**KUMARAGURU COLLEGE OF TECHNOLOGY**

**Department of Computer Science and Engineering**

**DATA PROCESSING TECHNIQUES**

**Final Assessment Report**

**Submitted By:**

**Name:Rithikka A**  
**Roll No:** 23BCS135

**Course:** Data Processing Techniques

**Date of Submission:** 16-October-2025

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| | **Sl. No.** | **Topic** | **Page No.** | | --- | --- | --- | | 1 | Front Page | 1 | | 2 | Data Preprocessing Challenge | 3 | | 3 | Real-Time Data Streaming Challenge | 8 | | 4 | Incremental Data Processing Challenge | 17 | | 5 | In-Memory Data Processing Challenge | 24 | | 6 | Conclusion | 27 | |

**1.Data Preprocessing Challenge**

**1.Aim**

To perform complete data preprocessing on a raw IoT sensor dataset using Apache Spark (PySpark) in Linux.  
The task involves:

* Handling missing values
* Fixing data type inconsistencies
* Removing duplicates
* Normalizing numerical data
* Adding a new engineered feature (Temperature-to-Humidity Ratio)

**2. Tools & Environment**

| **Tool/Technology** | **Version / Details** |
| --- | --- |
| Operating System | Ubuntu 22.04 LTS |
| Python | 3.10 |
| Apache Spark | 3.5.0 |
| PySpark | 3.5.0 |
| Pandas | 2.1.1 |
| Numpy | 1.26.0 |

**4.Step-by-Step Instructions**

**Step 1:** Install Required Tools

sudo apt update

sudo apt install python3-pip

pip3 install pyspark pandas numpy

**Step 2:** Prepare Dataset

* Create a CSV or in-memory dataset with columns: sensor\_id, temperature, humidity, pressure, status, timestamp.

**Step 3:** Start Spark Session

from pyspark.sql import SparkSession

spark = SparkSession.builder.appName("SensorDataPreprocessing").getOrCreate()

**Step 4:** Load Dataset

df = spark.read.csv("sensor\_data.csv", header=True, inferSchema=True)

**Step 5:** Handle Missing Values

from pyspark.sql.functions import mean, col, when

avg\_temp = df.select(mean(col("temperature"))).collect()[0][0]

avg\_hum = df.select(mean(col("humidity"))).collect()[0][0]

df = df.withColumn("temperature", when(col("temperature").isNull(), avg\_temp).otherwise(col("temperature")))

df = df.withColumn("humidity", when(col("humidity").isNull(), avg\_hum).otherwise(col("humidity")))

**Step 6:** Remove Duplicates

df = df.dropDuplicates()

**Step 7:** Normalize Numerical Columns

from pyspark.sql.functions import round as spark\_round

df = df.withColumn("temperature\_norm", spark\_round(col("temperature")/100, 2))

df = df.withColumn("humidity\_norm", spark\_round(col("humidity")/100, 2))

**Step 8:** Feature Engineering (Optional)

df = df.withColumn("temp\_hum\_ratio", spark\_round(col("temperature")/col("humidity"), 2))

**Step 9:** Show Clean Data

df.show()

**5. Python Code (preprocessing.py)**

from pyspark.sql import SparkSession

from pyspark.sql.functions import col, mean, when, round as spark\_round

# Start Spark session

spark = SparkSession.builder.appName("SensorDataPreprocessing").getOrCreate()

# Load sample data

data = [

("S1", 25.3, 60.2, 1013, "ON", "2025-10-16 08:00:00"),

("S2", None, 58.7, 1011, "OFF", "2025-10-16 08:01:00"),

("S3", 29.8, None, 1012, "ON", "2025-10-16 08:02:00"),

("S1", 25.3, 60.2, 1013, "ON", "2025-10-16 08:00:00") # Duplicate

]

columns = ["sensor\_id", "temperature", "humidity", "pressure", "status", "timestamp"]

df = spark.createDataFrame(data, columns)

# Handle missing values with mean

avg\_temp = df.select(mean(col("temperature"))).collect()[0][0]

avg\_hum = df.select(mean(col("humidity"))).collect()[0][0]

df = df.withColumn("temperature", when(col("temperature").isNull(), avg\_temp).otherwise(col("temperature")))

df = df.withColumn("humidity", when(col("humidity").isNull(), avg\_hum).otherwise(col("humidity")))

# Remove duplicates

df = df.dropDuplicates()

# Normalize temperature & humidity

df = df.withColumn("temperature\_norm", spark\_round(col("temperature") / 100, 2))

df = df.withColumn("humidity\_norm", spark\_round(col("humidity") / 100, 2))

# Feature engineering: Temp/Humidity ratio

df = df.withColumn("temp\_hum\_ratio", spark\_round(col("temperature") / col("humidity"), 2))

# Show final preprocessed data

df.show()

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A black screen with white text

AI-generated content may be incorrect.

