coding_challenge_cleaner

May 23, 2025

[ ]:
```

```

[ ]:
```
import psycopg2
import pandas as pd
from sqlalchemy import create_engine
from pathlib import Path
import json
import matplotlib.pyplot as plt
import telemetry_pb2
import time
import zmq
```

[ ]:
```
host = 'localhost'
#standard port for postgresql
port = '5432'
#name of my data base
database = 'coding_challenge'
user = 'postgres'
#can't change my password... awk
password = 'Jasper2020Ja!'

# Create connection string
engine = create_engine(f'postgresql+psycopg2://{user}:{password}@{host}:{port}/
 ↪{database}')
```

[ ]:
```
# verifying that i can access the database
# basic practice query
practice_query = 'SELECT * FROM missile_tracks_sensor_1;'

# Load query result into a DataFrame
df = pd.read_sql_query(practice_query, engine)

# Displaying the result
df.head()
```

1 This Section of code is written to validate raw data by:

Identifying null/ missing values
Flagging values that are out of bounds in Latitude, Longitude, and Altitude
Flagging radiometric intensity outliers
and finally saving the summary to a .json for future use

[ ]:
```python
#creating my own function to run through all the step to loop through later
#my broze layer validation function
def bronze_layer(sensor_name, engine, output_dir="reports"):


    # Pulling in the data
    query = f"SELECT * FROM {sensor_name}"
    df = pd.read_sql(query, con=engine)

    #Counting the Nulls
    null_counts = df.isnull().sum().to_dict()
    #Identifying the rows that have nulls-- not the whole rows just the row␣
↪numbers
    null_row_indices = df[df.isnull().any(axis=1)].index.tolist()


    #  Checking the lats, lons and altitude to make sure they are in range.
    #Altitude for missiles can't be below sea level
    out_of_bounds = df[
        (df["latitude"] <= -90) | (df["latitude"] >= 90) |
        (df["longitude"] <= -180) | (df["longitude"] >= 180) |
        (df["altitude"] < 0)
    ]

    #Identifying the rows that are out of bounds-- not the whole rows just the␣
↪row numbers
    out_of_bounds_indices = out_of_bounds.index.tolist()

    # Flagging the Radiometric outliers (IQR)
    # Chatgpt helped with these calculaitons
    #these calculations are the middle 50% of the gaussian distribution
    q1 = df["radiometric_intensity"].quantile(0.25)
    q3 = df["radiometric_intensity"].quantile(0.75)
    iqr = q3 - q1
    lower = q1 - 1.5 * iqr
    upper = q3 + 1.5 * iqr

    outliers = df[
        (df["radiometric_intensity"] < lower) |
        (df["radiometric_intensity"] > upper)
```

```
    ]
    #identifying outlier rows -- not the whole rows just the row numbers
    outlier_indices = outliers.index.tolist()

    # Building my summary dictionary
    #to be saved in the json in this order with these labels
    report = {
        "sensor": sensor_name,
        "row_count": len(df),
        "null_counts": null_counts,
        "null_row_indices": null_row_indices,
        "position_out_of_bounds_count": len(out_of_bounds),
        "position_out_of_bounds_indices": out_of_bounds_indices,
        "radiometric_outlier_count": len(outliers),
        "radiometric_outlier_indices": outlier_indices
    }

    #Saving the summary as JSON
    #and saving it to a reports directory with the sensor name +_validation
    Path(output_dir).mkdir(exist_ok=True)
    out_path = Path(output_dir) / f"{sensor_name}_validation.json"

    with open(out_path, "w") as f:
        json.dump(report, f, indent=2)

    print(f" Report saved: {out_path}")
    return report
```

[ ]:
```
# a for loop to go through each missile track table
#this runs it through the function defined above

for i in range(1, 6):
    bronze_layer(f"missile_tracks_sensor_{i}", engine)
```

This section of code is written to clean and enrich the data by: Interpolating small gaps in data, Excluding larger gaps from the data, Smoothing the radiometric intensity data to reduce noise, and finally saving the cleaned data in a .json file for future use,

2 Trade-offs

Method	Pros	Cons
Interpolation	Fills in the minor gaps	This can hide real anomalies
Exclusion	Avoids bad data	Reduces the amount of data
Smoothing	Smooths noise	Can distort the true nature

