!pip install gspread

import gspread

from google.auth import default

# Authenticate and authorize access

creds, \_ = default()

gc = gspread.authorize(creds)

import gspread

import pandas as pd

from oauth2client.service\_account import ServiceAccountCredentials

# Define the scope for Google Sheets and Google Drive APIs

scope = ["https://spreadsheets.google.com/feeds", "https://www.googleapis.com/auth/drive"]

# Load credentials from the JSON key file

creds = ServiceAccountCredentials.from\_json\_keyfile\_name("/content/glossy-premise-408811-fa45e81faf35.json", scope)

# Authenticate with Google Sheets

gc = gspread.authorize(creds)

# Open the Google Sheet by URL

sheet\_url = 'https://docs.google.com/spreadsheets/d/1PXHUDPv1E5qPUqin0HKNC6h-UrF0j1\_RJ9BnoS20gic/edit?gid=0#gid=0'

sheet = gc.open\_by\_url(sheet\_url).sheet1

# Fetch all records

data = sheet.get\_all\_records()

# Load data into a pandas DataFrame

df = pd.DataFrame(data)

# Display the first few rows

print(df.head())

import pandas as pd

def fetch\_data(sheet):

    # Fetch all records from Google Sheets

    data = sheet.get\_all\_records()

    # Convert the data to a DataFrame

    df = pd.DataFrame(data)

    # Optionally, convert the Timestamp column to datetime format

    if 'Timestamp' in df.columns:

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

    return df

import time

# Fetch the initial data

df = fetch\_data(sheet)

print("Initial data:")

print(df)

# Loop to continuously fetch updated data

while True:

    # Fetch the latest data

    df = fetch\_data(sheet)

    # Clear previous output and display the updated data

    from IPython.display import clear\_output

    clear\_output(wait=True)

    # Display the latest data

    print("Updated data:")

    print(df.tail())  # Show only the last few rows for clarity

    # Add a delay to match the ESP8266 update frequency (e.g., 5 seconds)

    time.sleep(5)

# Assuming `sheet` is already defined and authorized in Colab

import pandas as pd

# Fetch data

data = sheet.get\_all\_records()

df = pd.DataFrame(data)

# Convert timestamp to datetime format if not done already

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

def classify\_air\_quality(pm2\_5, pm10):

    if pm2\_5 <= 30 and pm10 <= 50:

        return 'Good'

    elif pm2\_5 <= 60 and pm10 <= 100:

        return 'Satisfactory'

    elif pm2\_5 <= 90 and pm10 <= 250:

        return 'Moderately Polluted'

    elif pm2\_5 <= 120 and pm10 <= 350:

        return 'Poor'

    elif pm2\_5 <= 250 and pm10 <= 430:

        return 'Very Poor'

    else:

        return 'Severe'

import pandas as pd

from sklearn.ensemble import IsolationForest, RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import classification\_report, accuracy\_score

import matplotlib.pyplot as plt

import matplotlib.dates as mdates

import numpy as np

# Define the classify\_air\_quality function

def classify\_air\_quality(pm2\_5, pm10):

    if pm2\_5 <= 30 and pm10 <= 50:

        return 'Good'

    elif pm2\_5 <= 60 and pm10 <= 100:

        return 'Satisfactory'

    elif pm2\_5 <= 90 and pm10 <= 250:

        return 'Moderately Polluted'

    elif pm2\_5 <= 120 and pm10 <= 350:

        return 'Poor'

    elif pm2\_5 <= 250 and pm10 <= 430:

        return 'Very Poor'

    else:

        return 'Severe'

# Define a function to perform anomaly detection and classification in Colab without modifying the sheet

def perform\_analysis(df):

    # Apply classification to each row

    df['AirQualityCategory'] = df.apply(lambda row: classify\_air\_quality(row['PM2\_5'], row['PM10']), axis=1)

    # Anomaly Detection using Isolation Forest

    features = df[['PM1\_0', 'PM2\_5', 'PM10']]

    iso\_forest = IsolationForest(contamination=0.05, random\_state=42)

    anomalies = iso\_forest.fit\_predict(features)

    df['Anomaly'] = np.where(anomalies == -1, 'Yes', 'No')  # Label anomalies as 'Yes' or 'No'