**8. Conclusion**

Data preprocessing successfully cleaned and standardized the raw IoT dataset.  
The dataset is now ready for downstream tasks like streaming analytics or model training.

**2.Real-Time Streaming using Apache Kafka**

**1.Aim**

To develop a real-time streaming application using Apache Kafka and Python that:

1. Produces simulated sensor data (IoT sensor readings).
2. Consumes and processes the data in real-time.
3. Performs basic analytics:
   * Rolling averages (temperature & humidity)
   * Regression prediction (humidity based on temperature)
   * Classification prediction (status ON/OFF)

**2. Tools & Environment**

| **Tool/Technology** | **Version / Details** |
| --- | --- |
| Operating System | Ubuntu 22.04 LTS |
| Python | 3.10 |
| Kafka | 3.9.0 |
| Kafka-python | 2.0.2 |
| Scikit-learn | 1.3.0 |
| Pandas / Numpy | 2.1.1 / 1.26.0 |
| Virtual Environment | Python venv |

**4.Step-by-Step Instructions:**

**1.Kafka Setup**

**# Start Zookeeper**

bin/zookeeper-server-start.sh config/zookeeper.properties

**# Start Kafka Broker**

bin/kafka-server-start.sh config/server.properties

**# Create Topic**

bin/kafka-topics.sh --create --topic sensor\_data --bootstrap-server localhost:9092 --partitions 1 --replication-factor 1

**# Verify Topic**

bin/kafka-topics.sh --list --bootstrap-server localhost:9092

# Output: sensor\_data

**5. Python Producer Code**

from kafka import KafkaProducer

import json, time, random

producer = KafkaProducer(

bootstrap\_servers='localhost:9092',

value\_serializer=lambda v: json.dumps(v).encode('utf-8')

)

sensor\_ids = ["S1","S2","S3","S4","S5"]

while True:

temp = round(random.uniform(20,35),2)

hum = round(random.uniform(40,80),2)

status = "OFF" if temp > 30 else "ON"

data = {

"sensor\_id": random.choice(sensor\_ids),

"temperature": temp,

"humidity": hum,

"status": status,

"timestamp": time.strftime("%Y-%m-%d %H:%M:%S")

}

producer.send('sensor\_data', value=data)

print(f"Produced: {data}")

time.sleep(1)

**5. Python Consumer Code**

from kafka import KafkaConsumer

import json

import numpy as np

from sklearn.linear\_model import LinearRegression, LogisticRegression

consumer = KafkaConsumer(

'sensor\_data',

bootstrap\_servers='localhost:9092',

auto\_offset\_reset='earliest',

value\_deserializer=lambda v: json.loads(v.decode('utf-8')),

consumer\_timeout\_ms=1000

)

window\_size = 5

temp\_window, hum\_window, status\_window = [],[],[]

reg\_model = LinearRegression()

clf\_model = LogisticRegression()

for message in consumer:

data = message.value

temp = data['temperature']

hum = data['humidity']

status = 1 if data['status']=="ON" else 0

temp\_window.append(temp)

hum\_window.append(hum)

status\_window.append(status)

if len(temp\_window) > window\_size:

temp\_window.pop(0)

hum\_window.pop(0)

status\_window.pop(0)

avg\_temp = round(np.mean(temp\_window),2)

avg\_hum = round(np.mean(hum\_window),2)

# Regression: Predict humidity from temperature

X\_reg = np.array(temp\_window).reshape(-1,1)

y\_reg = np.array(hum\_window)

reg\_model.fit(X\_reg, y\_reg)

pred\_hum = round(reg\_model.predict([[temp]])[0],2)

# Classification: Predict status ON/OFF

X\_clf = np.array(list(zip(temp\_window,hum\_window)))

y\_clf = np.array(status\_window)

pred\_status = "Unknown"

if len(X\_clf) >= 2:

clf\_model.fit(X\_clf, y\_clf)

pred\_status = "ON" if clf\_model.predict([[temp,hum]])[0]==1 else "OFF"

