AIR QUALITY ANALYSIS AND PREDICTION IN TAMILNADU

PHASE 3 Team member : R.RAVEENDAR

NM ID : au723721205033

**Introdution :**

**Understanding Air Quality through Data Analysis**

Air quality is a critical aspect of our daily lives, impacting both our health and the environment we inhabit. It is influenced by a complex interplay of natural factors, such as weather patterns, and human activities, such as industrial emissions and transportation. To gain insights into this crucial aspect of our environment, researchers and environmental agencies collect vast amounts of data on various air pollutants, atmospheric conditions, and geographic locations.This introduction delves into the world of air quality datasets and the importance of their analysis. These datasets often encompass measurements of key pollutants such as particulate matter (PM2.5 and PM10), ozone (O3), nitrogen dioxide (NO2), sulfur dioxide (SO2), carbon monoxide (CO), and more. They are collected through an extensive network of monitoring stations deployed in urban, suburban, and rural areas, providing a comprehensive view of air quality across regions and time.Analyzing air quality datasets can yield valuable insights into trends, seasonal variations, pollution sources, and the effectiveness of environmental policies. Researchers and data scientists use these datasets to develop predictive models for air quality forecasting and to assess the health risks associated with exposure to air pollutants.

In this exploration, we will discuss the key steps involved in preprocessing air quality data, the challenges associated with data quality, and the various techniques for analysis and visualization. Additionally, we will examine how air quality data is leveraged to inform policy decisions, mitigate pollution, and ultimately improve the quality of the air we breathe.

The study of air quality datasets is not only an exercise in data analysis but a crucial endeavor for public health, environmental sustainability, and the well-being of current and future generations. Let's embark on this journey to understand the vital role that data plays in addressing the challenges of air quality and working towards a cleaner, healthier planet.

**Preprocessing of air quality data sets :**

**Data Collection:**

Gather the air quality data from reliable sources, such as government agencies or research institutions. Make sure you have information on the pollutants you are interested in, the location, date, and time of measurement, and any associated metadata.

**Data Import:**

Load the data into your chosen data analysis environment, such as Python with libraries like Pandas, or R. You can read data from common file formats like CSV, Excel, or database tables.

**Data Inspection:**

Begin by examining the dataset to get a sense of its structure, including the number of rows and columns, data types, and any missing values. You should use methods like `.head()`, `.info()`, and `.describe()` in Pandas.

**Handling Missing Data:**

Air quality data often contains missing or erroneous values. You need to decide how to handle them. You can drop rows with missing data or use methods like interpolation or imputation to fill in missing values.

**Data Cleaning:**

- Remove duplicates if any.

- Correct or remove outliers that can distort analysis and modeling.

- Check for inconsistencies or errors in the data and correct them.

**Data Transformation:**

- Convert date and time columns to a standardized format.

- Extract relevant features from the date and time, such as day of the week or hour of the day.

- Convert data units to a consistent scale, if necessary (e.g., converting ppm to µg/m³).

- Calculate daily or hourly averages if your data is in a different format.

**Feature Engineering:**

- Create new features that might be useful, such as rolling averages or trends.

- Extract relevant geographical information if available (latitude, longitude, city, region, etc.).

**Data Scaling and Normalization:**

Depending on your analysis or modeling techniques, you might need to scale or normalize the data to ensure that different variables are on a similar scale.

**Data Encoding:**

If your data contains categorical variables, you may need to encode them into numerical values. One-hot encoding is a common method for this purpose.

**Data Splitting:**

If you plan to build predictive models, split the data into training and testing sets to evaluate model performance.

**Data Visualization:**

Visualize the preprocessed data to gain insights into the air quality trends, correlations, and anomalies. Visualization can help in understanding the data better and identifying potential patterns or relationships.

**Documentation:**

Maintain detailed documentation of all the preprocessing steps, as this will be crucial for reproducibility and sharing your work.

**Quality Control:**

Before proceeding with your analysis or modeling, perform a final quality check to ensure that the data is in the desired format and free from errors.

**Saving the Preprocessed Data:**

After preprocessing, save the data in a clean and structured format (e.g., CSV) for easy access in future analyses.

**MODELING PROCESS**

Modeling can range from simple statistical models to complex deep learning models, and the choice of model depends on the complexity of the problem and the availability of data. It's important to remember that modeling is just one part of the broader data analysis process, which includes problem formulation, data collection, preprocessing, and interpretation of results.