    # Train a classifier for Air Quality Category

    X = df[['PM1\_0', 'PM2\_5', 'PM10']]

    y = df['AirQualityCategory']

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

    # Train Random Forest Classifier

    classifier = RandomForestClassifier(random\_state=42)

    classifier.fit(X\_train, y\_train)

    y\_pred = classifier.predict(X\_test)

    # Accuracy and Classification Report

    accuracy = accuracy\_score(y\_test, y\_pred)

    print(f"Model Accuracy: {accuracy \* 100:.2f}%\n")

    print("Classification Report:")

    print(classification\_report(y\_test, y\_pred))

    # Summary of Results

    print("\n### Summary of Air Quality Analysis ###")

    print(f"Total data points: {len(df)}")

    print(f"Anomalies detected: {df['Anomaly'].value\_counts()['Yes']} out of {len(df)} ({(df['Anomaly'].value\_counts()['Yes'] / len(df) \* 100):.2f}%)")

    print("\n### Air Quality Category Distribution ###")

    category\_counts = df['AirQualityCategory'].value\_counts()

    for category, count in category\_counts.items():

        print(f"{category}: {count} ({(count / len(df) \* 100):.2f}%)")

    # Select random 50% of the data while ensuring anomalies are included

    anomalies\_df = df[df['Anomaly'] == 'Yes']

    non\_anomalies\_df = df[df['Anomaly'] == 'No'].sample(frac=0.5, random\_state=42)

    report\_df = pd.concat([anomalies\_df, non\_anomalies\_df]).sample(frac=1, random\_state=42).reset\_index(drop=True)

    # Display a formatted table

    display\_formatted\_table(report\_df)

    # Generate individual plots for PM1.0, PM2.5, and PM10

    plot\_pm\_levels(df, 'PM1\_0', 'PM1.0', '#8E44AD', '#E74C3C', 'PM1.0')

    plot\_pm\_levels(df, 'PM2\_5', 'PM2.5', '#2980B9', '#F39C12', 'PM2.5')

    plot\_pm\_levels(df, 'PM10', 'PM10', '#27AE60', '#D35400', 'PM10')

# Define a function to display the report as a beautifully formatted table

def display\_formatted\_table(report\_df):

    # Apply styles to highlight anomalies and color-code categories

    def color\_anomalies(val):

        return 'background-color: #ffcccc' if val == 'Yes' else ''

    def color\_categories(val):

        colors = {

            'Good': '#d4edda',

            'Satisfactory': '#c3e6cb',

            'Moderately Polluted': '#ffeeba',

            'Poor': '#f5c6cb',

            'Very Poor': '#f8d7da',

            'Severe': '#f5b7b1'

        }

        return f'background-color: {colors.get(val, "")}'

    styled\_df = report\_df.style.applymap(color\_anomalies, subset=['Anomaly']).applymap(color\_categories, subset=['AirQualityCategory']) \

        .set\_properties(\*\*{'border': '1px solid black', 'padding': '5px'}).set\_table\_styles(

        [{'selector': 'th', 'props': [('background-color', '#f2f2f2'), ('color', '#333'), ('font-weight', 'bold')]}]

    ).set\_caption("Air Quality Monitoring Report (Random 50% Sample)")

    # Display the styled DataFrame

    display(styled\_df)

# Function to create individual plots for each PM level

def plot\_pm\_levels(df, pm\_column, pm\_label, color\_line, color\_anomaly, title):

    plt.figure(figsize=(15, 6))

    # Plot PM level with line and markers

    plt.plot(df['Timestamp'], df[pm\_column], label=pm\_label, color=color\_line, marker='o', markersize=4, linestyle='-', linewidth=1.5)

    # Highlight anomalies

    plt.scatter(df.loc[df['Anomaly'] == 'Yes', 'Timestamp'], df.loc[df['Anomaly'] == 'Yes', pm\_column],

                color=color\_anomaly, label=f'{pm\_label} Anomalies', marker='x', s=100, zorder=5)