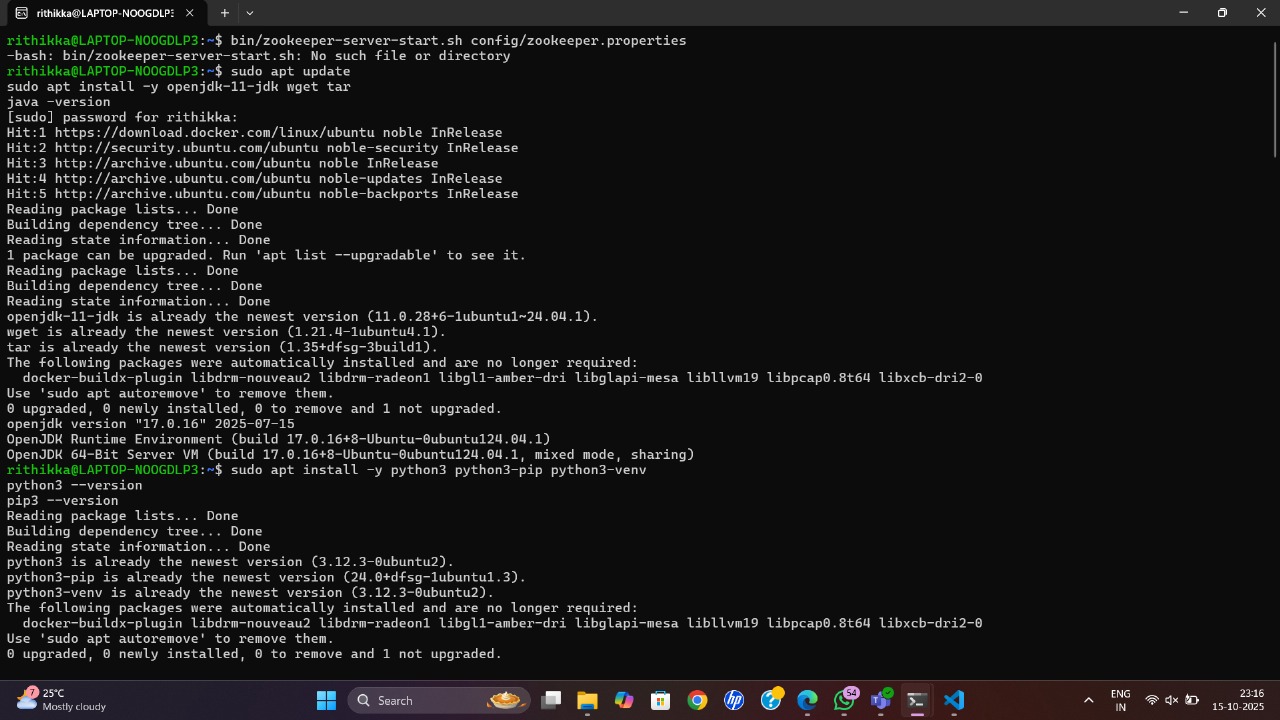
print(f"Consumed: {data}")