**Define the Problem:**

Before diving into modeling, it's essential to clearly define the problem you want to solve. This involves specifying the goal of your model, such as making predictions, classifying data, clustering data, or gaining insights. You should also consider the success criteria for your model.

**Data Preparation:**

This step involves collecting, cleaning, and preprocessing the data, as discussed earlier. Data should be transformed into a format suitable for modeling. This typically includes splitting the data into training and testing sets to assess model performance.

**Selecting a Model:**

Choose an appropriate modeling technique based on the nature of your problem. The choice of model can include regression for continuous outcomes, classification for categorical outcomes, clustering for grouping data, or dimensionality reduction for feature selection.

**Feature Selection and Engineering :**

It's crucial to select the most relevant features and, if necessary, engineer new features. Feature engineering involves creating new variables from existing ones or transforming existing variables to improve model performance.

**Model Training:**

In this step, the selected model is trained on the training data. The model learns to recognize patterns, relationships, or trends in the data. The training process involves adjusting model parameters to minimize the error or loss function.

**Model Evaluation:**

After training, you need to assess how well your model performs. Common evaluation metrics include accuracy, precision, recall, F1-score, mean squared error, and others, depending on the problem type. Cross-validation is often used to obtain a more reliable estimate of a model's performance.

**Hyperparameter Tuning:**

Many machine learning models have hyperparameters that can be tuned to optimize model performance. Techniques like grid search or random search can help you find the best hyperparameter values.

**Model Validation:**

Once you've fine-tuned your model, it's essential to validate it on the testing data to ensure that it generalizes well to unseen data. This step helps to detect overfitting, where a model performs well on training data but poorly on new data.

**Deployment:**

If the model meets your criteria and is ready for practical use, you can deploy it into your application or workflow. Deployment can involve integrating the model into a web application, mobile app, or a data pipeline.

**Monitoring and Maintenance:**

Models in production need to be regularly monitored to ensure they continue to perform well. Data distribution can change over time, requiring model retraining or updates to adapt to new patterns.

**Interpretability:**

Depending on the use case, model interpretability may be crucial. Understanding how a model makes decisions can be important for trust and regulatory compliance.

**Documentation and Communication:**

Throughout the modeling process, maintain detailed documentation of model choices, parameters, and performance. Effective communication of model results to stakeholders is essential.

**Iterative Process:**

Model building is often an iterative process. You may need to go back and make adjustments to the model based on new data or insights.

**Problem Formulation:**

1. **Problem Definition**: Clearly articulate the problem you're trying to solve. It's essential to have a precise problem statement that outlines what you're trying to achieve. Define the problem's scope and boundaries.
2. **Objectives**: State the objectives or goals of your project. What do you want to accomplish by addressing this problem? What are the specific outcomes or deliverables you aim to achieve?
3. **Audience and Stakeholders**: Identify the key stakeholders and audience for your project. Understanding their needs and expectations can help shape the problem formulation.
4. **Data-Driven Approach**: Describe how data will be used to address the problem. What data will be used, and how will it inform decision-making or solutions?
5. **Metrics and Success Criteria**: Define the metrics and success criteria by which you'll measure the project's success. For example, if you're building a predictive model, what evaluation metrics will be used to assess its performance?
6. **Hypotheses**: If applicable, formulate hypotheses that express your expectations about the problem. For instance, in hypothesis testing, you might have null and alternative hypotheses.
7. **Constraints and Assumptions**: Be clear about any constraints or assumptions underlying your problem formulation. These may include limitations on data, resources, or external factors that can impact your project.
8. **Project Plan**: Outline a high-level plan for how you intend to address the problem, including timelines, milestones, and tasks.
9. **Validation**: Ensure that the problem is relevant and valuable. Consider whether solving this problem will lead to actionable insights, better decision-making, or improved processes.
10. **Iterative Process**: Keep in mind that problem formulation can be an iterative process. As you learn more about the data and the problem, you may need to refine your problem statement and objectives.

