THINGS SKIPPED

Saturday, March 30, 2024 7:27 PM

why?

adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0) # How the initializing works

capture_analysis_output function

Automation

Wednesday, 27 March, 2024 2:21 PM

How to run a Python file in powershell

python CodeforAutomation.py

Log_file.xlsx file

- **id:** A numerical identifier, possibly for the scooter or the log entry.
- **timestamp & localtime:** The timestamp and local time of the log entry.
- **percentage_of_can_ids:** Possibly a measure of the amount of Controller Area Network (CAN) identifiers captured or processed.
- **MotorSpeed_340920578:** The speed of the scooter's motor.
- **PackCurr_6:** Current being drawn from the battery pack.
- **PackVol_6:** Voltage of the battery pack.
- **Throttle_408094978:** Throttle position or engagement level.
- **SOC_8:** State of Charge of the battery, a measure of the remaining battery life as a percentage.
- **DchgFetStatus_9, ChgFetStatus_9:** Status indicators for discharge and charge Field-Effect Transistors (FETs) in the battery management system.
- Various other parameters related to the scooter's battery management system, motor control, and environmental conditions like temperature.

The dataset contains a wide variety of metrics that can be incredibly useful for monitoring the scooter's performance, diagnosing issues, and understanding usage patterns. Here are a few steps on how to proceed with analyzing this data and building an automation dashboard:

km_file.xlsx file

- **id:** A numerical identifier, likely the same as in the first dataset, which could be used to join or correlate the two datasets.
- **timestamp & localtime:** These columns provide the timestamp and the local time when each data point was recorded, similar to the first dataset.
- **latitude & longitude:** The geographic coordinates of the scooter at each recorded point, which can be used to track its movements.
- **altitude:** The altitude in meters at each location point, which might be useful for understanding the scooter's performance in different elevations.

Wednesday, 27 March, 2024 4:24 PM

chatGPT prompt https://chat.openai.com/share/aa20e01c-0259-4167-8830-2855effc1008

Learn Execution flow of the code

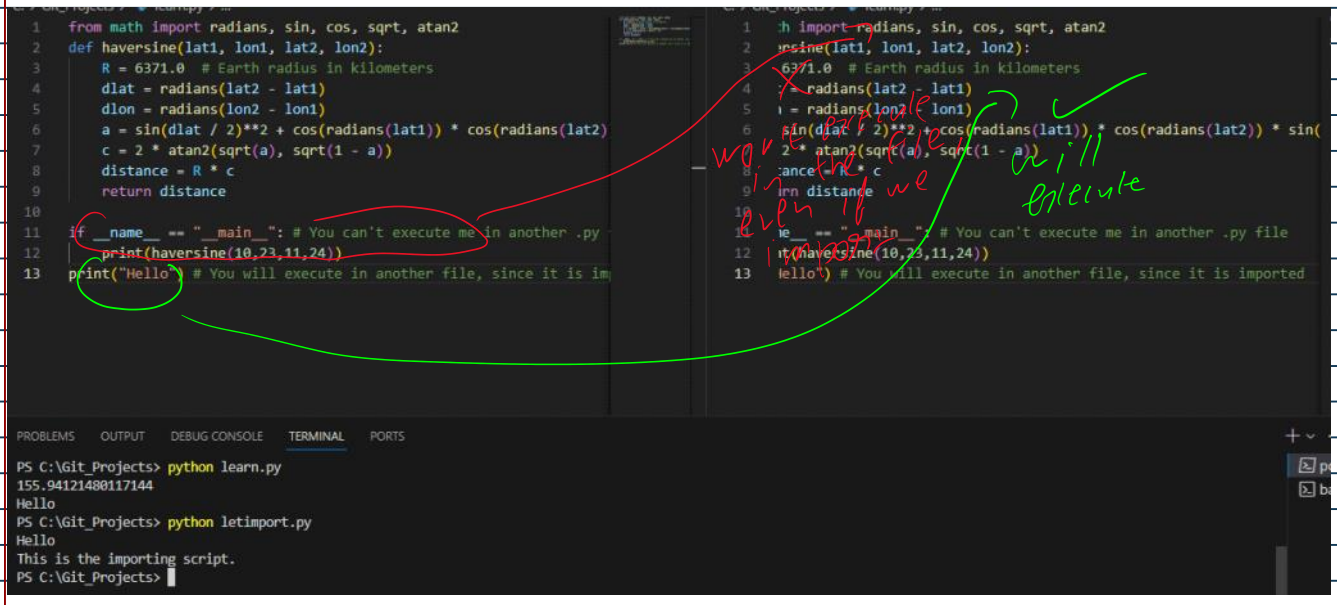
Execution Flow

1. The script initializes paths and lists to hold the DataFrames created from the CSV files.
2. It defines utility functions for distance calculations, data adjustments, plotting, and analysis.
3. It iterates over a specified directory structure, identifying and processing the relevant CSV files.
4. It performs a series of analyses on the data, including energy consumption analysis and distance calculations.
5. It generates graphical representations of the data and summarizes the findings in both PowerPoint and Word document formats.