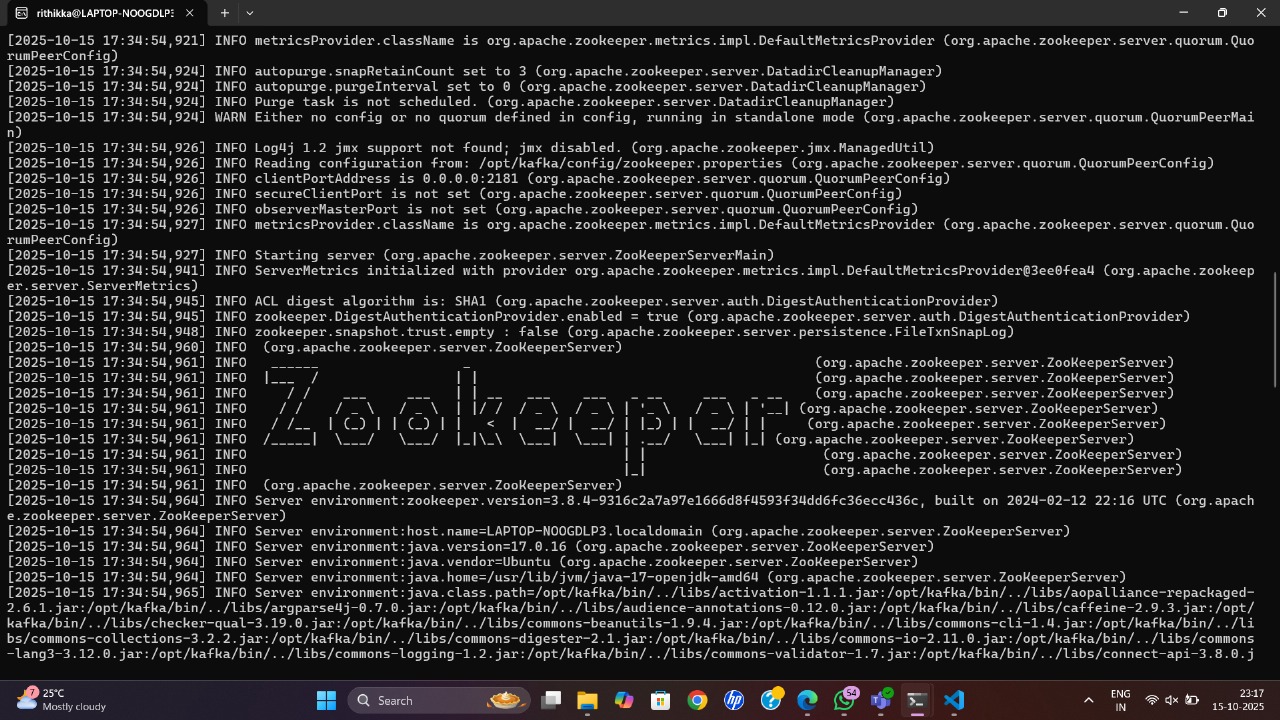
print(f"Rolling Avg Temp: {avg\_temp} | Rolling Avg Hum: {avg\_hum}")

print(f"Predicted Humidity: {pred\_hum}")

print(f"Predicted Status: {pred\_status}")

print("="\*80)



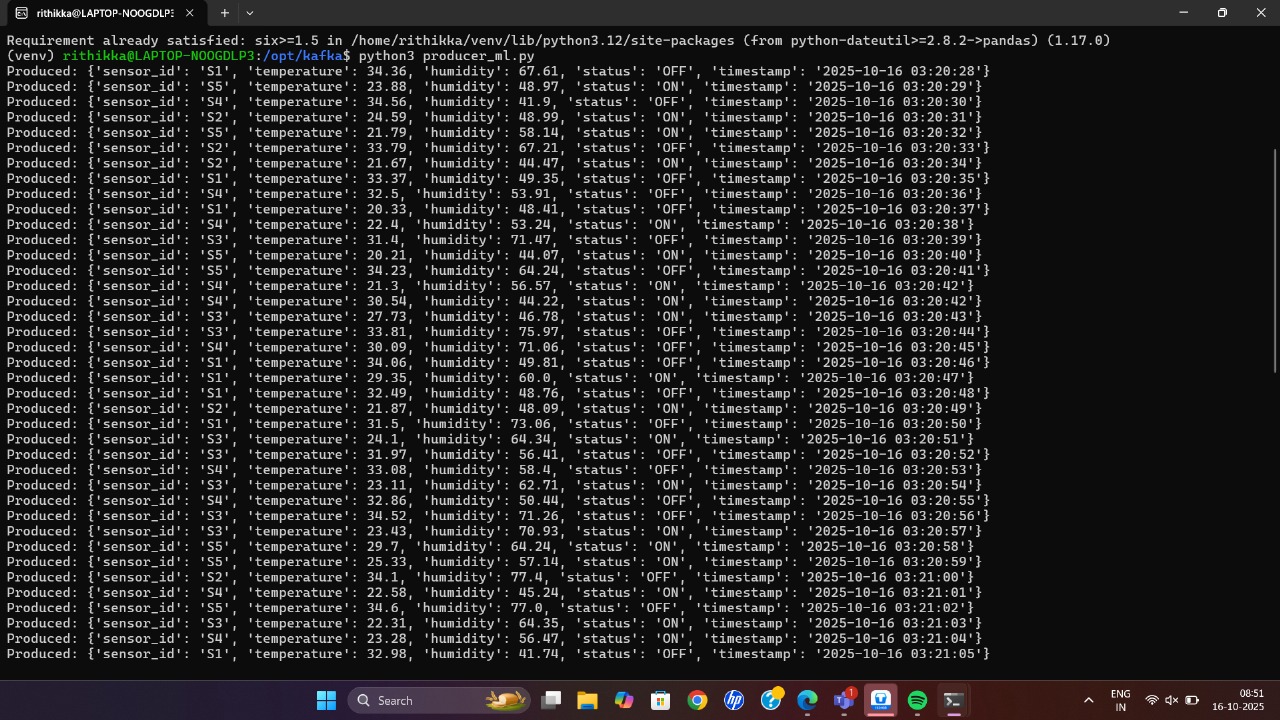


A screenshot of a computer screen

AI-generated content may be incorrect.