**Python code :**

# Get params

event = data.get('event')

# Get the state object from the name

state\_value = hass.states.get('input\_select.harmony\_hub').state

# Get info

dt = datetime.datetime.now()

#state.attributes.get('last\_triggered')

time = "%02d:%02d" % (dt.hour, dt.minute)

# Sensor update

hass.states.set('sensor.activity\_badge', state\_value, {

'friendly\_name': time, #state\_value,

'entity\_picture': '/local/images/activities/{}.png'.format(state\_value.lower()),

'unit\_of\_measurement': 'Act'

})

# ecrire\_etat\_entite.py

# Force l'écriture de l'état d'une entité (ses attributs ne sont pas modifiés)

#Récupération de l'entité à écrire

inputEntity = data.get('entity\_id')

#Récupératon de la valeur à écrire

inputState = data.get('state')

#Chargement de l'entité à son état actuel

inputStateObject = hass.states.get(inputEntity)

#On recopie les attributs

inputAttributesObject = inputStateObject.attributes.copy()

#Ecriture de la nouvelle valeur de l'entité avec conservation des attributs

hass.states.set(inputEntity, inputState, inputAttributesObject)

from getpass import getpass

from libdyson.cloud import DysonAccount

from libdyson.cloud.account import DysonAccountCN

from libdyson.exceptions import DysonOTPTooFrequently

print("Please choose your account region")

print("1: Mainland China")

print("2: Rest of the World")

region = input("Region [1/2]: ")

if region == "1":

account = DysonAccountCN()

mobile = input("Phone number: ")

verify = account.login\_mobile\_otp(f"+86{mobile}")

otp = input("Verification code: ")

verify(otp)

elif region == "2":

region = input("Region code: ")

account = DysonAccount()

email = input("Email: ")

password = getpass()

account.login\_email\_password(email, password, region)

else:

print(f"Invalid input {region}")

exit(1)

devices = account.devices()

for device in devices:

print()

print(f"Serial: {device.serial}")

print(f"Name: {device.name}")

print(f"Device Type: {device.product\_type}")

print(f"Credential: {device.credential}")

def log\_info(logger, data, msg):

log\_enabled = str(data.get("log\_enabled", "false"))

if log\_enabled.lower() == "true":

logger.debug(msg)

def log\_error(logger, msg):

logger.error(msg)

#Notify the error via persistent\_notification

domain = "persistent\_notification"

service = "create"

service\_data = {}

service\_data["notification\_id"] = "hass\_entities\_error"

service\_data["title"] = "\U000026A0 hass\_entities error"

service\_data["message"] = "{}".format(msg)

hass.services.call(domain, service, service\_data, False)

try:

#Execute requested action

script\_name = "hass\_entities.py"

action = data.get("action", "")

#Log start action execution

log\_info(logger, data, "Python Script: {} -> START of action: {}".format(script\_name, action))

if action == "":

log\_error(logger, "\*\*Required parameter 'action' is missing.\*\*\n\nScript NOT executed.")

elif action.lower() == "set\_state":

#Parameter -> action: set\_state (string, required)

#Parameter -> entity\_id (string, required)

#Parameter -> state (string, required)

#Parameter -> log\_enabled (bool)

#Getting entity

entity\_id = data.get("entity\_id", "")

if entity\_id == "":

log\_error(logger, "\*\*Required parameter 'entity\_id' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Getting original state and attributes

entity = hass.states.get(entity\_id)

if entity is None:

log\_error(logger, "\*\*Cannot find entity '{}'.\*\*\n\nAction '{}' NOT executed.".format(entity\_id, action.lower()))

else:

attributes = {}

for attr in entity.attributes:

attributes[attr] = entity.attributes.get(attr)

#Getting new state

new\_state = data.get("state", "")

if new\_state is None or new\_state == "":

log\_error(logger, "\*\*Required parameter 'state' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Setting new state

log\_info(logger, data, " Entity: '{}' -> New state: '{}'".format(entity\_id, new\_state))

hass.states.set(entity\_id, new\_state, attributes)

log\_info(logger, data, " DONE -> State set for entity '{}'".format(entity\_id))

elif action.lower() == "set\_attributes":

#Parameter -> action: set\_attributes (string, required)

#Parameter -> entity\_id (string, required)

#Parameter -> attributes (list, required)

#Parameter -> log\_enabled (bool)

#Getting entity

entity\_id = data.get("entity\_id", "")

if entity\_id == "":

log\_error(logger, "\*\*Required parameter 'entity\_id' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Getting original state and attributes

entity = hass.states.get(entity\_id)

if entity is None:

log\_error(logger, "\*\*Cannot find entity '{}'.\*\*\n\nAction '{}' NOT executed.".format(entity\_id, action.lower()))

else:

state = entity.state

attributes = {}

for attr in entity.attributes:

attributes[attr] = entity.attributes.get(attr)