Part of the code to learn

Data adjustments

Python FACT



The 691 line code- created ppt

Thursday, March 28, 2024 10:12 PM

```
import pandas as pd
from math import radians, sin, cos, sqrt, atan2

import sys
import io

import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

from PIL import Image
import os
import matplotlib.dates as mdates
import mplcursors  # Import mplcursors
from contextlib import redirect_stdout
from pptx.util import Inches, Pt  # Correcting the import statement

from pptx import Presentation
from pptx.util import Inches
from docx import Document
from docx.shared import Inches

# Path to the folder containing the CSV files
path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER"

# List to store DataFrames from each CSV file
dfs = []
def haversine(lat1, lon1, lat2, lon2):
    """
    Calculate the great-circle distance between two points
    on the Earth's surface using the Haversine formula.
    """
    # Convert latitude and longitude from degrees to radians
    lat1 = radians(lat1)
    lon1 = radians(lon1)
    lat2 = radians(lat2)
    lon2 = radians(lon2)

    # Radius of the Earth in kilometers
    R = 6371.0

    # Calculate differences in latitude and longitude
    dlat = lat2 - lat1
    dlon = lon2 - lon1

    # Calculate Haversine formula
    a = sin(dlat / 2)**2 + cos(lat1) * cos(lat2) * sin(dlon / 2)**2
    c = 2 * atan2(sqrt(a), sqrt(1 - a))
    distance = R * c

    return distance

# Define a function to set current to zero if RPM is zero for 10 or more consecutive points
def adjust_current(row):
    adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0)
    if row['MotorSpeed_340920578'] == 0:
        adjust_current.zero_count += 1
    else:
        adjust_current.zero_count = 0

    if adjust_current.zero_count >= 10:
        return 0
    else:
        return row['PackCurr_6']
def plot_ghps(log_file):

    data = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21\log_file.csv")

    # Apply the adjustment function to the DataFrame


    data['localtime'] = pd.to_datetime(data['localtime'], format='%d/%m/%Y %H:%M:%S.%f', dayfirst=True)
    data.set_index('localtime', inplace=True)
    data['PackCurr_6'] = data.apply(adjust_current, axis=1)




    # Create a figure and axes for plotting
    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Plot 'PackCurr_6' on primary y-axis
    line1, = ax1.plot(data.index, -data['PackCurr_6'], color='blue', label='PackCurr_6')
    ax1.set_ylabel('Pack Current (A)', color='blue')
    ax1.yaxis.set_label_coords(-0.1, 0.7)  # Adjust label position

    # Create secondary y-axis for 'MotorSpeed_340920578' (RPM)
    ax2 = ax1.twinx()
    line2, = ax2.plot(data.index, data['MotorSpeed_340920578'], color='green', label='Motor Speed')
    ax2.set_ylabel('Motor Speed (RPM)', color='green')

    # Add 'AC_Current_340920579' to primary y-axis
    line3, = ax1.plot(data.index, data['AC_Current_340920579'], color='red', label='AC Current')

    # Add 'AC_Voltage_340920580' scaled to 10x to the left side y-axis
    line4, = ax1.plot(data.index, data['AC_Voltage_340920580'] * 10, color='orange', label='AC Voltage (x10)')

    # Add 'Throttle_408094978' to the left side y-axis
    line5, = ax1.plot(data.index, data['Throttle_408094978'], color='lightgray', label='Throttle (%)')
```

```
    # Hide the y-axis label for 'AC_Current_340920579'
    ax1.get_yaxis().get_label().set_visible(False)

    # Set x-axis label and legend
    ax1.set_xlabel('Local Time')
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    # Add a title to the plot
    plt.title('Battery Pack, Motor Data, and Throttle')

    # Format x-axis ticks as hours:minutes:seconds
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))

    # Set grid lines lighter
    ax1.grid(True, linestyle=':', linewidth=0.5, color='gray')
    ax2.grid(True, linestyle=':', linewidth=0.5, color='gray')

    # Enable cursor to display values on graphs
    mplcursors.cursor([line1, line2, line3, line4, line5])

    # Save the plot as an image or display it
    plt.tight_layout()  # Adjust layout to prevent clipping of labels
    plt.savefig('graph.png')  # Save the plot as an image
    plt.show()
def analysis_Energy(log_file, km_file):
    dayfirst=True

    data = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21\log_file.csv")

    data_KM = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21\km_file.csv")

    total_duration = 0
    total_distance = 0
    Wh_km = 0
    SOC_consumed = 0

    # Check if 'localtime' column exists in data DataFrame
    if 'localtime' not in data.columns:
        print("Error: 'localtime' column not found in the DataFrame.")
        return None, None, None, None




    # Drop rows with missing values in 'SOCAh_8' column
    data.dropna(subset=['SOCAh_8'], inplace=True)

    # Convert the 'localtime' column to datetime s
    data['localtime'] = pd.to_datetime(data['localtime'], format='%d/%m/%Y %H:%M:%S.%f',dayfirst=True)

    # Calculate the start time and end time
    start_time = data['localtime'].min()
    end_time = data['localtime'].max()

    # Calculate the total time taken for the ride
    total_duration = end_time - start_time

    total_hours = total_duration.seconds // 3600
    total_minutes = (total_duration.seconds % 3600) // 60

    print(f"Total time taken for the ride: {int(total_hours):02d}:{int(total_minutes):02d}")

    # Calculate the time difference between consecutive rows
    data['Time_Diff'] = data['localtime'].diff().dt.total_seconds().fillna(0)

    # Set the 'localtime' column as the index
    data.set_index('localtime', inplace=True)

    # Resample the data to have one-second intervals and fill missing values with previous ones
    data_resampled = data.resample('s').ffill()

    # Calculate the time difference between consecutive rows
    data_resampled['Time_Diff'] = data_resampled.index.to_series().diff().dt.total_seconds().fillna(0)

    # Calculate the actual Ampere-hours (Ah) using the trapezoidal rule for numerical integration
    actual_ah = abs((data_resampled['PackCurr_6'] * data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours
    print("Actual Ampere-hours (Ah): {:.2f}".format(actual_ah))

    # Calculate the actual Watt-hours (Wh) using the trapezoidal rule for numerical integration
    watt_h = abs((data_resampled['PackCurr_6'] * data_resampled['PackVol_6'] * data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours
    print("Actual Watt-hours (Wh):{:.2f}" .format(watt_h))

    ###########  starting and ending ah
    starting_soc = data['SOCAh_8'].iloc[0]
    ending_soc = data['SOCAh_8'].iloc[-1]

    print("Starting SoC (Ah):{:.2f}".format (starting_soc))
    print("Ending SoC (Ah):{:.2f}".format  (ending_soc))

    ##################### KM ----------------------
    # Convert the 'localtime' column to datetime format
    data_KM['localtime'] = pd.to_datetime(data_KM['localtime'], format='%d/%m/%Y %H:%M:%S.%f', dayfirst=True)

    # Initialize total distance covered
    total_distance = 0

    # Iterate over rows to compute distance covered between consecutive points
    for i in range(1, len(data_KM)):
        # Get latitude and longitude of consecutive points
        lat1, lon1 = data_KM.loc[i - 1, 'latitude'], data_KM.loc[i - 1, 'longitude']
        lat2, lon2 = data_KM.loc[i, 'latitude'], data_KM.loc[i, 'longitude']

        # Calculate distance between consecutive points
        distance = haversine(lat1, lon1, lat2, lon2)

        # Add distance to total distance covered
```

```
    total_distance += distance

  print("Total distance covered (in kilometers):{:.2f}".format(total_distance))

  ##############  Wh/Km
  Wh_km = abs(watt_h / total_distance)
  print("WH/KM:{:.2f}". format (watt_h / total_distance))

  # Assuming 'data' is your DataFrame with 'SOC_8' column
  initial_soc = data['SOC_8'].iloc[0]  # Initial SOC percentage
  final_soc = data['SOC_8'].iloc[-1]   # Final SOC percentage

  # Calculate total SOC consumed
  total_soc_consumed =  abs(final_soc - initial_soc)

  print ("Total SOC consumed:{:.2f}".format (total_soc_consumed),"%")


  # Check if the mode remains constant or changes
  mode_values = data['Mode_Ack_408094978'].unique()

  if len(mode_values) == 1:
    # Mode remains constant throughout the log file
    mode = mode_values[0]
    if mode == 3:
      print("Mode is Custom mode.")
    elif mode == 2:
      print("Mode is Sports mode.")
    elif mode == 1:
      print("Mode is Eco mode.")
  else:
    # Mode changes throughout the log file
    mode_counts = data['Mode_Ack_408094978'].value_counts(normalize=True) * 100
    for mode, percentage in mode_counts.items():
      if mode == 3:
        print(f"Custom mode: {percentage:.2f}%")
      elif mode == 2:
        print(f"Sports mode: {percentage:.2f}%")
      elif mode == 1:
        print(f"Eco mode: {percentage:.2f}%")
 ##preak current##
  # Calculate power using PackCurr_6 and PackVol_6
  data_resampled['Power'] = data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

  # Find the peak power
  peak_power = data_resampled['Power'].min()
  print("Peak Power:", peak_power)

  # Calculate the average power
  average_power = data_resampled['Power'].mean()
  print("Average Power:", average_power)



  # Calculate energy regenerated (in watt-hours)
  energy_regenerated = ((data_resampled[data_resampled['Power'] > 0]['Power']*data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours
#########################################################################

  # Calculate total energy consumed (in watt-hours)
  total_energy_consumed =  ((data_resampled[data_resampled['Power'] < 0]['Power']*data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours
#########################################################################

  print("total",total_energy_consumed)
  print("total energy regenerated",energy_regenerated)



  # Calculate regenerative effectiveness as a percentage
  if total_energy_consumed != 0:
   regenerative_effectiveness = (energy_regenerated / total_energy_consumed) * 100
   print("Regenerative Effectiveness (%):", regenerative_effectiveness)
  else:
   print("Total energy consumed is 0, cannot calculate regenerative effectiveness.")

  # Calculate idling time percentage (RPM was zero for more than 5 seconds)
  idling_time = (data['MotorSpeed_340920578'] == 0).sum()
  idling_percentage = (idling_time / len(data)) * 100
  print("Idling time percentage:", idling_percentage)

  # Calculate time spent in specific speed ranges
  speed_ranges = [(0, 10), (10, 20), (20, 30), (30, 40), (40, 50),(50, 60),(60,70)]
  speed_range_percentages = {}

  for range_ in speed_ranges:
    speed_range_time = ((data['MotorSpeed_340920578'] * 0.016 >= range_[0]) & (data['MotorSpeed_340920578'] * 0.016 < range_[1])).sum()
    speed_range_percentage = (speed_range_time / len(data)) * 100
    speed_range_percentages[f"Time spent in {range_[0]}-{range_[1]} km/h"] = speed_range_percentage
    print(f"Time spent in {range_[0]}-{range_[1]} km/h: {speed_range_percentage:.2f}%")


#########################################################################################################################################################################################
######################

      # Calculate power using PackCurr_6 and PackVol_6
  data_resampled['Power'] = -data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

  # Find the peak power
  peak_power = data_resampled['Power'].max()

  # Get the maximum cell voltage
  max_cell_voltage = data_resampled['MaxCellVol_5'].max()

  # Find the index where the maximum voltage occurs
  max_index = data_resampled['MaxCellVol_5'].idxmax()
```

```
# Retrieve the corresponding cell ID using the index
max_cell_id = data_resampled['MaxVoltId_5'].loc[max_index]

# Get the minimum cell voltage
min_cell_voltage = data_resampled['MinCellVol_5'].min()

# Find the index where the minimum voltage occurs
min_index = data_resampled['MinCellVol_5'].idxmin()

# Retrieve the corresponding cell ID using the index
min_cell_id = data_resampled['MinVoltId_5'].loc[min_index]

voltage_difference = max_cell_voltage - min_cell_voltage

# Get the maximum temperature
max_temp = data_resampled['MaxTemp_7'].max()

# Find the index where the maximum temperature occurs
max_temp_index = data_resampled['MaxTemp_7'].idxmax()

# Retrieve the corresponding temperature ID using the index
max_temp_id = data_resampled['MaxTempId_7'].loc[max_temp_index]

# Get the minimum temperature
min_temp = data_resampled['MinTemp_7'].min()

# Find the index where the minimum temperature occurs
min_temp_index = data_resampled['MinTemp_7'].idxmin()

# Retrieve the corresponding temperature ID using the index
min_temp_id = data_resampled['MinTempId_7'].loc[min_temp_index]

# Calculate the difference in temperature
temp_difference = max_temp - min_temp

# Get the maximum temperature of FetTemp_8
max_fet_temp = data_resampled['FetTemp_8'].max()

# Get the maximum temperature of AfeTemp_12
max_afe_temp = data_resampled['AfeTemp_12'].max()

# Get the maximum temperature of PcbTemp_12
max_pcb_temp = data_resampled['PcbTemp_12'].max()

# Get the maximum temperature of MCU_Temperature_408094979
max_mcu_temp = data_resampled['MCU_Temperature_408094979'].max()

# Check for abnormal motor temperature at high RPMs
max_motor_temp = data_resampled['Motor_Temperature_408094979'].max()

# Check for abnormal motor temperature at high RPMs for at least 15 seconds
abnormal_motor_temp = (data_resampled['Motor_Temperature_408094979'] < 10) & (data_resampled['MotorSpeed_340920578'] > 3500)
abnormal_motor_temp_mask = abnormal_motor_temp.astype(int).groupby(abnormal_motor_temp.ne(abnormal_motor_temp.shift()).cumsum()).cumsum()

# Check if abnormal condition persists for at least 15 seconds
abnormal_motor_temp_detected = (abnormal_motor_temp_mask >= 120).any()

##########################################################################################################################################################################
##########



# Add these variables and logic to ppt_data
ppt_data = {
    "Total time taken for the ride": total_duration,
    "Actual Ampere-hours (Ah)": actual_ah,
    "Actual Watt-hours (Wh)": watt_h,
    "Starting SoC (Ah)": starting_soc,
    "Ending SoC (Ah)": ending_soc,
    "Total distance covered (in kilometers)": total_distance,
    "WH/KM": watt_h / total_distance,
    "Total SOC consumed": total_soc_consumed,
    "Mode": "",
    "Peak Power": peak_power,
    "Average Power": average_power,
    "Total Energy Regenerated": energy_regenerated,
    "Regenerative Effectiveness": regenerative_effectiveness,
    "Lowest Cell Voltage": min_cell_voltage,
    "Highest Cell Voltage": max_cell_voltage,
    "Difference in Cell Voltage": voltage_difference,
    "Minimum Temperature": min_temp,
    "Maximum Temperature": max_temp,
    "Difference in Temperature": temp_difference,
    "Maximum Fet Temperature": max_fet_temp,
    "Maximum Afe Temperature": max_afe_temp,
    "Maximum PCB Temperature": max_pcb_temp,
    "Maximum MCU Temperature": max_mcu_temp,
    "Maximum Motor Temperature": max_motor_temp,
    "Abnormal Motor Temperature Detected": abnormal_motor_temp_detected
}
mode_values = data_resampled['Mode_Ack_408094978'].unique()
if len(mode_values) == 1:
    mode = mode_values[0]
    if mode == 3:
        ppt_data["Mode"] = ["Custom mode"]
    elif mode == 2:
        ppt_data["Mode"] = ["Sports mode"]
    elif mode == 1:
        ppt_data["Mode"] = ["Eco mode"]
else:
    # Mode changes throughout the log file
    mode_counts = data_resampled['Mode_Ack_408094978'].value_counts(normalize=True) * 100
    ppt_data["Mode"] = []  # Initialize list to store modes
    for mode, percentage in mode_counts.items():
        if mode == 3:
            ppt_data["Mode"].append(f"Custom mode: {percentage:.2f}%")
        elif mode == 2:
```

```
            ppt_data["Mode"].append(f"Sports mode: {percentage:.2f}%")
        elif mode == 1:
            ppt_data["Mode"].append(f"Eco mode: {percentage:.2f}%")

    print("ppt",len(ppt_data))

    # Add calculated parameters to ppt_data
    ppt_data["Idling time percentage"] = idling_percentage
    ppt_data.update(speed_range_percentages)
################################################################################################### recent data

  # Calculate power using PackCurr_6 and PackVol_6
  data_resampled['Power'] = -data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

  # Find the peak power
  peak_power = data_resampled['Power'].max()
  print("Peak Power:", peak_power)

  # Get the maximum cell voltage
  max_cell_voltage = data_resampled['MaxCellVol_5'].max()

  # Find the index where the maximum voltage occurs
  max_index = data_resampled['MaxCellVol_5'].idxmax()

  # Retrieve the corresponding cell ID using the index
  max_cell_id = data_resampled['MaxVoltId_5'].loc[max_index]

    # Get the minimum cell voltage
  min_cell_voltage = data_resampled['MinCellVol_5'].min()

  # Find the index where the minimum voltage occurs
  min_index = data_resampled['MinCellVol_5'].idxmin()

  # Retrieve the corresponding cell ID using the index
  min_cell_id = data_resampled['MinVoltId_5'].loc[min_index]

  voltage_difference = max_cell_voltage - min_cell_voltage


  print("Lowest Cell Voltage:", min_cell_voltage, "V, Cell ID:", min_cell_id)
  print("Highest Cell Voltage:", max_cell_voltage, "V, Cell ID:", max_cell_id)
  print("Difference in Cell Voltage:", voltage_difference, "V")

  # Get the maximum temperature
  max_temp = data_resampled['MaxTemp_7'].max()

  # Find the index where the maximum temperature occurs
  max_temp_index = data_resampled['MaxTemp_7'].idxmax()

  # Retrieve the corresponding temperature ID using the index
  max_temp_id = data_resampled['MaxTempId_7'].loc[max_temp_index]


    # Get the minimum temperature
  min_temp = data_resampled['MinTemp_7'].min()

  # Find the index where the minimum temperature occurs
  min_temp_index = data_resampled['MinTemp_7'].idxmin()

  # Retrieve the corresponding temperature ID using the index
  min_temp_id = data_resampled['MinTempId_7'].loc[min_temp_index]
  # Calculate the difference in temperature
  temp_difference = max_temp - min_temp

  # Print the information
  print("Maximum Temperature:", max_temp, "C, Temperature ID:", max_temp_id)
  print("Minimum Temperature:", min_temp, "C, Temperature ID:", min_temp_id)
  print("Difference in Temperature:", temp_difference, "C")

  # Get the maximum temperature of FetTemp_8
  max_fet_temp = data_resampled['FetTemp_8'].max()
  print("Maximum Fet Temperature:", max_fet_temp, "C")

  # Get the maximum temperature of AfeTemp_12
  max_afe_temp = data_resampled['AfeTemp_12'].max()
  print("Maximum Afe Temperature:", max_afe_temp, "C")

  # Get the maximum temperature of PcbTemp_12
  max_pcb_temp = data_resampled['PcbTemp_12'].max()
  print("Maximum PCB Temperature:", max_pcb_temp, "C")

  # Get the maximum temperature of MCU_Temperature_408094979
  max_mcu_temp = data_resampled['MCU_Temperature_408094979'].max()
  print("Maximum MCU Temperature:", max_mcu_temp, "C")

    # Check for abnormal motor temperature at high RPMs
  max_motor_temp = data_resampled['Motor_Temperature_408094979'].max()


  print("Maximum Motor Temperature:", max_motor_temp, "C")


  # Check for abnormal motor temperature at high RPMs for at least 15 seconds
  abnormal_motor_temp = (data_resampled['Motor_Temperature_408094979'] < 10) & (data_resampled['MotorSpeed_340920578'] > 3500)

  # Convert to a binary mask indicating consecutive occurrences
  abnormal_motor_temp_mask = abnormal_motor_temp.astype(int).groupby(abnormal_motor_temp.ne(abnormal_motor_temp.shift()).cumsum()).cumsum()

  # Check if abnormal condition persists for at least 15 seconds
  if (abnormal_motor_temp_mask >= 15).any():
    print("Abnormal motor temperature detected: NTC has issues - very low temperature at high RPMs")

#########################################################################################################################################################################
####

    return total_duration, total_distance, Wh_km, total_soc_consumed,ppt_data
```

```
folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21"
def capture_analysis_output(log_file, km_file, folder_path):
    #try:
        # Capture print statements
        analysis_output = io.StringIO()
        output_file = "analysis_results.docx"

        with redirect_stdout(analysis_output):
            total_duration, total_distance, Wh_km, total_soc_consumed,ppt_data = analysis_Energy(log_file, km_file)
        analysis_output = analysis_output.getvalue()

        # Extract folder name from folder_path
        folder_name = os.path.basename(folder_path)

        # Create a new PowerPoint presentation
        prs = Presentation()

        # Add title slide with 'Selawik' style
        title_slide_layout = prs.slide_layouts[0]
        slide = prs.slides.add_slide(title_slide_layout)
        title = slide.shapes.title
        title.text = f"Analysis Results from Folder - {folder_name}"
        title.text_frame.paragraphs[0].font.bold = True
        title.text_frame.paragraphs[0].font.size = Pt(36)  # Corrected to Pt
        title.text_frame.paragraphs[0].font.name = 'Selawik'

        rows = len(ppt_data)+1
        cols = 2
        table_slide_layout = prs.slide_layouts[5]
        slide = prs.slides.add_slide(table_slide_layout)
        shapes = slide.shapes
        title_shape = shapes.title
        title_shape.text = "Analysis Results:"

            # Centering the title horizontally
        title_shape.left = Inches(0.5)
        title_shape.top = Inches(0.5)  # Adjust as needed

        # Setting the font size of the title
        title_shape.text_frame.paragraphs


        # Define maximum number of rows per slide
        max_rows_per_slide = 13
        # Add some space between title and table
        title_shape.top = Inches(0.5)

        table = shapes.add_table(max_rows_per_slide+1, cols, Inches(1), Inches(1.5), Inches(8), Inches(5)).table
        table.columns[0].width = Inches(4)
        table.columns[1].width = Inches(4)
        table.cell(0, 0).text = "Metric"
        table.cell(0, 1).text = "Value"


        # Initialize row index
        row_index = 1

        # Iterate over data and populate the table
        for key, value in ppt_data.items():
            # Check if current slide has reached maximum rows
            if row_index > max_rows_per_slide:
                # Add a new slide
                slide = prs.slides.add_slide(table_slide_layout)
                shapes = slide.shapes
                title_shape = shapes.title
                title_shape.text = "Analysis Results:"

                # Add a new table to the new slide
                table = shapes.add_table(max_rows_per_slide+1, cols, Inches(1), Inches(1.5), Inches(8), Inches(5)).table
                table.columns[0].width = Inches(4)
                table.columns[1].width = Inches(4)
                table.cell(0, 0).text = "Metric"
                table.cell(0, 1).text = "Value"


                # Reset row index for the new slide
                row_index = 1

            # Populate the table
            table.cell(row_index, 0).text = key
            table.cell(row_index, 1).text = str(value)

            # Increment row index
            row_index += 1




        # Add image slide with title and properly scaled image
        slide_layout = prs.slide_layouts[5]
        slide = prs.slides.add_slide(slide_layout)

        # Remove the unwanted title placeholder
        for shape in slide.shapes:
            if shape.is_placeholder:
                slide.shapes._spTree.remove(shape._element)

        # Add the title
        title_shape = slide.shapes.add_textbox(Inches(1), Inches(0.5), prs.slide_width - Inches(2), Inches(1))
        title_shape.text = "Graph Analysis"

        # Add the image and adjust its position and size
        graph_width = prs.slide_width - Inches(1)
```

```
        graph_height = prs.slide_height - Inches(2)
        left = (prs.slide_width - graph_width) / 2
        top = (prs.slide_height - graph_height) / 2 + Inches(1)
        pic = slide.shapes.add_picture('graph.png', left, top, width=graph_width, height=graph_height)

        # Save the presentation
        output_file_name = f"{folder_path}/analysis_{folder_name}.pptx"
        prs.save(output_file_name)

    #except Exception as e:
        #print("Error:", e)



#folder_path = "/home/sanjith/Documents/Graphs _ creta/15-51_16-00"

# Get the list of files in the folder
files = os.listdir(folder_path)

# Initialize variables to store file paths
log_file = None
km_file = None



# Path to the main folder containing subfolders
main_folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER"

# Iterate over subfolders
for subfolder in os.listdir(main_folder_path):
    subfolder_path = os.path.join(main_folder_path, subfolder)
    print(subfolder)
    if os.path.isdir(subfolder_path):
        log_file = None
        km_file = None
        # Find 'log' and 'km' files
        l=0
        for file in os.listdir(subfolder_path):
            if file.startswith('log') and file.endswith('.csv'):
                log_file = os.path.join(subfolder_path, file)
                l = 1
            elif file.startswith('km') and file.endswith('.csv'):
                km_file = os.path.join(subfolder_path, file)
                l =2


# Read the CSV file into a pandas DataFrame

            if (l ==2):
                total_duration =0
                total_distance =0
                Wh_km =0
                SOC_consumed=0
                mode_values=0


                ### plot graphs
                ##plot_ghps(log_file)
                total_duration, total_distance, Wh_km,SOC_consumed,ppt_data=analysis_Energy(log_file,km_file)
                capture_analysis_output(log_file, km_file, subfolder_path)
```