2.0.1 Visual Confirmation

See plot below showing radiometric intensity before/after smoothing.

```
[ ]: def silver_layer(sensor_name, engine, gap_threshold=5, smooth_window=10,␣
     ↪output_dir="clean"):

         # Reading in the table of data
         df = pd.read_sql(f"SELECT * FROM {sensor_name}", con=engine)

         #Sorting data by time
         #creating a data frame to log the change (delta) in time (t) of the unix␣
     ↪timestamp
         #using .fillna(0) to fill in the value above the first with 0
         df = df.sort_values("unix_timestamp")
         df["delta_t"] = df["unix_timestamp"].diff().fillna(0)

         #Creating a table/data frame to log the gaps that are greater than 5␣
     ↪seconds .
         df["time_gap_flag"] = df["delta_t"] > 5

         #Interpolating the smaller gaps
         #making a copy of the table to use for interpolation
         interpolated = df.copy()

         # Identifying the small gaps (less then 5 seconds)to be interpolated,
         #the columns that can be interpolated, and then the rows that need to be␣
     ↪interpolated
         interpolated = df.copy()
         columns_to_interp = ["latitude", "longitude", "altitude",␣
     ↪"radiometric_intensity"]
         safe_rows = ~df["time_gap_flag"]

         #Locating the rows within the columns that need to be interoplated and then␣
     ↪interpolating them linearly
         interpolated.loc[safe_rows, columns_to_interp] = df.loc[safe_rows,␣
     ↪columns_to_interp].interpolate(method="linear")

         #Excluding the rows with large gaps
         cleaned = interpolated[~df["time_gap_flag"]].copy().reset_index(drop=True)

         # Smoothing radiometric intensity with a rolling window of 10 data points␣
     ↪(identified in the function parameters)
         cleaned["radiometric_smoothed"] = cleaned["radiometric_intensity"].
     ↪rolling(window=smooth_window, center=True).mean()

         # Ploting the radiometric intensity smooth v original
```

```
    plt.figure(figsize=(12, 5))
    plt.plot(df["unix_timestamp"], df["radiometric_intensity"],␣
↪label="Original", alpha=0.4)
    plt.plot(cleaned["unix_timestamp"], cleaned["radiometric_smoothed"],␣
↪label="Smoothed", linewidth=2)
    plt.title("Radiometric Intensity: Original vs. Smooth ")
    plt.xlabel("Time")
    plt.ylabel("Intensity")
    plt.legend()
    plt.grid(True)
    plt.show()


    #  Saving to a .json
    Path(output_dir).mkdir(exist_ok=True)
    out_path = Path(output_dir) / f"{sensor_name}_cleaned.json"
    #saving the output in a more table layout or readablity
    cleaned.to_json(f"clean/{sensor_name}_cleaned.json", orient="records",␣
↪indent=2)

    # Adding the cleaned data to the Database in pgadmin4
    table_name = "cleaned_telemetry"
    try:
        cleaned.to_sql(table_name, engine, if_exists="append", index=False)
        print(f" {sensor_name} saved to PostgreSQL table '{table_name}'")
    except Exception as e:
        print(f"Failed to write {sensor_name} to PostgreSQL: {e}")

    print(f"Cleaned JSON saved: {out_path}")
    return cleaned
```

[ ]:
```
for i in range(1, 6):
    silver_layer(f"missile_tracks_sensor_{i}", engine)
```

3 This section of code was written to serialize and stream data:

```
Serializes telemetry data row-by-row using Protobuf serialization
Then streams the data using ZeroMQ streaming.
    - If there is an error it will skip that row, but this could result in loss of data
    - If the connection fails that error will be cought and printed

I have set up a seperate notebook to simulate streaming between two sources.
```

[ ]:
```
#Creating my own function to serialize and stream data
```

```python
def serialize_stream(cleaned_df, zmq_port=5555, max_delay=1.0,␣
↪log_name="stream"):

    #setting up ZeroMQ
    #I am not familiar with ZeroMQ so chatgpt created this
    try:
        context = zmq.Context()
        socket = context.socket(zmq.PUSH)
        socket.bind(f"tcp://127.0.0.1:{zmq_port}")
        #printing a start message
        print(f" Streaming to ZeroMQ socket on port {zmq_port}...\n")

    except Exception as e:
        print(f"[ERROR] Could not connect to port {zmq_port}: {e}")
        return

    #Sorting data by time
    cleaned_df = cleaned_df.sort_values("unix_timestamp").reset_index(drop=True)
    n = len(cleaned_df)

    start_time = time.time()
    #loop throw each row of data
    for i in range(n):
        row = cleaned_df.iloc[i]

        # Create Protobuf message
        #I've never used protobufs so chatgpt created this
        msg = telemetry_pb2.TelemetryData(
            unix_timestamp=int(row["unix_timestamp"]),
            latitude=float(row["latitude"]),
            longitude=float(row["longitude"]),
            altitude=float(row["altitude"]),
            radiometric_intensity=float(row["radiometric_intensity"]),
            radiometric_smoothed=float(row["radiometric_smoothed"])
        )

        # Turning the message defined above into binary
        try:
            binary_data = msg.SerializeToString()
            socket.send(binary_data)
            print(f"[{i}] Sent message at timestamp {row['unix_timestamp']}")
        except Exception as e:
            print(f"[ERROR] Failed to serialize or send message at index {i}:␣
↪{e}")
            continue  # Skip this row and keep streaming

        #simulates the sending of data based on the times
```

```
        if i < len(cleaned_df) - 1:
            delta = cleaned_df.iloc[i + 1]["unix_timestamp"] -␣
↪row["unix_timestamp"]
            time.sleep(min(delta, max_delay))


        end_time = time.time()
        duration = end_time - start_time
        throughput = n / duration
        latency = duration / n


    print("\n Stream complete.")
    print(f"Throughput: {throughput:.2f} messages/sec")
    print(f" Average latency per message: {latency:.6f} sec")


    socket.close()
    context.term()
```