A computer screen with a black screen

AI-generated content may be incorrect.



A screenshot of a computer

AI-generated content may be incorrect.

**9. Conclusion**

* A complete real-time data streaming system is successfully implemented using Kafka and Python.
* The system supports real-time analytics, ML-based predictions, and dynamic data visualization in terminal.
* This setup can be extended to larger datasets or multiple topics for advanced streaming analytics.

**3.Incremental Data Processing Report – Sensor Data**

**1. Objective**

The goal of this challenge is to implement an incremental data processing pipeline using Change Data Capture (CDC) techniques. The system tracks changes in sensor data in real-time and updates metrics without recomputing from scratch.

Key Goals:

* Track data changes automatically (insert, update, delete).
* Update computed metrics incrementally.
* Maintain online summaries such as Average Temperature and Average Humidity.

**2. Tools and Technologies**

| **Component** | **Tool / Library** |
| --- | --- |
| Change Data Capture | Simulated via Python event stream |
| Incremental Data Processing | Python Dictionary / List for dataset |
| Metrics | Average Temperature, Average Humidity |
| Persistence | Pickle (optional) |

Optional tools for real deployment:

* Apache Kafka / Kafka Connect – captures CDC events from a database
* Apache Flink – real-time stream processing
* Scikit-learn – incremental model updates for ML tasks

**6.Step-by-Step Instructions**

**Step 1:** Prepare Initial Dataset

dataset = [

{"sensor\_id":"S001","temperature":30.2,"humidity":60},

{"sensor\_id":"S002","temperature":29.5,"humidity":65},

{"sensor\_id":"S003","temperature":28.7,"humidity":70},

]

**Step 2:** Compute Initial Metrics

avg\_temp = sum(d['temperature'] for d in dataset)/len(dataset)

avg\_hum = sum(d['humidity'] for d in dataset)/len(dataset)

print(avg\_temp, avg\_hum)

**Step 3:** Apply CDC Events

# Example: Insert new sensor

dataset.append({"sensor\_id":"S004","temperature":31.1,"humidity":59})

# Example: Update sensor

for d in dataset:

if d["sensor\_id"]=="S001": d["temperature"]=32.0

# Example: Delete sensor

dataset = [d for d in dataset if d["sensor\_id"]!="S003"]

**Step 4:** Compute Updated Metrics

avg\_temp = sum(d['temperature'] for d in dataset)/len(dataset)

avg\_hum = sum(d['humidity'] for d in dataset)/len(dataset)

print(avg\_temp, avg\_hum)



A screen shot of a computer screen

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.

A black screen with a black background

AI-generated content may be incorrect.

**Conclusion**

* Incremental data processing updates models using only new or changed data.
* Change Data Capture (CDC) helps capture real-time changes efficiently.
* Reduces overall processing time and resource usage.
* Enables near real-time analytics and insights.
* Ideal for dynamic and continuously evolving datasets.

**4.In-Memory Data Processing Report**

**1. Objective**

The goal of this task was to implement an in-memory data processing system to efficiently analyze sensor data. The focus was on:

* Performing analytics on data stored in memory to optimize processing time.
* Handling real-time updates to the dataset using insert, update, and delete operations.
* Computing key metrics such as average temperature and humidity.
* Demonstrating performance improvements through in-memory processing.

**3.Step-by-Step Instructions**

**Step 1:** Prepare Dataset in Memory

dataset = [

{"sensor":"S1","temp":25.3,"hum":60.2},

{"sensor":"S2","temp":28.7,"hum":58.7},

{"sensor":"S3","temp":29.8,"hum":59.6}

]

**Step 2:** Compute Metrics

avg\_temp = sum(d['temp'] for d in dataset)/len(dataset)

avg\_hum = sum(d['hum'] for d in dataset)/len(dataset)

print(avg\_temp, avg\_hum)

**Step 3:** Update Data Incrementally

# Insert new reading

dataset.append({"sensor":"S4","temp":27.5,"hum":61.0})