Thursday, March 28, 2024 10:13 PM

This function is called within Analysis_Energy function.

Iterate over rows of GPS data

Let's first learn functions used in this code

1. haversine - measures distance b/w two points on earth's surface (in km)

2. adjust_current(row) - Adjust "current reading" to 0 if motorspeed_340920578 is 0 for 10 or more consecutive data points
   (A single row is used)

3. plot_gps(log_file) - Plots → PackCurr_6, MotorSpeed_34..., AC current, AC voltage, throttle percentage etc.

   Input - log file
   Output - Graph image

4. analysis_Energy(log_file, km_file)

- **Inputs**: Paths to the log file (operational data) and km file (GPS data).
- **Output**: Various calculated metrics including total ride duration, total distance covered, energy efficiency, and state of charge consumed. Also, a dictionary containing data for a PowerPoint presentation.

5. capture_analysis_output(log_file, km_file, folder_path)

   Output :- power point, word

2. Adjust_current function

(motorspeed ~~battery current~~)

```
# Define a function to set current to zero if RPM is zero for 10 or more consecutive points
def adjust_current(row):
    adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0)
    if row['MotorSpeed_340920578'] == 0:
        adjust_current.zero_count += 1
    else:
        adjust_current.zero_count = 0

    if adjust_current.zero_count >= 10:
        return 0
    else:
        return row['PackCurr_6']
def plot_gps(log_file):
```

or even more?

In Truth some negligible current will be there due to battery's self discharge, electronics in standby mode or other sensors

Purpose :- we want data mainly when scooter is running → so when the data shows some non-zero value for current, then in future we may also collect other parameter value corresponding to non-zero current value which are actually supposed to be zero from scooter's idle motor's perspective
↳ so put fake current data == 0

Thursday, March 28, 2024 11:12 PM

```python
    # Define a function to set current to zero if RPM is zero for 10 or more consecutive points
    def adjust_current(row):
        adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0)
        if row['MotorSpeed_340920578'] == 0:
            adjust_current.zero_count += 1
        else:
            adjust_current.zero_count = 0

        if adjust_current.zero_count >= 10:
            return 0
        else:
            return row['PackCurr_6']
```

```python
# Sample data resembling rows from your DataFrame
data = [
    {'MotorSpeed_340920578': 0, 'PackCurr_6': 5},
    {'MotorSpeed_340920578': 0, 'PackCurr_6': 4},
    # Add more data points as needed...
]

# Initialize a list to keep track of adjusted current values
adjusted_currents = []

# Initialize a counter for consecutive zero motor speeds
zero_count = 0

# Iterate over each row in the data
for row in data:
    # Check if the motor speed is zero
    if row['MotorSpeed_340920578'] == 0:
        zero_count += 1  # Increment the zero count
    else:
        zero_count = 0  # Reset the count if motor speed is not zero

    # Adjust the current based on the consecutive zero count
    if zero_count >= 10:
        adjusted_currents.append(0)  # Set current to zero if zero count is 10 or mo
    else:
        adjusted_currents.append(row['PackCurr_6'])  # Otherwise, use the original c

# At this point, `adjusted_currents` contains the adjusted current values for each r
print(adjusted_currents)
```

```python
data['localtime'] = pd.to_datetime(data['localtime'], format='%d
data.set_index('localtime', inplace=True)
data['PackCurr_6'] = data.apply(adjust_current, axis=1)
```

The increment of the zero_count variable within the adjust_current function happens only when the function is called and when the specific condition within the function (i.e., row['MotorSpeed_340920578'] == 0)

adjust_current

Definition Search

In this file

53 def adjust_current(row):

6 References Search

In this file

54 adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0)

54 = getattr(adjust_current, 'zero_count', 0)

56 adjust_current.zero_count += 1

58 adjust_current.zero_count = 0

60 if adjust_current.zero_count >= 10:

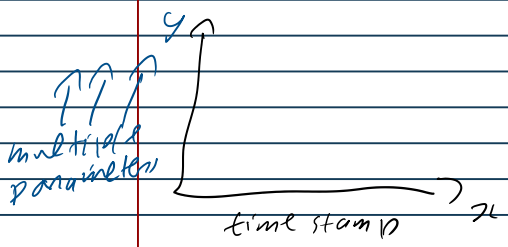
73 data.apply(adjust_current, axis=1)

Row-wise Application: Pandas then iterates over each row of the DataFrame data, passing each row as a Series to the adjust_current function. This iteration mimics a loop, where each row is processed one at a time.

