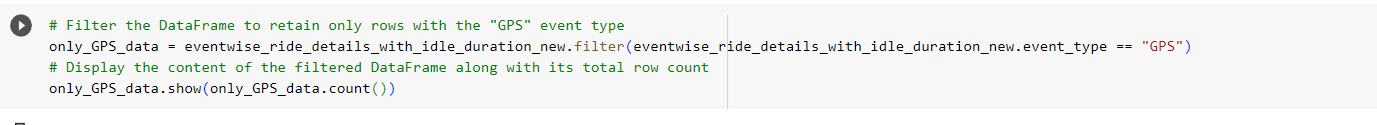
A screenshot of a computer code

Description automatically generated

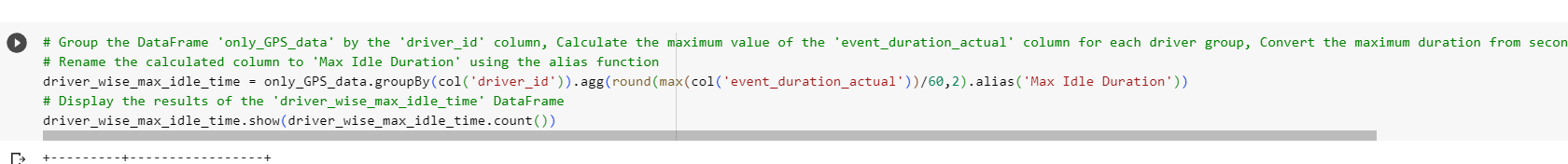
A screen shot of a computer code

Description automatically generated

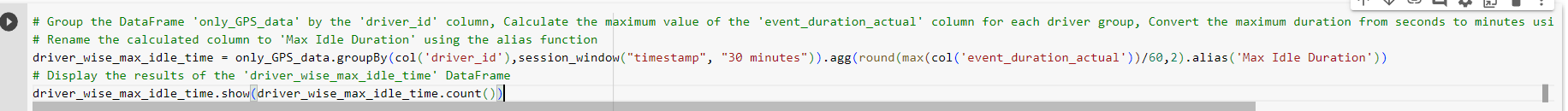
**Step 2:** Filter only the “GPS” information, followed by comparing the event duration with idle threshold value of 30 minute and mark each event with a idle duration flag. If the flag is 1 that indicates the driver is idle otherwise not.



**Approach 1 – Without using session window:**



**Approach 2 –Using session window of 30 minutes:**



Using either session window or not does not make any changes in the final output.

**Explanation of Task 3 code:**

* Selects specific columns "driver\_id", "timestamp", and "event\_type" from the ride\_information DataFrame.
* Drops rows where all columns have missing (null) values.
* Displays the resulting DataFrame named eventwise\_ride.
* Defines a window specification for partitioning the data by driver\_id and ordering by timestamp.
* Creates a new DataFrame eventwise\_ride\_within\_window by adding a new column prev\_timestamp, which contains the timestamp of the previous row within the same partition.
* Displays the contents of the DataFrame eventwise\_ride\_within\_window.
* Calculates the duration of each event within the specified window by subtracting the prev\_timestamp from the current timestamp. Adds a new column event\_duration to the DataFrame eventwise\_ride\_details\_with\_idle\_duration.
* Displays the DataFrame containing ride details along with calculated event durations.
* Creates a window specification for ordering by the "driver\_id" column.
* Adds a new column event\_duration\_actual to the DataFrame by calculating the lag of the "event\_duration" column with an offset of -1 using the defined window specification. This effectively moves the "event\_duration" value one row up in the DataFrame for each driver.
* Displays the resulting DataFrame eventwise\_ride\_details\_with\_idle\_duration\_new with the added column. The "count()" function is used to show all rows in the DataFrame without truncation.
* Filters the DataFrame to retain only rows with the "GPS" event type, creating a new DataFrame named only\_GPS\_data.
* Displays the content of the filtered DataFrame only\_GPS\_data along with its total row count.
* Adds a new column "Idle Duration Flag" to the DataFrame idle\_sessions. This column is calculated based on whether the "event\_duration\_actual" is greater than the idle threshold in seconds (30 minutes).
* Displays the DataFrame idle\_sessions containing the added column.
* Prints a descriptive message indicating the purpose of the following code block.
* Filters the DataFrame idle\_sessions to retain rows where the "Idle Duration Flag" column is greater than 0, effectively filtering out periods of driver activity and keeping only rows corresponding to idle sessions.
* Displays the filtered DataFrame idle\_sessions using the show method.
* Groups the DataFrame only\_GPS\_data by the 'driver\_id' column.
* Calculates the maximum value of the 'event\_duration\_actual' column for each driver group, converts the maximum duration from seconds to minutes using the round function with 2 decimal places, and renames the calculated column to 'Max Idle Duration' using the alias function. (This steps has been done using session window of 30 min as well as without any session window and displayed the result in both cases)
* Displays the results of the DataFrame driver\_wise\_max\_idle\_time.