    # Enhance the plot aesthetics

    plt.title(f'{title} Levels with Anomaly Detection', fontsize=16, weight='bold')

    plt.xlabel('Timestamp', fontsize=14)

    plt.ylabel('Concentration (µg/m³)', fontsize=14)

    plt.grid(visible=True, color='gray', linestyle='--', linewidth=0.5, alpha=0.7)

    plt.xticks(rotation=45)

    plt.yticks(fontsize=12)

    # Format x-axis for dates for better readability

    plt.gca().xaxis.set\_major\_locator(mdates.HourLocator(interval=1))  # Adjust interval as needed

    plt.gca().xaxis.set\_major\_formatter(mdates.DateFormatter('%H:%M:%S'))

    # Add a legend with a styled background

    plt.legend(loc='upper right', fontsize=12, frameon=True, fancybox=True, framealpha=0.9, shadow=True, borderpad=1)

    # Show the plot with improved style

    plt.tight\_layout()

    plt.show()

# Fetch data from Google Sheets (assuming `sheet` is already defined)

data = sheet.get\_all\_records()

df = pd.DataFrame(data)

# Convert timestamp to datetime if necessary

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

# Run the analysis

perform\_analysis(df)

import gspread

import pandas as pd

import numpy as np

from oauth2client.service\_account import ServiceAccountCredentials

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import confusion\_matrix, classification\_report

from sklearn.preprocessing import LabelEncoder

import matplotlib.pyplot as plt

import seaborn as sns

# -------------------------------------------------------------------

# 1. Connect to Google Sheets and Fetch Data

# -------------------------------------------------------------------

def get\_dataframe\_from\_sheets(json\_keyfile\_path, sheet\_url):

    """

    Connect to Google Sheets using a service account JSON keyfile and fetch data.

    Returns a pandas DataFrame with a parsed Timestamp column.

    """

    scope = ["https://spreadsheets.google.com/feeds",

             "https://www.googleapis.com/auth/drive"]

    creds = ServiceAccountCredentials.from\_json\_keyfile\_name(json\_keyfile\_path, scope)

    gc = gspread.authorize(creds)

    sheet = gc.open\_by\_url(sheet\_url).sheet1  # or select another worksheet if needed

    data = sheet.get\_all\_records()

    df = pd.DataFrame(data)

    # Convert Timestamp column to datetime if it exists

    if 'Timestamp' in df.columns:

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

    return df

# -------------------------------------------------------------------

# 2. Define Air Quality Classification Logic

# -------------------------------------------------------------------

def classify\_air\_quality(pm2\_5, pm10):

    """

    Returns a string label for air quality based on PM2.5 and PM10.

    """

    if pm2\_5 <= 30 and pm10 <= 50:

        return 'Good'

    elif pm2\_5 <= 60 and pm10 <= 100:

        return 'Satisfactory'

    elif pm2\_5 <= 90 and pm10 <= 250:

        return 'Moderately Polluted'

    elif pm2\_5 <= 120 and pm10 <= 350:

        return 'Poor'

    elif pm2\_5 <= 250 and pm10 <= 430:

        return 'Very Poor'

    else:

        return 'Severe'

# -------------------------------------------------------------------

# 3. Train a Random Forest Classifier

# -------------------------------------------------------------------

def train\_air\_quality\_model(df):

    """

    - Creates an 'AirQualityCategory' column using classify\_air\_quality.

    - Trains a Random Forest classifier to predict the category from PM1\_0, PM2\_5, PM10.

    - Returns the trained model, X\_test, y\_test, and predictions.