Friday, March 29, 2024 11:20 AM

- to load & visualize EV scooter

3. Plot_ghps function



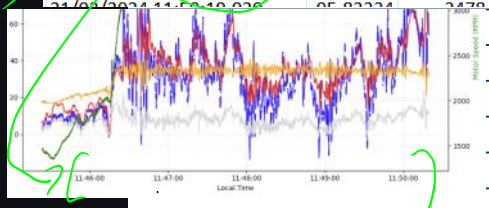
plot_ghps function is designed to visualize various telemetry data from the EV scooter's log files, such as pack current, motor speed, AC current and voltage, and throttle percentage over time.

Friday, March 29, 2024 11:16 AM

3. plot_ghps function

```
        return row['PackCurr_6']
def plot_ghps(log_file):
    data = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21\log_fi
    # Apply the adjustment function to the DataFrame

        Applies the adjust_current function to each row of the DataFrame to adjust the PackCurr_6
        column based on consecutive zero readings in the motor speed, as explained previously.

    data['localtime'] = pd.to_datetime(data['localtime'], format='%d/%m/%Y %H:%M:%S.%f', dayfirst=True)
    data.set_index('localtime', inplace=True)
    data['PackCurr_6'] = data.apply(adjust_current, axis=1)

    # Create a figure and axes for plotting
    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Plot 'PackCurr_6' on primary y-axis
    line1, = ax1.plot(data.index, -data['PackCurr_6'], color='blue', label='PackCurr_6')
    ax1.set_ylabel('Pack Current (A)', color='blue')
    ax1.yaxis.set_label_coords(-0.1, 0.7)  # Adjust label position

    # Create secondary y-axis for 'MotorSpeed_340920578' (RPM)
    ax2 = ax1.twinx()
    line2, = ax2.plot(data.index, data['MotorSpeed_340920578'], color='green', label='Motor Speed')
    ax2.set_ylabel('Motor Speed (RPM)', color='green')

    # Add 'AC_Current_340920579' to primary y-axis
    line3, = ax1.plot(data.index, data['AC_Current_340920579'], color='red', label='AC Current')

    # Add 'AC_Voltage_340920580' scaled to 10x to the left side y-axis
    line4, = ax1.plot(data.index, data['AC_Voltage_340920580'] * 10, color='orange', label='AC Voltage (x10)')

    # Add 'Throttle_408094978' to the left side y-axis
    line5, = ax1.plot(data.index, data['Throttle_408094978'], color='lightgray', label='Throttle (%)')

    # Hide the y-axis label for 'AC_Current_340920579'
    ax1.get_yaxis().get_label().set_visible(False)

    # Set x-axis label and legend
    ax1.set_xlabel('Local Time')
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    # Add a title to the plot
    plt.title('Battery Pack, Motor Data, and Throttle')

    # Format x-axis ticks as hours:minutes:seconds
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))

    # Set grid lines lighter
    ax1.grid(True, linestyle=':', linewidth=0.5, color='gray')
    ax2.grid(True, linestyle=':', linewidth=0.5, color='gray')

    # Enable cursor to display values on graphs
    mplcursors.cursor([line1, line2, line3, line4, line5])

    # Save the plot as an image or display it
    plt.tight_layout()  # Adjust layout to prevent clipping of labels
    plt.savefig('graph.png')  # Save the plot as an image
    plt.show()
def analysis_Energy(log_file, km_file):
```

localtime	percentage	MotorSpe
21/03/2024 11:50:19.540	95.83334	3475
21/03/2024 11:50:19.468	33.33333	3466
21/03/2024 11:50:19.379	79.16666	3479
21/03/2024 11:50:19.280	95.83334	3485
21/03/2024 11:50:19.200	91.66666	3466
21/03/2024 11:50:19.100	91.66666	3475



4. analysis_Energy fun

Friday, March 29, 2024 12:45 PM

The analysis_Energy function is designed to perform a comprehensive analysis of energy consumption and efficiency based on the data from electric vehicle (EV) scooters. This involves analyzing operational data (like motor speed, battery current, and voltage) and GPS data (for calculating distances traveled).

Data Preparation and Cleaning

python Copy code

```
if 'localtime' not in data.columns:
    print("Error: 'localtime' column not found in the DataFrame.")
    return None, None, None, None

data.dropna(subset=['SOCAh_8'], inplace=True)
data['localtime'] = pd.to_datetime(data['localtime'], format='%d/%m/%Y %H:%M:%S.%f',
data.set_index('localtime', inplace=True)
```

The function checks for the necessary columns, handles missing values, and converts the `localtime` column to a datetime object. It then sets `localtime` as the index for easier time-series analysis.

Ride Duration Calculation

python Copy code

```
start_time = data['localtime'].min()
end_time = data['localtime'].max()
total_duration = end_time - start_time
```

Calculates the total duration of the ride by finding the minimum and maximum `localtime` values, which represent the start and end times of the data recording.

Replace NaN with '0' eg:- at the start of the dataset there is no previous row to subtract from.

Given time elapsed between consecutive data points

(difference between each row & it's predecessor) in 'localtime' column

```
data['Time_Diff'] = data['localtime'].diff().dt.total_seconds().fillna(0)
data_resampled = data.resample('s').ffill()
actual_ah = abs((data_resampled['PackCurr_6'] * data_resampled['Time_Diff']).sum()) / 3600
watt_h = abs((data_resampled['PackCurr_6'] * data_resampled['PackVol_6'] * data_resampled['Time_Diff']).sum() / 3600
```

CONVERT time difference → Total seconds

sum: we all intervals of the dataset output one entire

→ no need, it will be always "1"

"Calculate" ampere-hours" & "watt-hours" using numerical integration over time intervals between data points

Purpose: make it uniform time-interval

Data made in a way where 1 second time interval '.ffill()' fills the gaps with last available value.

11:07 - 11:02 = 5 secs = Time difference

Pack current (10A) x 5 sec = ampere-seconds for each INTERVAL

".sum" (summation Σ) → Adds up all products over all INTERVALS

Then in Div. by 3600 we will get (conversion seconds to hours)

Ampere-Hours (Ah) :- amount of charge transferred by a steady current of 1A flowing for an hour.

Watt-Hours (Wh) :- how much work can be done (energy) by a device running at 1 watt of power for one hour.

Relating to Battery Usage:

- **Ampere-Hours (Ah):** Represents the total charge transferred from the battery. It's a direct measure of battery usage in terms of charge depletion.
- **Watt-Hours (Wh):** Represents the total energy consumed. It accounts for both the voltage and current, providing a comprehensive view of how much power has been used, reflecting both

Relating to Battery Usage:

- **Ampere-Hours (Ah):** Represents the total charge transferred from the battery. It's a direct measure of battery usage in terms of charge depletion.
- **Watt-Hours (Wh):** Represents the total energy consumed. It accounts for both the voltage and current, providing a comprehensive view of how much power has been used, reflecting both the driving efficiency and battery energy depletion.

Friday, March 29, 2024 3:19 PM

To check Total Distance travelled.

GPS Distance Calculation

python Copy code

```
for i in range(1, len(data_KM)):
    lat1, lon1 = data_KM.loc[i - 1, 'latitude'], data_KM.loc[i - 1, 'longitude']
    lat2, lon2 = data_KM.loc[i, 'latitude'], data_KM.loc[i, 'longitude']
    distance = haversine(lat1, lon1, lat2, lon2)
    total_distance += distance
```

Iterates through the GPS data to calculate the total distance traveled by summing up the distances between consecutive GPS points using the Haversine formula.

In this BIG function, we will go through --> 1. Data loading 2. Calculating Ride Duration
3. Distance calc (Haversine) 4. Energy Consumption & Efficiency
5. SOC Analysis 6. Performance metrics
Eco also Sports, Custom modes
7. Voltage & Temp analysis 8. Regenerative Braking Analysis
9. OUTPUT - Prints all

4. analysis_Energy fun- code

Friday, March 29, 2024 12:03 PM

```
64  > def plot_ghps(log_file): ...
125         plt.show()
126     def analysis_Energy(log_file, km_file):
127         dayfirst=True
128
129         data = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER
130
131         data_KM = pd.read_csv(r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOL
132
133         total_duration = 0
134         total_distance = 0
135         Wh_km = 0
136         SOC_consumed = 0
137
138         # Check if 'localtime' column exists in data DataFrame
139         if 'localtime' not in data.columns:
140             print("Error: 'localtime' column not found in the DataFrame.")
141             return None, None, None, None
142
143
144
145
146         # Drop rows with missing values in 'SOCAh_8' column
147         data.dropna(subset=['SOCAh_8'], inplace=True)
148
149         # Convert the 'localtime' column to datetime s
150         data['localtime'] = pd.to_datetime(data['localtime'], format='%d/%m/%Y %H:%M:%S.%f',dayf
151
152         # Calculate the start time and end time
153         start_time = data['localtime'].min()
154         end_time = data['localtime'].max()
155
156         # Calculate the total time taken for the ride
157         total_duration = end_time - start_time
158
159         total_hours = total_duration.seconds // 3600
160         total_minutes = (total_duration.seconds % 3600) // 60
161
162         print(f"Total time taken for the ride: {int(total_hours):02d}:{int(total_minutes):02d}")
163
164         # Calculate the time difference between consecutive rows
165         data['Time_Diff'] = data['localtime'].diff().dt.total_seconds().fillna(0)
166
167         # Set the 'localtime' column as the index
168         data.set_index('localtime', inplace=True)
169
170         # Resample the data to have one-second intervals and fill missing values with previous one
171         data_resampled = data.resample('s').ffill()
172
173         # Calculate the time difference between consecutive rows
174         data_resampled['Time_Diff'] = data_resampled.index.to_series().diff().dt.total_seconds().f
175
176         # Calculate the actual Ampere-hours (Ah) using the trapezoidal rule for numerical integrat
```

data loading in

watt-hours per kilometer (measure of energy efficiency)

SOC of 100% means fully charged
SOC of 0% → completely discharged

To calculate total time for ride.

↓ To calculate Ah and Wh

Ampere-Hours (Ah) :- amount of charge transferred by a steady current of 1A flowing for an hour.