#Getting new attributes

new\_attributes = data.get("attributes", "")

if new\_attributes is None or new\_attributes == "":

log\_error(logger, "\*\*Required parameter 'attributes' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Setting new attributes

for new\_attr in new\_attributes:

for k,v in new\_attr.items(): #This should iterate just once

log\_info(logger, data, " Entity: '{}' -> New attribute '{}': '{}'".format(entity\_id, k, v))

attributes[k] = v

hass.states.set(entity\_id, state, attributes)

log\_info(logger, data, " DONE -> Attributes set for entity '{}'".format(entity\_id))

elif action.lower() == "set\_state\_attributes":

#Parameter -> action: set\_state\_attributes (string, required)

#Parameter -> entity\_id (string, required)

#Parameter -> state (string, required)

#Parameter -> attributes (list, required)

#Parameter -> log\_enabled (bool)

#Getting entity

entity\_id = data.get("entity\_id", "")

if entity\_id == "":

log\_error(logger, "\*\*Required parameter 'entity\_id' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Getting original state and attributes

entity = hass.states.get(entity\_id)

if entity is None:

log\_error(logger, "\*\*Cannot find entity '{}'.\*\*\n\nAction '{}' NOT executed.".format(entity\_id, action.lower()))

else:

state = entity.state

attributes = {}

for attr in entity.attributes:

attributes[attr] = entity.attributes.get(attr)

#Getting new state

new\_state = data.get("state", "")

if new\_state is None or new\_state == "":

log\_error(logger, "\*\*Required parameter 'state' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

state = new\_state

#Setting new state

log\_info(logger, data, " Entity: '{}' -> New state: '{}'".format(entity\_id, new\_state))

#Getting new attributes

new\_attributes = data.get("attributes", "")

if new\_attributes is None or new\_attributes == "":

log\_error(logger, "\*\*Required parameter 'attributes' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Setting new attributes

for new\_attr in new\_attributes:

for k,v in new\_attr.items(): #This should iterate just once

log\_info(logger, data, " Entity: '{}' -> New attribute '{}': '{}'".format(entity\_id, k, v))

attributes[k] = v

hass.states.set(entity\_id, state, attributes)

log\_info(logger, data, " DONE -> State & attributes set for entity '{}'".format(entity\_id))

elif action.lower() == "delete\_attribute":

#Parameter -> action: delete\_attribute (string, required)

#Parameter -> entity\_id (string, required)

#Parameter -> attribute (string, required)

#Parameter -> log\_enabled (bool)

#Getting entity

entity\_id = data.get("entity\_id", "")

if entity\_id == "":

log\_error(logger, "\*\*Required parameter 'entity\_id' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

#Getting original state and attributes

entity = hass.states.get(entity\_id)

if entity is None:

log\_error(logger, "\*\*Cannot find entity '{}'.\*\*\n\nAction '{}' NOT executed.".format(entity\_id, action.lower()))

else:

state = entity.state

attributes = {}

#Getting attribute to delete

del\_attribute = data.get("attribute", "")

if del\_attribute == "":

log\_error(logger, "\*\*Required parameter 'attribute' is missing.\*\*\n\nAction '{}' NOT executed.".format(action.lower()))

else:

for attr in entity.attributes:

if attr != del\_attribute:

attributes[attr] = entity.attributes.get(attr)

#Setting attributes

hass.states.set(entity\_id, state, attributes)

log\_info(logger, data, " DONE -> Attribute '{}' deleted from entity '{}'".format(del\_attribute, entity\_id))

else:

log\_error(logger, "\*\*Invalid action provided ('{}').\*\*\n\nExpected: 'set\_state', 'set\_attributes', 'set\_state\_attributes', 'delete\_attribute'.".format(action))

#Log end action execution

log\_info(logger, data, "Python Script: {} -> END of action: {}".format(script\_name, action))

except Exception as e:

log\_error(logger, "\*\*An unhandled error has occurred.\*\*\n\n{}".format(e))

**Reference of dataset :**

[**https://tn.data.gov.in/resource/location-wise-daily-ambient-air-quality-tamil-nadu-year-2014**](https://tn.data.gov.in/resource/location-wise-daily-ambient-air-quality-tamil-nadu-year-2014)

**Conclusion :**

Load the data using pandas. Explore the dataset to understand its characteristics. Preprocess the data, including handling missing values, renaming columns, converting data types, and feature engineering. Optionally, save the preprocessed data for future use.