**Output of Task 3:**

A close-up of a line

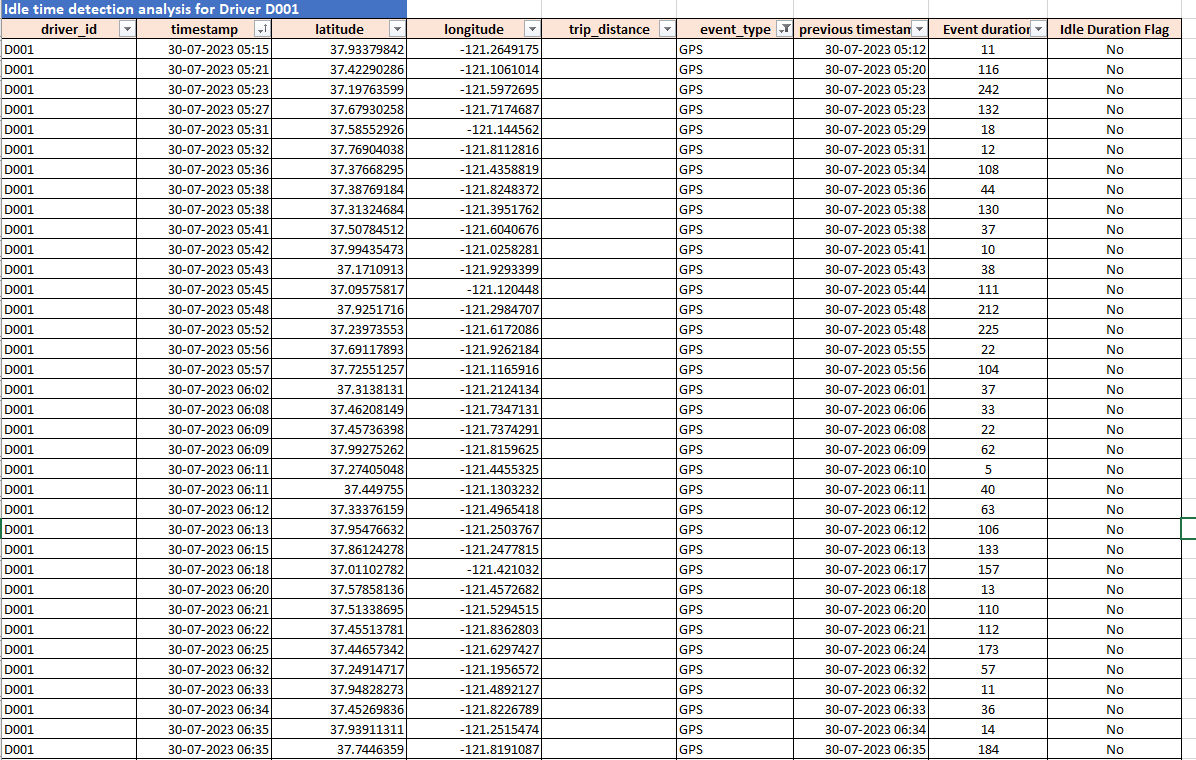
Description automatically generated

A paper with numbers and numbers

Description automatically generated

With the programming result I have conclusion that None of the drivers are idle for 30min continuously.

Same analysis has been done in the excel file for a sample driver data from the py-ride dataset for D001 driver id which also indicates same of coding output. Below is the screenshot for the same,



**Analysis Excel file for Q2 :**



Figure Analysis of the Q2 Each Task with given data

**Summary:**

|  |  |
| --- | --- |
| Task | Summary of approach and analysis |
| Task 1 | In this task, the initial step involves gathering information for all drivers operating within the provided timestamp range. The dataset is examined to identify distinct driver identifiers. A detailed Excel analysis was performed, revealing that during the time interval of 5:00 AM to 5:15 AM, a total of 47 unique driver IDs were recorded. This observation aligns precisely with the findings obtained from the code. The systematic approach ensures accurate identification of drivers and their respective activity during the specified time window. |
| Task 2 | The task revolves around calculating accurate trip durations. A comprehensive strategy was employed, considering both "Trip" and "GPS" event types. This selection aimed to establish consecutive timestamps crucial for differentiating between trips and idle periods. After calculating time durations for each event, the focus shifted to isolating "Trip" event entries. This facilitated a clear correlation between entries and their corresponding trip durations. The introduction of a 15-minute tumbling window approach enhanced the analysis. Grouping trip data in such windows allowed straightforward averaging, enabling the calculation of average trip durations. This approach maintains timestamp integrity and provides insights into drivers' trip trends. |
| Task 3 | This task interprets the dataset as driver-related data, distinguishing non-trip ("GPS") states from trip ("Trip") states. The goal is to identify periods of driver inactivity or idle time indicated by "GPS" events. To address this, event durations are computed, and instances exceeding the 30-minute threshold are classified as idle time. This process accurately identifies periods of inactivity, while excluding events below the threshold. The analysis was applied to driver D001, revealing the successful detection of idle times. The alignment between manual analysis and code-based findings reaffirms the effectiveness of the approach. This systematic method offers insights into drivers' activity and idle periods. |

**End of Capstone Project Report**