Watt-Hours (Wh) :- how much work can be done (energy) by a device running at 1 watt of power for one hour.

```
actual_ah = abs((data_resampled['PackCurr_6'] * data_resampled['Time_Diff']).sum()) / 3600
print("Actual Ampere-hours (Ah): {:.2f}".format(actual_ah))

# Calculate the actual Watt-hours (Wh) using the trapezoidal rule for numerical integration
watt_h = abs((data_resampled['PackCurr_6'] * data_resampled['PackVol_6'] * data_resampled['
print("Actual Watt-hours (Wh):{:.2f}" .format(watt_h))

###########   starting and ending ah
starting_soc = data['SOCAh_8'].iloc[0]
ending_soc = data['SOCAh_8'].iloc[-1]

print("Starting SoC (Ah):{:.2f}".format (starting_soc))
print("Ending SoC (Ah):{:.2f}".format  (ending_soc))

#################### KM --------------------
# Convert the 'localtime' column to datetime format
data_KM['localtime'] = pd.to_datetime(data_KM['localtime'], format='%d/%m/%Y %H:%M:%S.%f',

# Initialize total distance covered
total_distance = 0

# Iterate over rows to compute distance covered between consecutive points
for i in range(1, len(data_KM)):
    # Get latitude and longitude of consecutive points
    lat1, lon1 = data_KM.loc[i - 1, 'latitude'], data_KM.loc[i - 1, 'longitude']
    lat2, lon2 = data_KM.loc[i, 'latitude'], data_KM.loc[i, 'longitude']

    # Calculate distance between consecutive points
    distance = haversine(lat1, lon1, lat2, lon2)

    # Add distance to total distance covered
    total_distance += distance

print("Total distance covered (in kilometers):{:.2f}".format(total_distance))

###############   Wh/Km
Wh_km = abs(watt_h / total_distance)
print("WH/KM:{:.2f}". format (watt_h / total_distance))

# Assuming 'data' is your DataFrame with 'SOC_8' column
initial_soc = data['SOC_8'].iloc[0]  # Initial SOC percentage
final_soc = data['SOC_8'].iloc[-1]   # Final SOC percentage

# Calculate total SOC consumed
total_soc_consumed =  abs(final_soc - initial_soc)

print ("Total SOC consumed:{:.2f}".format (total_soc_consumed),"%")


# Check if the mode remains constant or changes
mode_values = data['Mode_Ack_408094978'].unique()
```

```python
    mode_values = data['Mode_Ack_408094978'].unique()

    if len(mode_values) == 1:
        # Mode remains constant throughout the log file
        mode = mode_values[0]
        if mode == 3:
            print("Mode is Custom mode.")
        elif mode == 2:
            print("Mode is Sports mode.")
        elif mode == 1:
            print("Mode is Eco mode.")
    else:
        # Mode changes throughout the log file
        mode_counts = data['Mode_Ack_408094978'].value_counts(normalize=True) * 100
        for mode, percentage in mode_counts.items():
            if mode == 3:
                print(f"Custom mode: {percentage:.2f}%")
            elif mode == 2:
                print(f"Sports mode: {percentage:.2f}%")
            elif mode == 1:
                print(f"Eco mode: {percentage:.2f}%")
    ##preak current##
      # Calculate power using PackCurr_6 and PackVol_6
      data_resampled['Power'] = data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

      # Find the peak power
      peak_power = data_resampled['Power'].min()
      print("Peak Power:", peak_power)

      # Calculate the average power
      average_power = data_resampled['Power'].mean()
      print("Average Power:", average_power)

      # Calculate energy regenerated (in watt-hours)
      energy_regenerated = ((data_resampled[data_resampled['Power'] > 0]['Power']*data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours

      # Calculate total energy consumed (in watt-hours)
      total_energy_consumed =  ((data_resampled[data_resampled['Power'] < 0]['Power']*data_resampled['Time_Diff']).sum()) / 3600  # Convert seconds to hours

      print("total",total_energy_consumed)
      print("total energy regenerated",energy_regenerated)

      # Calculate regenerative effectiveness as a percentage
      if total_energy_consumed != 0:
       regenerative_effectiveness = (energy_regenerated / total_energy_consumed) * 100
       print("Regenerative Effectiveness (%):", regenerative_effectiveness)
      else:
       print("Total energy consumed is 0, cannot calculate regenerative effectiveness.")
```

Understanding the Modes

Eco Mode (mode == 1): Likely the most energy-efficient setting, offering the best range by possibly limiting power output or speed.
Sports Mode (mode == 2): Offers higher performance at the cost of increased energy consumption.
Custom Mode (mode == 3): Allows for user-defined settings that could vary in terms of energy efficiency and performance.

Regenerative braking is a technology used in electric vehicles (EVs), hybrid electric vehicles (HEVs), and electric bicycles or scooters, which captures the vehicle's kinetic energy during deceleration and converts it back into electrical energy. Instead of dissipating this energy as heat through conventional brakes, regenerative braking systems use the electric motor as a generator to convert the kinetic energy into electrical energy, which is then stored in the vehicle's battery.

Kinetic energy is the energy that an object possesses due to its motion. When an electric vehicle (EV) or any vehicle is in motion, it has kinetic energy proportional to its mass and the square of its velocity ($KE = 1/2\ mv^2$). During deceleration—when the vehicle is slowing down—this kinetic energy doesn't just disappear; it must be transformed or transferred due to the conservation of energy, a fundamental principle of physics.

In traditional vehicles with conventional braking systems (friction brakes), deceleration is achieved by converting the vehicle's kinetic energy into heat through friction. Brake pads press against the brake rotors, creating friction that slows down the wheels, turning the kinetic energy into thermal energy, which is then dissipated into the air.

```
        print("Total energy consumed is 0, cannot calculate regenerative effectiveness.")

    # Calculate idling time percentage (RPM was zero for more than 5 seconds)
    idling_time = (data['MotorSpeed_340920578'] == 0).sum()
    idling_percentage = (idling_time / len(data)) * 100
    print("Idling time percentage:", idling_percentage)

    # Calculate time spent in specific speed ranges
    speed_ranges = [(0, 10), (10, 20), (20, 30), (30, 40), (40, 50),(50, 60),(60,70)]
    speed_range_percentages = {}

    for range_ in speed_ranges:
        speed_range_time = ((data['MotorSpeed_340920578'] * 0.016 >= range_[0]) & (data['MotorSpeed_340920578'] * 0.016 < range_[1])).sum()
        speed_range_percentage = (speed_range_time / len(data)) * 100
        speed_range_percentages[f"Time spent in {range_[0]}-{range_[1]} km/h"] = speed_range_percentage
        print(f"Time spent in {range_[0]}-{range_[1]} km/h: {speed_range_percentage:.2f}%")


##########################################################################################################################################################


        # Calculate power using PackCurr_6 and PackVol_6
    data_resampled['Power'] = -data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

    # Find the peak power
    peak_power = data_resampled['Power'].max()

    # Get the maximum cell voltage
    max_cell_voltage = data_resampled['MaxCellVol_5'].max()
```

```
    max_cell_voltage = data_resampled['MaxCellVol_5'].max()

    # Find the index where the maximum voltage occurs
    max_index = data_resampled['MaxCellVol_5'].idxmax()

    # Retrieve the corresponding cell ID using the index
    max_cell_id = data_resampled['MaxVoltId_5'].loc[max_index]

    # Get the minimum cell voltage
    min_cell_voltage = data_resampled['MinCellVol_5'].min()

    # Find the index where the minimum voltage occurs
    min_index = data_resampled['MinCellVol_5'].idxmin()

    # Retrieve the corresponding cell ID using the index
    min_cell_id = data_resampled['MinVoltId_5'].loc[min_index]

    voltage_difference = max_cell_voltage - min_cell_voltage

    # Get the maximum temperature
    max_temp = data_resampled['MaxTemp_7'].max()

    # Find the index where the maximum temperature occurs
    max_temp_index = data_resampled['MaxTemp_7'].idxmax()

    # Retrieve the corresponding temperature ID using the index
    max_temp_id = data_resampled['MaxTempId_7'].loc[max_temp_index]
```

1. Peak Power
Scientific Basis: Peak power (the maximum value of Power) indicates the highest rate at which the scooter's motor consumes or generates energy during the observed period. High peak power during acceleration can indicate good performance but may also stress the battery and electrical components.
EV Perspective: Understanding peak power demands helps in designing electrical systems and components that can withstand these peaks without overheating or failing prematurely.
2. Cell Voltage Monitoring
Max/Min Cell Voltage & Difference: Monitoring the maximum and minimum cell voltages (MaxCellVol_5 and MinCellVol_5) and their difference provides insight into the battery's state of health and balance. A large voltage difference can indicate imbalanced cells, which may reduce battery efficiency and lifespan.
Voltage vs. Health: Voltage levels give a direct measure of the state of charge and health of individual cells within the battery pack. Cells with significantly lower voltages may be deteriorating faster than others.
3. Temperature Monitoring
Max/Min Temperatures & Difference: The maximum and minimum temperatures (MaxTemp_7 and MinTemp_7), along with specific component temperatures (FETs, AFEs, PCB, MCU, and motor), are critical for ensuring the scooter operates within safe thermal limits. Excessive temperature can indicate inefficiencies or potential failures.
Thermal Management: Effective thermal management is essential for maintaining battery health, ensuring safety, and optimizing performance. High temperatures can accelerate battery degradation, while very low temperatures can reduce battery efficiency and power output.
4. Component-Specific Temperatures
FET (Field-Effect Transistor) Temperature: Indicates the health and efficiency of the power electronics. High temperatures may suggest inefficiencies or overloading.
AFE (Analog Front End) Temperature: Reflects the condition of the circuitry managing the battery's monitoring and safety functions.
PCB (Printed Circuit Board) Temperature: High temperatures can signal potential issues with the electrical components or circuit design.
MCU (Microcontroller Unit) Temperature: Critical for ensuring the central processing unit operates within safe limits.
Motor Temperature: Directly affects performance and longevity. High temperatures may indicate mechanical issues, excessive loading, or cooling system inadequacies.
5. Abnormal Motor Temperature at High RPMs
Monitoring motor temperature relative to RPM provides insights into the motor's efficiency and cooling system performance. Abnormal temperatures at high RPMs could signal issues like inadequate cooling, mechanical failures, or overloading.

Sunday, March 31, 2024 4:15 PM

Using and monitoring Field-Effect Transistor (FET) temperature in EV scooters is crucial due to several reasons, mainly related to the efficiency, safety, and reliability of the scooter's power electronics system. FETs are semiconductor devices that play a critical role in controlling power flow in various parts of the scooter, including the motor controller, battery management system, and other electronic circuits.

FETs generate heat as they switch on and off to control power flow.

Motor Control:- Speed and Torque Management: FETs are used in the motor controllers of EV scooters to modulate the power supplied to the electric motors. By rapidly switching the current on and off (a process known as Pulse Width Modulation, or PWM), FETs can precisely control the motor's speed and torque, offering smooth acceleration and deceleration.

```
# Get the minimum temperature
min_temp = data_resampled['MinTemp_7'].min()

# Find the index where the minimum temperature occurs
min_temp_index = data_resampled['MinTemp_7'].idxmin()

# Retrieve the corresponding temperature ID using the index
min_temp_id = data_resampled['MinTempId_7'].loc[min_temp_index]

# Calculate the difference in temperature
temp_difference = max_temp - min_temp

# Get the maximum temperature of FetTemp_8
max_fet_temp = data_resampled['FetTemp_8'].max()

# Get the maximum temperature of AfeTemp_12
max_afe_temp = data_resampled['AfeTemp_12'].max()

# Get the maximum temperature of PcbTemp_12
max_pcb_temp = data_resampled['PcbTemp_12'].max()

# Get the maximum temperature of MCU_Temperature_408094979
max_mcu_temp = data_resampled['MCU_Temperature_408094979'].max()

# Check for abnormal motor temperature at high RPMs
max_motor_temp = data_resampled['Motor_Temperature_408094979'].max()

# Check for abnormal motor temperature at high RPMs for at least 15 seconds
abnormal_motor_temp = (data_resampled['Motor_Temperature_408094979'] < 10) & (data_resampled['MotorSpeed_340920578'] > 3500)
abnormal_motor_temp_mask = abnormal_motor_temp.astype(int).groupby(abnormal_motor_temp.ne(abnormal_motor_temp.shift()).cumsum()).cumsum()

# Check if abnormal condition persists for at least 15 seconds
abnormal_motor_temp_detected = (abnormal_motor_temp_mask >= 120).any()
```

to identify abnormal conditions in an electric vehicle (EV) scooter, specifically looking for situations where the motor temperature is abnormally low (< 10 degrees, likely Celsius) while the motor speed is high (> 3500 RPM) for at least 15 seconds. This could indicate a sensor malfunction or data recording error, as such low temperatures are unlikely under high-speed conditions.

