**Environmental Monitoring in Parking System in IOT**

Environmental monitoring in a parking system typically involves using various sensors and technologies to collect data, which can then be processed and analyzed to make informed decisions or trigger actions. Implementing a complete system with code can be a complex task, but I can provide a simple example in Python to demonstrate the concept of environmental monitoring within a parking system using a temperature sensor as an illustration. In a real-world scenario, you would need the appropriate hardware and APIs for other sensors and systems.

Environmental monitoring in a parking system refers to the practice of using various sensors, technologies, and data analysis methods to continuously assess and manage environmental conditions within and around a parking facility. The data collected through environmental monitoring systems can be used for real-time decision-making, improved security, energy efficiency, and enhancing the overall user experience within the parking facility. Drivers can access this information through a mobile app or website, or through signs that are posted in the parking lot itself.

**Import the necessary Libraries:**

**import time**

**import datetime**

**import js**

**import logging**

***# Sensor and data collection libraries***

**import RPi.GPIO as GPIO**

**import Adafruit\_DHT**

**import smbus**

**import psutil**

***# Data analysis and processing***

**import numpy**

**import pandas as pd**

**import matplotlib.pyplot as plt**

**import seaborn as sns**

***# Communication and alerts***

**import smtplib # For sending email alerts**

***# Database libraries***

**import sqlite3**

***# IoT integration***

**import paho.mqtt.client as mqtt # For MQTT communication**

***# Import device-specific SDKs or libraries***

***# Security and encryption***

**from cryptography.fernet import Fernet**

**Loading the Dataset:**

**import pandas as pd**

**dataset\_path = 'your\_dataset.csv'**

**df = pd.read\_csv(dataset\_path)**

**print(df.head())**

**mean\_temperature = df['Temperature'].mean()**

**print(f"Mean Temperature: {mean\_temperature}°C")**

**Data Cleaning:**

1.Handling missing data: Removes rows with missing values.

2.Handling outliers: Filters the dataset to include only temperature values within a specified range.

3.Data type conversion: Converts the 'Timestamp' column to a datetime data type.

4.Removing duplicates: Eliminates duplicate rows.

5.Renaming columns: Renames columns for clarity.

**df = df.dropna()**

**lower\_bound = 10**

**upper\_bound = 30**

**df = df[(df['Temperature'] >= lower\_bound) & (df['Temperature'] <= upper\_bound)]**

**df['Timestamp'] = pd.to\_datetime(df['Timestamp'])**

**df = df.drop\_duplicates()**

**df = df.rename(columns={'Temp': 'Temperature'})**

**df = df.reset\_index(drop=True)**

**Data Analysis:**

1. Basic data summary: Calculating summary statistics (e.g., mean, standard deviation) of numeric columns.

2. Time series analysis: Aggregating data over time (e.g., hourly and daily) to examine trends.

3.Data visualization: Creating a line plot to visualize hourly mean temperature.

4.Correlation analysis: Calculating the correlation matrix to identify relationships between variables.

5.Data filtering: Filtering data based on specific conditions (e.g., high temperature).

6.Aggregation: Aggregating data by specific categories (e.g., monthly mean temperature).

**df['Timestamp'] = pd.to\_datetime(df['Timestamp'])**

**df.set\_index('Timestamp', inplace=True)**

**hourly\_mean\_temp = df['Temperature'].resample('H').mean()**

**daily\_max\_temp = df['Temperature'].resample('D').max()**

***# 3. Data Visualization***

***# Create plots to visualize data***

**plt.figure(figsize=(10, 6))**

**sns.lineplot(data=hourly\_mean\_temp, x=hourly\_mean\_temp.index, y=hourly\_mean\_temp.values)**

**plt.title('Hourly Mean Temperature')**

**plt.xlabel('Time')**

**plt.ylabel('Temperature (°C)')**

**plt.show()**

***# 4. Correlation Analysis***

***# Analyze correlations between variables***

**correlation\_matrix = df.corr()**

***# 5. Data Filtering***

***# Filter data based on specific conditions***

**high\_temperature\_data = df[df['Temperature'] > 25]**

***# 6. Aggregation***

***# Aggregate data by specific categories***

**monthly\_mean\_temp = df.groupby(df.index.month)['Temperature'].mean()**

***# Display the results***

**print("Summary Statistics:")**

**print(summary\_stats)**

**print("\nHourly Mean Temperature:")**

**print(hourly\_mean\_temp.head())**

**print("\nDaily Maximum Temperature:")**

**print(daily\_max\_temp.head())**

**print("\nCorrelation Matrix:")**

**print(correlation\_matrix)**

**print("\nHigh Temperature Data:")**

**print(high\_temperature\_data.head())**

**print("\nMonthly Mean Temperature:")**

**print(monthly\_mean\_temp)**

***# Visualizations can be further customized and saved to files for reports*.**

You can expand upon these analyses or perform more complex statistical and machine learning analyses, depending on the questions and goals of your environmental monitoring system.

**import random**

**import time**

***# Simulate a temperature sensor***

**def read\_temperature():**

**temperature = random.uniform(10, 30)**

**return temperature**

**while True:**

**temperature = read\_temperature()**

***# Check temperature threshold***

**if temperature > 25:**

***# Temperature is too high, trigger an action (e.g., turn on fans or alert personnel)***

**print(f"Temperature is too high: {temperature}°C")**

**time.sleep(300)**

This code snippet simulates a temperature sensor and checks the temperature every 5 minutes. If the temperature goes above a certain threshold (25°C in this case), it prints a message indicating that the temperature is too high. In a real-world implementation, you would replace the temperature simulation code with actual sensor readings and replace the print statements with actions or alerts that are relevant to your parking system's needs.

**Circuit diagram:**