    """

    # Ensure PM columns exist

    for col in ['PM1\_0', 'PM2\_5', 'PM10']:

        if col not in df.columns:

            raise ValueError(f"Missing column: {col}")

    # Create the label

    df['AirQualityCategory'] = df.apply(lambda row: classify\_air\_quality(row['PM2\_5'], row['PM10']), axis=1)

    # Prepare features (X) and target (y)

    X = df[['PM1\_0', 'PM2\_5', 'PM10']].copy()

    y = df['AirQualityCategory'].copy()

    # Encode string labels numerically for classification

    encoder = LabelEncoder()

    y\_encoded = encoder.fit\_transform(y)

    # Split into train/test sets

    X\_train, X\_test, y\_train, y\_test = train\_test\_split(

        X, y\_encoded, test\_size=0.2, random\_state=42

    )

    # Train a Random Forest

    rf = RandomForestClassifier(n\_estimators=100, random\_state=42)

    rf.fit(X\_train, y\_train)

    # Predict on the test set

    y\_pred = rf.predict(X\_test)

    return rf, X\_test, y\_test, y\_pred, encoder

def plot\_six\_figures(df, model, X\_test, y\_test, y\_pred, encoder):

    """

    Produce six separate plots to give a comprehensive overview of data and model performance.

    """

    # --------------------------------------------

    # A. Prepare Data for Plotting

    # --------------------------------------------

    # Create a combined DataFrame for test data

    results\_df = X\_test.copy()

    results\_df['ActualCategory'] = encoder.inverse\_transform(y\_test)

    results\_df['PredictedCategory'] = encoder.inverse\_transform(y\_pred)

    # Higher DPI for HD clarity

    plt.rcParams['figure.dpi'] = 150

    # --------------------------------------------

    # Plot 1: Time-series of PM Levels (if Timestamp available)

    # --------------------------------------------

    if 'Timestamp' in df.columns:

        plt.figure(figsize=(10, 5))

        plt.plot(df['Timestamp'], df['PM1\_0'], label='PM1.0', color='blue')

        plt.plot(df['Timestamp'], df['PM2\_5'], label='PM2.5', color='red')

        plt.plot(df['Timestamp'], df['PM10'], label='PM10', color='green')

        plt.xlabel('Timestamp')

        plt.ylabel('Concentration (µg/m³)')

        plt.title('Time-Series of PM1.0, PM2.5, and PM10')

        plt.legend()

        plt.tight\_layout()

        plt.show()

    # --------------------------------------------

    # Plot 2: Distribution of PM1.0, PM2.5, PM10

    # --------------------------------------------

    plt.figure(figsize=(10, 5))

    sns.histplot(df['PM1\_0'], color='blue', kde=True, label='PM1.0', alpha=0.5)

    sns.histplot(df['PM2\_5'], color='red', kde=True, label='PM2.5', alpha=0.5)

    sns.histplot(df['PM10'], color='green', kde=True, label='PM10', alpha=0.5)

    plt.title('Distribution of PM1.0, PM2.5, and PM10')

    plt.xlabel('Concentration (µg/m³)')

    plt.ylabel('Count')

    plt.legend()

    plt.tight\_layout()

    plt.show()

    # --------------------------------------------

    # Plot 3: Correlation Heatmap

    # --------------------------------------------

    plt.figure(figsize=(6, 5))

    corr\_matrix = df[['PM1\_0','PM2\_5','PM10']].corr()

    sns.heatmap(corr\_matrix, annot=True, cmap='Blues', fmt=".2f")

    plt.title('Correlation Heatmap for PM Values')

    plt.tight\_layout()

    plt.show()

    # --------------------------------------------

    # Plot 4: Confusion Matrix

    # --------------------------------------------

    cm = confusion\_matrix(y\_test, y\_pred)

    plt.figure(figsize=(6, 5))

    sns.heatmap(cm, annot=True, fmt='d', cmap='Purples',

                xticklabels=encoder.classes\_,

                yticklabels=encoder.classes\_)

    plt.title('Confusion Matrix')

    plt.xlabel('Predicted Category')

    plt.ylabel('Actual Category')

    plt.tight\_layout()

    plt.show()

    # --------------------------------------------

    # Plot 5: Feature Importance (Random Forest)

    # --------------------------------------------

    plt.figure(figsize=(6, 5))

    importances = model.feature\_importances\_

    feature\_names = X\_test.columns

    sns.barplot(x=importances, y=feature\_names, palette='viridis')

    plt.title('Feature Importance in Random Forest')

    plt.xlabel('Relative Importance')

    plt.tight\_layout()

    plt.show()