SKIP

```
# Add these variables and logic to ppt_data
ppt_data = {
    "Total time taken for the ride": total_duration,
    "Actual Ampere-hours (Ah)": actual_ah,
    "Actual Watt-hours (Wh)": watt_h,
    "Starting SoC (Ah)": starting_soc,
    "Ending SoC (Ah)": ending_soc,
    "Total distance covered (in kilometers)": total_distance,
    "WH/KM": watt_h / total_distance,
    "Total SOC consumed": total_soc_consumed,
    "Mode": "",
    "Peak Power": peak_power,
    "Average Power": average_power,
    "Total Energy Regenerated": energy_regenerated,
    "Regenerative Effectiveness": regenerative_effectiveness,
    "Lowest Cell Voltage": min_cell_voltage,
    "Highest Cell Voltage": max_cell_voltage,
    "Difference in Cell Voltage": voltage_difference,
    "Minimum Temperature": min_temp,
    "Maximum Temperature": max_temp,
    "Difference in Temperature": temp_difference,
    "Maximum Fet Temperature": max_fet_temp,
    "Maximum Afe Temperature": max_afe_temp,
    "Maximum PCB Temperature": max_pcb_temp,
    "Maximum MCU Temperature": max_mcu_temp,
    "Maximum Motor Temperature": max_motor_temp,
    "Abnormal Motor Temperature Detected": abnormal_motor_temp_detected
}
mode_values = data_resampled['Mode_Ack_408094978'].unique()
```

to add data

```
        }
        mode_values = data_resampled['Mode_Ack_408094978'].unique()
        if len(mode_values) == 1:
            mode = mode_values[0]
            if mode == 3:
                ppt_data["Mode"] = ["Custom mode"]
            elif mode == 2:
                ppt_data["Mode"] = ["Sports mode"]
            elif mode == 1:
                ppt_data["Mode"] = ["Eco mode"]
        else:
            # Mode changes throughout the log file
            mode_counts = data_resampled['Mode_Ack_408094978'].value_counts(normalize=True) * 100
            ppt_data["Mode"] = []  # Initialize list to store modes
            for mode, percentage in mode_counts.items():
                if mode == 3:
                    ppt_data["Mode"].append(f"Custom mode: {percentage:.2f}%")
                elif mode == 2:
                    ppt_data["Mode"].append(f"Sports mode: {percentage:.2f}%")
                elif mode == 1:
                    ppt_data["Mode"].append(f"Eco mode: {percentage:.2f}%")

            print("ppt",len(ppt_data))

            # Add calculated parameters to ppt_data
            ppt_data["Idling time percentage"] = idling_percentage
            ppt_data.update(speed_range_percentages)
    ########################################################################################################## recent data

    # Calculate power using PackCurr_6 and PackVol_6
    data_resampled['Power'] = -data_resampled['PackCurr_6'] * data_resampled['PackVol_6']

    # Find the peak power
    peak_power = data_resampled['Power'].max()
    print("Peak Power:", peak_power)

    # Get the maximum cell voltage
    max_cell_voltage = data_resampled['MaxCellVol_5'].max()

    # Find the index where the maximum voltage occurs
    max_index = data_resampled['MaxCellVol_5'].idxmax()

    # Retrieve the corresponding cell ID using the index
    max_cell_id = data_resampled['MaxVoltId_5'].loc[max_index]

        # Get the minimum cell voltage
    min_cell_voltage = data_resampled['MinCellVol_5'].min()

    # Find the index where the minimum voltage occurs
    min_index = data_resampled['MinCellVol_5'].idxmin()

    # Retrieve the corresponding cell ID using the index
    min_cell_id = data_resampled['MinVoltId_5'].loc[min_index]

    voltage_difference = max_cell_voltage - min_cell_voltage


    print("Lowest Cell Voltage:", min_cell_voltage, "V, Cell ID:", min_cell_id)
    print("Highest Cell Voltage:", max_cell_voltage, "V, Cell ID:", max_cell_id)
    print("Difference in Cell Voltage:", voltage_difference, "V")

    # Get the maximum temperature
    max_temp = data_resampled['MaxTemp_7'].max()

    # Find the index where the maximum temperature occurs
```

already did this

```
        # Find the index where the maximum temperature occurs
        max_temp_index = data_resampled['MaxTemp_7'].idxmax()

        # Retrieve the corresponding temperature ID using the index
        max_temp_id = data_resampled['MaxTempId_7'].loc[max_temp_index]


            # Get the minimum temperature
        min_temp = data_resampled['MinTemp_7'].min()

        # Find the index where the minimum temperature occurs
        min_temp_index = data_resampled['MinTemp_7'].idxmin()

        # Retrieve the corresponding temperature ID using the index
        min_temp_id = data_resampled['MinTempId_7'].loc[min_temp_index]
        # Calculate the difference in temperature
        temp_difference = max_temp - min_temp

        # Print the information
        print("Maximum Temperature:", max_temp, "C, Temperature ID:", max_temp_id)
        print("Minimum Temperature:", min_temp, "C, Temperature ID:", min_temp_id)
        print("Difference in Temperature:", temp_difference, "C")

        # Get the maximum temperature of FetTemp_8
        max_fet_temp = data_resampled['FetTemp_8'].max()
        print("Maximum Fet Temperature:", max_fet_temp, "C")

        # Get the maximum temperature of AfeTemp_12
        max_afe_temp = data_resampled['AfeTemp_12'].max()
        print("Maximum Afe Temperature:", max_afe_temp, "C")

        # Get the maximum temperature of PcbTemp_12
        max_pcb_temp = data_resampled['PcbTemp_12'].max()
        print("Maximum PCB Temperature:", max_pcb_temp, "C")

        # Get the maximum temperature of MCU_Temperature_408094979
        max_mcu_temp = data_resampled['MCU_Temperature_408094979'].max()
```

```
        # Get the maximum temperature of MCU_Temperature_408094979
        max_mcu_temp = data_resampled['MCU_Temperature_408094979'].max()
        print("Maximum MCU Temperature:", max_mcu_temp, "C")

            # Check for abnormal motor temperature at high RPMs
        max_motor_temp = data_resampled['Motor_Temperature_408094979'].max()


        print("Maximum Motor Temperature:", max_motor_temp, "C")


        # Check for abnormal motor temperature at high RPMs for at least 15 seconds
        abnormal_motor_temp = (data_resampled['Motor_Temperature_408094979'] < 10) & (data_resampled['MotorSpeed_340920578'] > 3500)

        # Convert to a binary mask indicating consecutive occurrences
        abnormal_motor_temp_mask = abnormal_motor_temp.astype(int).groupby(abnormal_motor_temp.ne(abnormal_motor_temp.shift()).cumsum()).cumsum()

        # Check if abnormal condition persists for at least 15 seconds
        if (abnormal_motor_temp_mask >= 15).any():
            print("Abnormal motor temperature detected: NTC has issues - very low temperature at high RPMs")

    ##########################################################################################################################################

        return total_duration, total_distance, Wh_km, total_soc_consumed,ppt_data


    folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21"
```

Return these values into "analysis_Energy" function

later this function will be called

5. capture_analysis_output function

Sunday, March 31, 2024 5:04 PM

skip

5. capture_analysis_output function- code

Sunday, March 31, 2024 5:04 PM

SKIP ppt function skipped

```
folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER\MAR_21"
def capture_analysis_output(log_file, km_file, folder_path):
    #try:
        # Capture print statements
        analysis_output = io.StringIO()
        output_file = "analysis_results.docx"

        with redirect_stdout(analysis_output):
            total_duration, total_distance, Wh_km, total_soc_consumed,ppt_data = analysis_Energy(log_file, km_file)
        analysis_output = analysis_output.getvalue()

        # Extract folder name from folder_path
        folder_name = os.path.basename(folder_path)

        # Create a new PowerPoint presentation
        prs = Presentation()

        # Add title slide with 'Selawik' style
        title_slide_layout = prs.slide_layouts[0]
        slide = prs.slides.add_slide(title_slide_layout)
        title = slide.shapes.title
        title.text = f"Analysis Results from Folder - {folder_name}"
        title.text_frame.paragraphs[0].font.bold = True
        title.text_frame.paragraphs[0].font.size = Pt(36)  # Corrected to Pt
        title.text_frame.paragraphs[0].font.name = 'Selawik'

        rows = len(ppt_data)+1
        cols = 2
        table_slide_layout = prs.slide_layouts[5]
        slide = prs.slides.add_slide(table_slide_layout)
```

function called (inside ppt output function)

```
        slide = prs.slides.add_slide(table_slide_layout)
        shapes = slide.shapes
        title_shape = shapes.title
        title_shape.text = "Analysis Results:"

                # Centering the title horizontally
        title_shape.left = Inches(0.5)
        title_shape.top = Inches(0.5)  # Adjust as needed

        # Setting the font size of the title
        title_shape.text_frame.paragraphs


        # Define maximum number of rows per slide
        max_rows_per_slide = 13
        # Add some space between title and table
        title_shape.top = Inches(0.5)

        table = shapes.add_table(max_rows_per_slide+1, cols, Inches(1), Inches(1.5), Inches(8), Inches(5)).table
        table.columns[0].width = Inches(4)
        table.columns[1].width = Inches(4)
        table.cell(0, 0).text = "Metric"
        table.cell(0, 1).text = "Value"


        # Initialize row index
```

```
    # Initialize row index
    row_index = 1

    # Iterate over data and populate the table
    for key, value in ppt_data.items():
        # Check if current slide has reached maximum rows
        if row_index > max_rows_per_slide:
            # Add a new slide
            slide = prs.slides.add_slide(table_slide_layout)
            shapes = slide.shapes
            title_shape = shapes.title
            title_shape.text = "Analysis Results:"

            # Add a new table to the new slide
            table = shapes.add_table(max_rows_per_slide+1, cols, Inches(1), Inches(1.5), Inches(8), Inches(5)).table
            table.columns[0].width = Inches(4)
            table.columns[1].width = Inches(4)
            table.cell(0, 0).text = "Metric"
            table.cell(0, 1).text = "Value"


            # Reset row index for the new slide
            row_index = 1

        # Populate the table
        table.cell(row_index, 0).text = key
        table.cell(row_index, 1).text = str(value)
```

```
        table.cell(row_index, 1).text = str(value)

        # Increment row index
        row_index += 1




    # Add image slide with title and properly scaled image
    slide_layout = prs.slide_layouts[5]
    slide = prs.slides.add_slide(slide_layout)

    # Remove the unwanted title placeholder
    for shape in slide.shapes:
        if shape.is_placeholder:
            slide.shapes._spTree.remove(shape._element)

    # Add the title
    title_shape = slide.shapes.add_textbox(Inches(1), Inches(0.5), prs.slide_width - Inches(2), Inches(1))
    title_shape.text = "Graph Analysis"

    # Add the image and adjust its position and size
    graph_width = prs.slide_width - Inches(1)
    graph_height = prs.slide_height - Inches(2)
    left = (prs.slide_width - graph_width) / 2
```

```
    left = (prs.slide_width - graph_width) / 2
    top = (prs.slide_height - graph_height) / 2 + Inches(1)
    pic = slide.shapes.add_picture('graph.png', left, top, width=graph_width, height=graph_height)

    # Save the presentation
    output_file_name = f"{folder_path}/analysis_{folder_name}.pptx"
    prs.save(output_file_name)
```