[ ]:

```
for i in range(1, 6):
    sensor = f"missile_tracks_sensor_{i}"
    cleaned = pd.read_json(f"clean/{sensor}_cleaned.json")
    serialize_stream(cleaned, zmq_port=5555 + i, log_name=sensor)
```

4 Efficiency and Performance Metrics

Testing Environment:

```
Machine: MacBook Pro 3.1 GHz Dual-Core Intel Core i5
Memory: 8 GB 2133 MHz LPDDR3
Python: 3, running in Jupyter Notebook
Streaming method: ZeroMQ
Serialization: Google Protobuf
```

Used a serialize_stream() function to stream telemetry data from one cleaned sensor file:

```
Timed 1 sensor to compute: ~12 min
Total stream time:
Average throughput (messages per second):242.60
Average latency per message: 0.004122 sec
```

Optimizations Used:

```
Protobuf for fast serialization
ZeroMQ for message transport
No intermediate files used in streaming
```

PostgreSQL writes using to_sql(..., if_exists="append")
Streaming runs in a loop with no redundant computation

[3]:
```
#This section of code is almost the same as the above but is being used to test␣
↪how the above can handle bust data.
#for ease I made this chunk to be run with out any of the other code


import psycopg2
import pandas as pd
from sqlalchemy import create_engine
from pathlib import Path
import json
import matplotlib.pyplot as plt
import telemetry_pb2
import time
import zmq


def simulate_burst(cleaned_df, repeat=10, zmq_port=5555):


    context = zmq.Context()
    socket = context.socket(zmq.PUSH)
    socket.bind(f"tcp://127.0.0.1:{zmq_port}")

    total_messages = len(cleaned_df) * repeat
    start = time.time()

    for r in range(repeat):
        for i, row in cleaned_df.iterrows():
            try:
                msg = telemetry_pb2.TelemetryData(
                    unix_timestamp=int(row["unix_timestamp"]),
                    latitude=float(row["latitude"]),
                    longitude=float(row["longitude"]),
                    altitude=float(row["altitude"]),
                    radiometric_intensity=float(row["radiometric_intensity"]),
                    radiometric_smoothed=float(row["radiometric_smoothed"])
                )
                socket.send(msg.SerializeToString())
            except Exception as e:
                print(f"[ERROR] Row {i}: {e}")
                continue

    end = time.time()
    duration = end - start
```

```
    throughput = total_messages / duration
    latency = duration / total_messages

    print(f"Simulated {total_messages} messages in {duration:.2f} seconds")
    print(f"Throughput: {throughput:.2f} msgs/sec")
    print(f"Average latency: {latency:.6f} sec")

    socket.close()
    context.term()
```

[4]:
```
df = pd.read_json("clean/missile_tracks_sensor_1_cleaned.json")
simulate_burst(df, repeat=20, zmq_port=5555)
```

```
Simulated 3536240 messages in 680.78 seconds
Throughput: 5194.39 msgs/sec
Average latency: 0.000193 sec
```

4.1 Scalability Demonstration: Bust scenario

To test whether or not this script can handle burst data, a for loop is used to quickly run and send data repeatedly, sending thousands of messages to a ZeroMQ stream in memory.

Testing Environment:

```
simulate_burst() function sent 3,000,000 messages in a loop

Machine: MacBook Pro 3.1 GHz Dual-Core Intel Core i5
Memory: 8 GB 2133 MHz LPDDR3
Python: 3, running in Jupyter Notebook
Streaming method: ZeroMQ
Serialization: Google Protobuf
```

Results:

- Total messages: 3,536,240
- Duration: 680.78 seconds
- Throughput:~ 5194.39 messages/second
- Latency: ~0.000193 seconds per message

These results confirm that the pipeline can sustain high message volumes on very modest hardware, and would do even better with multiprocessing or distributed receivers.

[ ]:
```
```