# Update existing

for d in dataset:

if d["sensor"]=="S2": d["temp"]=29.0

**Step 4:** Recompute Metrics

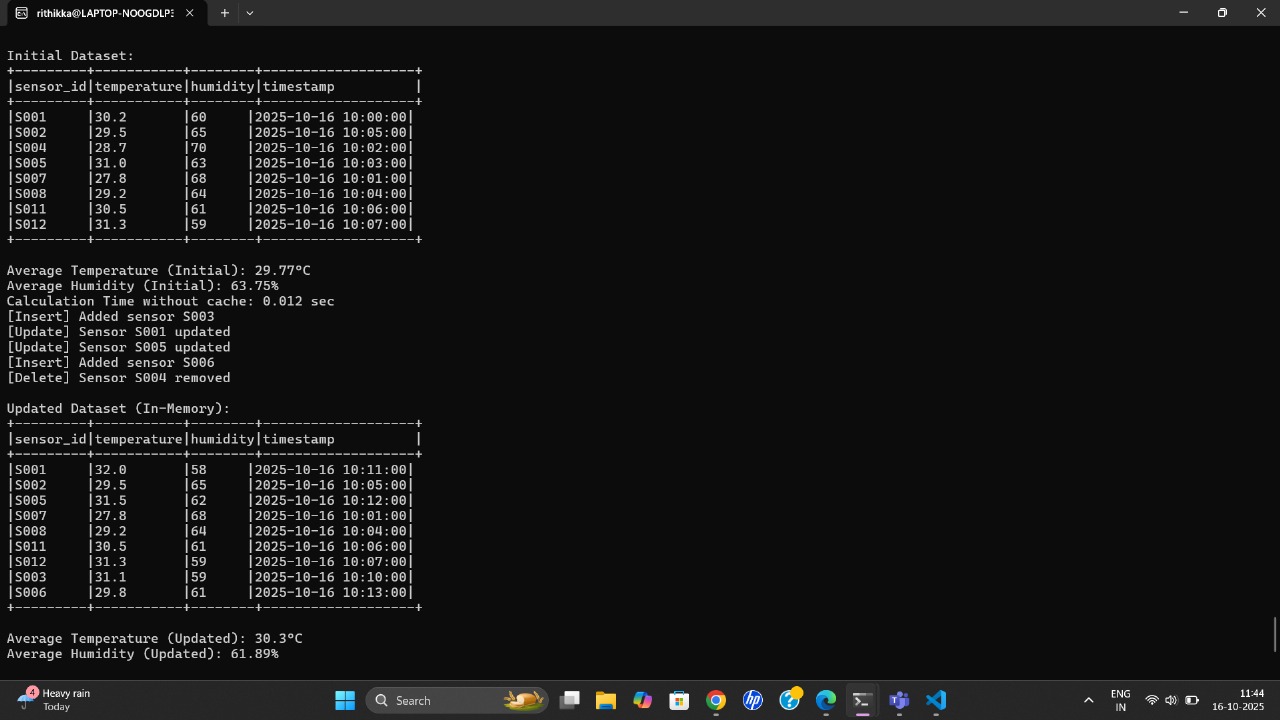
avg\_temp = sum(d['temp'] for d in dataset)/len(dataset)

avg\_hum = sum(d['hum'] for d in dataset)/len(dataset)

print(avg\_temp, avg\_hum)

**Step 5:** Observe Performance

* All calculations happen in memory, no disk I/O → faster processing.



A computer screen with white text

AI-generated content may be incorrect.

**5. Conclusion**

* Successfully implemented an in-memory data processing workflow that simulates real-time updates on sensor data.
* Achievements:
  + Efficiently applied insert, update, and delete operations in memory.
  + Calculated analytics metrics before and after updates in real-time.
  + Demonstrated performance improvements through in-memory caching.
  + Maintained a summary of data changes for monitoring purposes.

**Final Conclusion:**

In this report, i performed comprehensive data preprocessing on a raw IoT sensor dataset using Apache Spark (PySpark) in a Linux environment. The key steps included handling missing values, resolving data type inconsistencies, removing duplicates, and normalizing the data.

Through these preprocessing steps, the dataset was transformed into a clean, consistent, and structured form suitable for accurate analysis and modeling. This process not only improves the quality of insights derived from the data but also enhances the efficiency and reliability of downstream data processing tasks.

Overall, effective data preprocessing is a crucial step in any data-driven project, ensuring that the analysis, machine learning models, and real-time applications can deliver meaningful and trustworthy results. The experience gained through this exercise reinforces the importance of data cleaning, validation, and standardization in practical scenarios involving large-scale IoT datasets.