end of function

Remaining code

Sunday, March 31, 2024 5:08 PM

```
            prs.save(output_file_name)

        #except Exception as e:
            #print("Error:", e)



#folder_path = "/home/sanjith/Documents/Graphs _ creta/15-51_16-00"

# Get the list of files in the folder
files = os.listdir(folder_path)

# Initialize variables to store file paths
log_file = None
km_file = None




# Path to the main folder containing subfolders
main_folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard\MAIN_FOLDER"

# Iterate over subfolders
for subfolder in os.listdir(main_folder_path):
    subfolder_path = os.path.join(main_folder_path, subfolder)
    print(subfolder)
    if os.path.isdir(subfolder_path):
        log_file = None
        km_file = None
        # Find 'log' and 'km' files
        l=0
        for file in os.listdir(subfolder_path):
            if file.startswith('log') and file.endswith('.csv'):
                log_file = os.path.join(subfolder_path, file)
                l = 1
            elif file.startswith('km') and file.endswith('.csv'):
                km_file = os.path.join(subfolder_path, file)
                l =2


    # Read the CSV file into a pandas DataFrame

            if (l ==2):
                total_duration =0
                total_distance =0
                Wh_km =0
                SOC_consumed=0
                mode_values=0


                ### plot graphs
                ##plot_ghps(log_file)
                total_duration, total_distance, Wh_km,SOC_consumed,ppt_data=analysis_Energy(log_file,km_file)
                capture_analysis_output(log_file, km_file, subfolder_path)
```

```
log_file_found = False
km_file_found = False

for file in os.listdir(subfolder_path):
    if file.startswith('log') and file.endswith('.csv'):
        log_file = os.path.join(subfolder_path, file)
        log_file_found = True
    elif file.startswith('km') and file.endswith('.csv'):
        km_file = os.path.join(subfolder_path, file)
        km_file_found = True

# Proceed only if both files are found
if log_file_found and km_file_found:
    # Reset analysis variables
    total_duration = 0
    total_distance = 0
    Wh_km = 0
    SOC_consumed = 0
    mode_values = 0
```