    # --------------------------------------------

    # Plot 6: Actual vs Predicted Category Counts

    # --------------------------------------------

    plt.figure(figsize=(8, 5))

    actual\_counts = results\_df['ActualCategory'].value\_counts()

    pred\_counts = results\_df['PredictedCategory'].value\_counts()

    # Ensure all categories appear in both

    all\_cats = sorted(set(encoder.classes\_))

    actual\_vals = [actual\_counts.get(cat, 0) for cat in all\_cats]

    pred\_vals = [pred\_counts.get(cat, 0) for cat in all\_cats]

    x = np.arange(len(all\_cats))

    width = 0.35

    plt.bar(x - width/2, actual\_vals, width, label='Actual', color='steelblue')

    plt.bar(x + width/2, pred\_vals, width, label='Predicted', color='darkorange')

    plt.xticks(x, all\_cats, rotation=45)

    plt.title('Actual vs. Predicted Category Counts')

    plt.xlabel('Air Quality Category')

    plt.ylabel('Count')

    plt.legend()

    plt.tight\_layout()

    plt.show()

def main():

    # Update the path to your JSON keyfile and the URL of your Google Sheet

    json\_keyfile\_path = "/content/glossy-premise-408811-fa45e81faf35.json"

    sheet\_url = "https://docs.google.com/spreadsheets/d/1PXHUDPv1E5qPUqin0HKNC6h-UrF0j1\_RJ9BnoS20gic/edit?usp=sharing"

    # 1) Fetch Data

    df = get\_dataframe\_from\_sheets(json\_keyfile\_path, sheet\_url)

    print("Data Snapshot:")

    print(df.head())

    # 2) Train Model

    model, X\_test, y\_test, y\_pred, encoder = train\_air\_quality\_model(df)

    # 3) Display Evaluation

    print("\nClassification Report:")

    print(classification\_report(y\_test, y\_pred, target\_names=encoder.classes\_))

    # 4) Generate 6 HD Plots

    plot\_six\_figures(df, model, X\_test, y\_test, y\_pred, encoder)

# Run the end-to-end pipeline (comment out if you only want to import functions above)

if \_\_name\_\_ == "\_\_main\_\_":

    main()

import pandas as pd

import numpy as np

from sklearn.ensemble import RandomForestClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.preprocessing import LabelEncoder

from sklearn.metrics import classification\_report

def classify\_air\_quality(pm2\_5, pm10):

    if pm2\_5 <= 30 and pm10 <= 50:

        return 'Good'

    elif pm2\_5 <= 60 and pm10 <= 100:

        return 'Satisfactory'

    elif pm2\_5 <= 90 and pm10 <= 250:

        return 'Moderately Polluted'

    elif pm2\_5 <= 120 and pm10 <= 350:

        return 'Poor'

    elif pm2\_5 <= 250 and pm10 <= 430:

        return 'Very Poor'

    else:

        return 'Severe'

# Create the AirQualityCategory column

df['AirQualityCategory'] = df.apply(lambda row: classify\_air\_quality(row['PM2\_5'], row['PM10']), axis=1)

# Prepare features and target

X = df[['PM1\_0', 'PM2\_5', 'PM10']]

y = df['AirQualityCategory']

# Encode the target labels into numbers

encoder = LabelEncoder()

y\_encoded = encoder.fit\_transform(y)

# Split the data into training and testing sets

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y\_encoded, test\_size=0.2, random\_state=42

)

# Train a Random Forest Classifier

rf = RandomForestClassifier(n\_estimators=100, random\_state=42)

rf.fit(X\_train, y\_train)

# Make predictions on the test set

y\_pred = rf.predict(X\_test)

# Create a DataFrame to display actual vs predicted values

results\_df = X\_test.copy()

results\_df['ActualCategory'] = encoder.inverse\_transform(y\_test)

results\_df['PredictedCategory'] = encoder.inverse\_transform(y\_pred)

print("Predicted vs. Actual Values (Test Set):")

print(results\_df.head())

print("\nClassification Report:")

print(classification\_report(y\_test, y\_pred, target\_names=encoder.classes\_))