OpenAI integration- 434 line code

Sunday, March 31, 2024 7:05 PM

```
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import mplcursors  # Import mplcursors
from matplotlib.widgets import CheckButtons  # Import CheckButtons
import numpy as np  # Import numpy for handling NaN values

import os

from openai import OpenAI

# OPENAI_API_KEY = ''

folder_path = r"C:\Git_Projects\Automationdashboard\Automationdashboard"

# Get the list of files in the folder

files = os.listdir(folder_path)

# Initialize variables to store file paths
log_file = None

# Find files starting with 'log' and 'km'
for file in files:
    if file.startswith('log') and file.endswith('.csv'):
        log_file = os.path.join(folder_path, file)

# Read the CSV file into a pandas DataFrame
data = pd.read_csv(log_file)

# Convert 'localtime' column to datetime format and set it as index
data['localtime'] = pd.to_datetime(data['localtime'])
data.set_index('localtime', inplace=True)

# Define a function to set current to zero if RPM is zero for 10 or more consecutive points
def adjust_current(row):
    adjust_current.zero_count = getattr(adjust_current, 'zero_count', 0)
    if row['MotorSpeed_340920578'] == 0:
        adjust_current.zero_count += 1
    else:
        adjust_current.zero_count = 0

    if adjust_current.zero_count >= 10:
        return 0
    else:
        return row['PackCurr_6']

def generate_label(fault_name, max_pack_dc_current, max_ac_current, min_pack_dc_current, fault_timestamp,
                   current_speed, throttle_percentage, relevant_data, max_battery_voltage):
    error_cause = ""
    if max_pack_dc_current < -130:
        error_cause = "battery overcurrent"
    elif max_ac_current > 220:
        error_cause = "motor overcurrent"
    elif max_battery_voltage > 69:
        error_cause = "battery overvoltage"

    completion = client.chat.completions.create(
        model="gpt-3.5-turbo",
        messages=[
            {"role": "system", "content": "Please provide a summary explaining why the fault occurred along with occurrence time and cause."},
            {"role": "system", "content": f"The fault '{fault_name}' was triggered due to {error_cause} at {fault_timestamp}."},
            {"role": "user", "content": f"Speed at fault occurrence: {current_speed * 0.016} km/h, Throttle percentage: {throttle_percentage}."},
            {"role": "user", "content": f"{gpt_analyze_data(max_pack_dc_current, max_ac_current,min_pack_dc_current, current_speed, throttle_percentage, relevant_data,
fault_name,max_battery_voltage)}"}

        ]
    )

    generated_messages = completion.choices[0].message.content.split('\n')

    # Print out the generated messages for debugging
    print("Generated Messages:", generated_messages)

    # Extract content from dictionaries and join into a single string
    return

def gpt_analyze_data(max_pack_dc_current, max_ac_current, min_pack_dc_current, current_speed, throttle_percentage,
                     relevant_data, fault_name, max_battery_voltage):
    # Placeholder for GPT-analyzed statements
    analyzed_statements = []

    print(fault_name)

    # Generate statements based on data analysis
    if fault_name == 'ChgPeakProt_9':
        if min_pack_dc_current > 80:
            analyzed_statements.append(
                {"role": "system",
                 "content": f"The DC Regen current exceeded the limit 60A form battery, the current limit is now {min_pack_dc_current}A."})

    if fault_name == 'Overcurrent_Fault_408094978':
        if max_ac_current > 220:
            analyzed_statements.append(
                {"role": "system", "content": "The AC current exceeded the limit, suggesting a potential motor overcurrent."})
        if current_speed * 0.016 < 10:
            analyzed_statements.append(
                {"role": "system",
                 "content": "The vehicle was moving at a low speed, which may indicate challenging terrain or obstacles."})
        if throttle_percentage > 80:
```

```
        analyzed_statements.append(
            {"role": "system", "content": "The throttle percentage was high, indicating high torque demand."})
    if max_ac_current > 220 and throttle_percentage > 80:
        analyzed_statements.append(
            {"role": "system",
             "content": "The motor overcurrent could be triggered due to high torque demand, possibly caused by obstructions or the wheel being stuck for a prolonged period."})
#### should go on to different catogry
    if max_pack_dc_current < -60 and continuous_dc_exceeded_limit(max_pack_dc_current, relevant_data):
        analyzed_statements.append(
            {"role": "system",
             "content": "The DC current exceeded the continuous maximum limit for 90 seconds, indicating a potential battery overcurrent."})

    if max_pack_dc_current < -120:
        analyzed_statements.append(
            {"role": "system",
             "content": f"The DC current exceeded the the max current allowed by the battery i.e. 120A, the current is {max_pack_dc_current}"})
################################################
  if fault_name == 'DriveError_Controller_OverVoltag_408094978':
    if max_battery_voltage > 70:
        analyzed_statements.append(
            {"role": "user",
             "content": f"The controller voltage {max_battery_voltage}V exceeded 70V, was actual suggesting potential overvoltage during high regenerative braking."})

        # Check if ChgFetStatus_9 has gone to zero and if negative current exceeds -60A
        if 'ChgFetStatus_9' in relevant_data and 'PackCurr_6' in relevant_data:
            chg_fet_status = relevant_data['ChgFetStatus_9']
            pack_dc_current = relevant_data['PackCurr_6']

            analyzed_statements.append(
                {"role": "user",
                 "content": f"Over ovltage error is occured due to ChgFetStatus_9 going to 0 means battery has disconnected."})


            # Check if ChgFetStatus_9 has gone to zero and negative current exceeds -60A
        if (chg_fet_status == 0).any() and (pack_dc_current * -1 < -60).any():
            analyzed_statements.append(
                {"role": "user",
                 "content": f"ChgFetStatus_9 went to zero because negative current was {pack_dc_current*-1} exceeding -60A,  suggesting potential Overcurrent during high regenerative
braking."})

#########################################################################################################################
            prev_mode_value = None
            timestamp = None



            # Additional condition to check the rate of change of RPM
            if 'MotorSpeed_340920578' in relevant_data and 'PackCurr_6' in relevant_data:
                rpm_series = relevant_data['MotorSpeed_340920578']
                pack_curr_series = relevant_data['PackCurr_6']
                timestamp_series = relevant_data['localtime']

                # Create a DataFrame with RPM values, 'PackCurr_6', and corresponding timestamps
                df = pd.DataFrame({'MotorSpeed_340920578': rpm_series, 'PackCurr_6': pack_curr_series, 'Timestamp': timestamp_series})

                # Find the timestamp when 'PackCurr_6' becomes more than 60
                timestamp_pack_curr_exceed_60 = df.loc[df['PackCurr_6'] > 60, 'Timestamp'].iloc[0]
                print(timestamp_pack_curr_exceed_60)


                # Find t2, 1.5 seconds before t1
                timestamp_t2 = timestamp_pack_curr_exceed_60 - pd.Timedelta(seconds=1.5)
                print(timestamp_t2)

                # Find motor speed at t1 and t2 only if DataFrame is not empty
                if not df.empty:
                    # Find motor speed at t1 when current > 60A
                    motor_speed_t1 = df.loc[df['Timestamp'] == timestamp_pack_curr_exceed_60, 'MotorSpeed_340920578']

                    if not motor_speed_t1.empty:
                        motor_speed_t1 = motor_speed_t1.iloc[0]
                        print("Motor speed at t1 (when current > 60A):", motor_speed_t1)


                    # Find motor speed at t2, 1.5 seconds before t1
                    nearest_timestamp_t2 = df.loc[(df['Timestamp'] <= timestamp_t2), 'Timestamp'].max()
                    print(nearest_timestamp_t2)

                    motor_speed_t2 = df.loc[df['Timestamp'] == nearest_timestamp_t2, 'MotorSpeed_340920578']

                    if not motor_speed_t2.empty:
                        motor_speed_t2 = motor_speed_t2.iloc[0]
                        print("Motor speed at t2 (1.5 seconds before t1):", motor_speed_t2)


            # Check if there's a change in Mode_Ack_408094978
            if 'Mode_Ack_408094978' in relevant_data:
                mode_ack = relevant_data['Mode_Ack_408094978']
                unique_values = np.unique(mode_ack)  # Get unique values, handling NaN



                if len(unique_values) > 1:

                                        # Assuming relevant_data['Mode_Ack_408094978'] contains the data
                    mode_ack_series = relevant_data['Mode_Ack_408094978']
                    timestamp_series = relevant_data['localtime']  # Assuming relevant_data['localtime'] contains the timestamps

                    # Create a DataFrame with mode values and corresponding timestamps
                    df = pd.DataFrame({'Mode_Ack_408094978': mode_ack_series, 'Timestamp': timestamp_series})

                    # Filter DataFrame to rows where mode is 1
                    mode_1_df = df[df['Mode_Ack_408094978'] == 1][::-1]

                    # If there are rows where mode is 1, get the first occurrence
                    if not mode_1_df.empty:
                        first_occurrence = mode_1_df.iloc[0]
```

```
                    exact_time_mode_1 = first_occurrence['Timestamp']

                    print("Timestamp of mode change:", exact_time_mode_1)
##############################
                    analyzed_statements.append({
  "role": "user",
  "content": f"Mode change at time ({exact_time_mode_1}) could be the reason for the sudden reduction in speed and the resulting high current greater than 60A."
})
##############################


                            # Assuming relevant_data['PackCurr_6'] contains the data
                    pack_curr_series = relevant_data['PackCurr_6']
                    timestamp_series = relevant_data['localtime']  # Assuming relevant_data['localtime'] contains the timestamps

                    # Create a DataFrame with 'PackCurr_6' values and corresponding timestamps
                    df = pd.DataFrame({'PackCurr_6': pack_curr_series, 'Timestamp': timestamp_series})

                    # Filter DataFrame to rows where 'PackCurr_6' is less than -60
                    less_than_minus_60_df = df[df['PackCurr_6'] > 60]

                    # If there are rows where 'PackCurr_6' is less than -60, get the first occurrence
                    if not less_than_minus_60_df.empty:
                        first_occurrence = less_than_minus_60_df.iloc[0]
                        timestamp = first_occurrence['Timestamp']
                        print("Timestamp when 'PackCurr_6' goes below -60:", timestamp)

                                # Assuming relevant_data['MotorSpeed_340920578'] contains the motor speed data
                    motor_speed_series = relevant_data['MotorSpeed_340920578']
                    timestamp_series = relevant_data['localtime']  # Assuming relevant_data['localtime'] contains the timestamps

                    # Create a DataFrame with motor speed values and corresponding timestamps
                    df = pd.DataFrame({'MotorSpeed': motor_speed_series, 'Timestamp': timestamp_series})

                    # Filter DataFrame to rows where timestamps are between exact_time_mode_1 and timestamp
                    filtered_df = df[(df['Timestamp'] >= exact_time_mode_1) & (df['Timestamp'] <= timestamp)]

                    # Calculate the motor speed drop as the difference between maximum and minimum motor speed within the specified time range
                    motor_speed_drop = filtered_df['MotorSpeed'].max() - filtered_df['MotorSpeed'].min()

                    # Calculate the time difference in seconds and milliseconds
                    time_difference_seconds = (timestamp - exact_time_mode_1).total_seconds()
                    time_difference_milliseconds = time_difference_seconds * 1000

                    print("Motor Speed Drop:", motor_speed_drop)
                    print("Time taken for motor speed drop (seconds):", time_difference_seconds)
                    print("Time taken for motor speed drop (milliseconds):", time_difference_milliseconds)

##################################
                    # Calculate the rate of change of RPM (RPM/s)
                    time_difference_seconds = (timestamp_pack_curr_exceed_60 - timestamp_t2).total_seconds()
                    rate_of_change_rpm = motor_speed_drop / time_difference_seconds

                    print("Rate of change of RPM:", rate_of_change_rpm)

                    if motor_speed_drop > 150:
                        analyzed_statements.append({
            "role": "user",
            "content": f"The rate of change of motor speed (RPM) ({rate_of_change_rpm}) exceeds 150 RPM, potentially causing high current to the battery."
        })
##################################

    print("here",analyzed_statements)
    return analyzed_statements


def continuous_dc_exceeded_limit(max_pack_dc_current, data):
    # Set the threshold for continuous DC current limit
    dc_current_threshold = 60  # Amperes

    # Set the duration for continuous DC current limit in seconds
    duration_threshold_seconds = 90

    # Check if the maximum DC current exceeds the threshold
    if max_pack_dc_current > dc_current_threshold:
        # Get the index of the first occurrence of the maximum DC current exceeding the threshold
        start_index = data.index[data['PackCurr_6'] > dc_current_threshold][0]

        # Calculate the end index considering the duration threshold
        end_index = start_index + pd.Timedelta(seconds=duration_threshold_seconds)

        # Check if the DC current exceeds the threshold continuously for the duration threshold
        continuous_exceeded = all(data.loc[start_index:end_index, 'PackCurr_6'] > dc_current_threshold)

        return continuous_exceeded

    return False


def analyze_fault(csv_file, fault_name):
    # Load CSV data into a DataFrame
    data = pd.read_csv(csv_file)

    # Convert 'localtime' to datetime
    data['localtime'] = pd.to_datetime(data['localtime'])

    # Check if the column exists in the DataFrame
    if fault_name not in data.columns:
        print(f"Column '{fault_name}' not found in the DataFrame.")
        return

    # Filter rows where fault occurred
    fault_data = data[data[fault_name] == 1]

    if fault_data.empty:
        print(f"No {fault_name} data found.")
        return
```

```
    else:
        print(f"{fault_name} data found.")

    # Sort fault data by 'localtime' in ascending order
    fault_data.sort_values(by='localtime', inplace=True)

    # Extract timestamp of the first occurrence of the fault
    fault_timestamp = fault_data.iloc[0]['localtime']

    # Calculate start time (5 minutes before the fault)
    start_time = fault_timestamp - pd.Timedelta(minutes=5)
    # Calculate end time (2 minutes after the fault)
    end_time = fault_timestamp + pd.Timedelta(minutes=2)

    # Filter data for 5 minutes before and after the fault
    relevant_data = data[(data['localtime'] >= start_time) & (data['localtime'] <= end_time)]

    print(relevant_data['localtime'])

    # Find the maximum AC and DC currents before the fault occurrence
    max_ac_current = relevant_data['AC_Current_340920579'].max()
    max_dc_current = relevant_data['BatteryCurrent_340920578'].max()
    max_pack_dc_current = relevant_data['PackCurr_6'].min()
    min_pack_dc_current = relevant_data['PackCurr_6'].max()

    # Find maximum battery voltage
    max_battery_voltage = (relevant_data['BatteryVoltage_340920578'].max())*10

    # Capture current speed and throttle percentage at fault occurrence
    current_speed = fault_data.loc[fault_data['localtime'] == fault_timestamp, 'MotorSpeed_340920578'].values[0]
    throttle_percentage = fault_data.loc[fault_data['localtime'] == fault_timestamp, 'Throttle_408094978'].values[0]


    generate_label(fault_name, max_pack_dc_current, max_ac_current, min_pack_dc_current, fault_timestamp,current_speed, throttle_percentage, relevant_data,
max_battery_voltage)

    print("Occurrence Time:", fault_timestamp)
    print("Maximum AC Current before fault:", max_ac_current, "A")
    print("Maximum DC Current before fault from MCU:", max_dc_current, "A")
    print("Maximum DC Current before fault from Battery:", max_pack_dc_current, "A")
    print("Maximum Battery Voltage:", max_battery_voltage, "V")

    # Create a figure and axes for plotting
    fig, ax1 = plt.subplots(figsize=(10, 6))

    # Plot 'PackCurr_6' on primary y-axis
    line1, = ax1.plot(relevant_data['localtime'], -relevant_data['PackCurr_6'], color='blue', label='PackCurr_6')
    ax1.set_ylabel('Pack Current (A)', color='blue')
    ax1.yaxis.set_label_coords(-0.1, 0.7)  # Adjust label position

    # Create secondary y-axis for 'MotorSpeed_340920578' (RPM)
    ax2 = ax1.twinx()
    line2, = ax2.plot(relevant_data['localtime'], relevant_data['MotorSpeed_340920578'], color='green', label='Motor Speed')
    ax2.set_ylabel('Motor Speed (RPM)', color='green')

    # Add 'AC_Current_340920579' to primary y-axis
    line3, = ax1.plot(relevant_data['localtime'], relevant_data['AC_Current_340920579'], color='red', label='AC Current')

    # Add 'AC_Voltage_340920580' scaled to 10x to the left side y-axis
    line4, = ax1.plot(relevant_data['localtime'], relevant_data['AC_Voltage_340920580'] * 10, color='orange', label='AC Voltage (x10)')

    # Add 'Throttle_408094978' to the left side y-axis
    line5, = ax1.plot(relevant_data['localtime'], relevant_data['Throttle_408094978'], color='lightgray', label='Throttle (%)')

    # Add 'DchgFetStatus_9*10' to the left side y-axis
    line6, = ax1.plot(relevant_data['localtime'], relevant_data['DchgFetStatus_9'] * 10, color='purple', label='DchgFetStatus_9 (x10)')

    # Add 'ChgFetStatus_9*10' to the left side y-axis
    line7, = ax1.plot(relevant_data['localtime'], relevant_data['ChgFetStatus_9'] * 10, color='brown', label='ChgFetStatus_9 (x10)')

    # Add 'BatteryVoltage_340920578*10' to the left side y-axis
    line8, = ax1.plot(relevant_data['localtime'], relevant_data['BatteryVoltage_340920578'] * 10, color='magenta', label='BatteryVoltage_340920578 (x10)')

    line9, = ax1.plot(relevant_data['localtime'], relevant_data['SOC_8'] , color='magenta', label='SOC_8 ')

    line10, = ax1.plot(relevant_data['localtime'], relevant_data['Mode_Ack_408094978'] *10, color='green', label='Mode_Ack_408094978 ')


    # Hide the y-axis label for 'AC_Current_340920579'
    ax1.get_yaxis().get_label().set_visible(False)

    # Add vertical line for fault occurrence
    ax1.axvline(x=fault_timestamp, color='gray', linestyle='--', label='Fault Occurrence')

    # Set x-axis label and legend
    ax1.set_xlabel('Local Time')
    ax1.legend(loc='upper left')
    ax2.legend(loc='upper right')

    # Add a title to the plot
    plt.title('Battery Pack, Motor Data, and Throttle')

    # Format x-axis ticks as hours:minutes:seconds
    ax1.xaxis.set_major_formatter(mdates.DateFormatter('%H:%M:%S'))

    # Set grid lines lighter
    ax1.grid(True, linestyle=':', linewidth=0.5, color='gray')
    ax2.grid(True, linestyle=':', linewidth=0.5, color='gray')

    # Enable cursor for data points
    mplcursors.cursor(hover=True)

    # Create checkboxes
    rax = plt.axes([0.8, 0.1, 0.15, 0.3])  # Adjust position to the right after the graph
    labels = ('PackCurr_6', 'AC_Current_340920579', 'MotorSpeed_340920578', 'AC_Voltage_340920580', 'Throttle_408094978', 'DchgFetStatus_9', 'ChgFetStatus_9', 'BatteryVoltage_
340920578','SOC_8','Mode_Ack_408094978')
```

```
    lines = [line1, line3, line2, line4, line5, line6, line7, line8]
    visibility = [line.get_visible() for line in lines]
    check = CheckButtons(rax, labels, visibility)

    def func(label):
        index = labels.index(label)
        lines[index].set_visible(not lines[index].get_visible())
        plt.draw()

    check.on_clicked(func)

    plt.show()

client = OpenAI()

# Call the function for 'DriveError_Controller_OverVoltag_408094978'
analyze_fault(log_file, 'DriveError_Controller_OverVoltag_408094978')
```