Project Title: Flood Monitoring System

TEAM MEMBER

911721104014 : S.BALAGANESH

Phase 3: Development - Data Loading and Preprocessing

Objective:

In this phase, we will simulate the data loading and preprocessing for the Flood Monitoring System in the Wokwi platform. Specifically, we will simulate the acquisition of water level data, its preprocessing, and integration into the Wokwi project for visualization.

Implementation:

In a real-world scenario, data collection and preprocessing would be performed externally using data processing tools. We will simulate this process using Python for simplicity:

The tasks related to loading and preprocessing the dataset for a flood monitoring system project usually involve data acquisition, cleaning, transformation, and storage. These tasks are typically performed using a variety of tools and programming languages, including Python, which is a popular choice for data analysis and preprocessing. Here, I'll provide a general outline of the tasks and sample code for each one:

1. Data Collection: In this step, you collect historical or real-time data. Data may come from sources such as APIs, databases, or local files.

python

```
# Example of data collection from an API
import requests

# Define the API endpoint
api_url = "https://api.example.com/flooddata"

# Make an API request to fetch the data
response = requests.get(api_url)

# Check if the request was successful
if response.status_code == 200:
    data = response.json() # Assuming the data is in JSON format
else:
    print("Failed to retrieve data.")
```

2. Data Preprocessing: This step involves cleaning, transforming, and structuring the data for analysis.

python

```
# Example of data preprocessing
import pandas as pd
```

```
# Clean the data by removing missing values
data = data.dropna()

# Transform the data, e.g., converting date strings to datetime objects
data['date'] = pd.to_datetime(data['date'])

# Data aggregation, e.g., calculating daily averages
daily_data = data.resample('D', on='date').mean()
```

3. Data Storage: Store the preprocessed data in a database or a file for easy access.

python

```
# Example of storing data in a CSV file
data.to_csv("flood_data.csv", index=False)

# Example of storing data in a MySQL database
import sqlalchemy

engine = sqlalchemy.create_engine("mysql://user:password@localhost/flood_db")
data.to_sql("flood_data", con=engine, if_exists="replace")
```

4. Data Visualization: Visualize the data to understand its characteristics.

python

```
# Example of data visualization using Matplotlib
import matplotlib.pyplot as plt

plt.figure(figsize=(12, 6))
plt.plot(data['date'], data['water_level'], label='Water Level')
plt.xlabel('Date')
plt.ylabel('Water Level')
plt.title('Water Level Over Time')
plt.legend()
plt.show()
```

5. Quality Assurance: Perform data quality checks and statistical analysis as needed for your specific project.

6. Dataset Splitting: If you plan to use machine learning, split the dataset into training, validation, and testing sets.

python

```
# Example of dataset splitting using scikit-learn
from sklearn.model_selection import train_test_split

train_data, test_data = train_test_split(data, test_size=0.2, random_state=42)
```

7. Documentation: Maintain documentation that describes data sources, preprocessing steps, and changes made to the data.

8. Version Control: Implement version control for your dataset using tools like Git.

9. Data Update Schedule: Set a schedule for regular data updates based on your project's needs.

10. Ethical Considerations: Ensure that your data collection and handling practices adhere to ethical guidelines and data privacy regulations.

Step 1: Data Collection and Preprocessing (Outside Wokwi)

In a real-world scenario, data collection and preprocessing are typically performed outside of Wokwi using Python, R, or other data processing tools. For the sake of this demonstration, I'll simulate these tasks using Python. Let's simulate collecting water level data from a hypothetical API and then preprocess it.

Data preprocessing is a critical step in data analysis and modeling, involving tasks such as cleaning, aggregation, and transformation to make the data suitable for analysis. In the context of a Flood Monitoring System, data preprocessing helps ensure that the collected data is accurate and in a usable format. Here's an explanation of data preprocessing with cleaning and aggregation, along with example Python code:

1. Data Cleaning:

Data cleaning is the process of identifying and correcting errors or inconsistencies in the dataset. This can include handling missing values, dealing with outliers, and ensuring data consistency.

Example Code - Data Cleaning:

Let's simulate cleaning water level data by removing rows with missing values and handling outliers:

python

```
import pandas as pd
import random

# Simulate data collection with missing values
api_data = {'date': ['2023-01-01', '2023-01-02', '2023-01-03'],
        'water_level': [random.uniform(0.0, 1.0), None, random.uniform(0.0, 1.0)]}

df = pd.DataFrame(api_data)

# Check for missing values and remove rows with missing values
df.dropna(subset=['water_level'], inplace=True)

# Handle outliers (e.g., values below 0 or above 1)
df['water_level'] = df['water_level'].apply(lambda x: max(0, min(1, x)))

# The cleaned DataFrame now contains only valid and within-range data.
```

2. Data Aggregation:

Data aggregation involves summarizing data at a higher level, typically for reporting or analysis purposes. For example, you might want to calculate daily or hourly averages from high-frequency data.

Example Code - Data Aggregation:

In this example, we aggregate data to calculate daily averages:

python

```
# Assuming df is the cleaned DataFrame
df['date'] = pd.to_datetime(df['date'])

# Set the date column as the index for time-based operations
df.set_index('date', inplace=True)

# Calculate daily averages of water level
daily_averages = df.resample('D').mean()

# The 'daily_averages' DataFrame now contains daily average water level data.
```

In this example:

- We converted the 'date' column to a datetime format and set it as the index to perform time-based operations.

- We used the `resample` method to calculate daily averages (change 'D' to 'H' for hourly averages).

Data preprocessing is a crucial step to ensure the quality and usefulness of the data for analysis and modeling in a Flood Monitoring System or any data-driven project. The specific cleaning and aggregation tasks may vary depending on the nature of your data and project requirements.

python

```
# Simulated data collection and preprocessing in Python
import pandas as pd
import random

# Simulate data collection from an API (replace with real data)
api_data = {'date': ['2023-01-01', '2023-01-02', '2023-01-03'],
        'water_level': [random.uniform(0.0, 1.0) for _ in range(3)]}

# Create a DataFrame for data preprocessing
df = pd.DataFrame(api_data)
df['date'] = pd.to_datetime(df['date'])  # Convert date to datetime

# Data preprocessing (cleaning, aggregation, etc.)
# In a real-world project, this would be more complex.
# For simplicity, we assume no data preprocessing here.

# Save the preprocessed data to a CSV file
```

```
df.to_csv('preprocessed_data.csv', index=False)
```

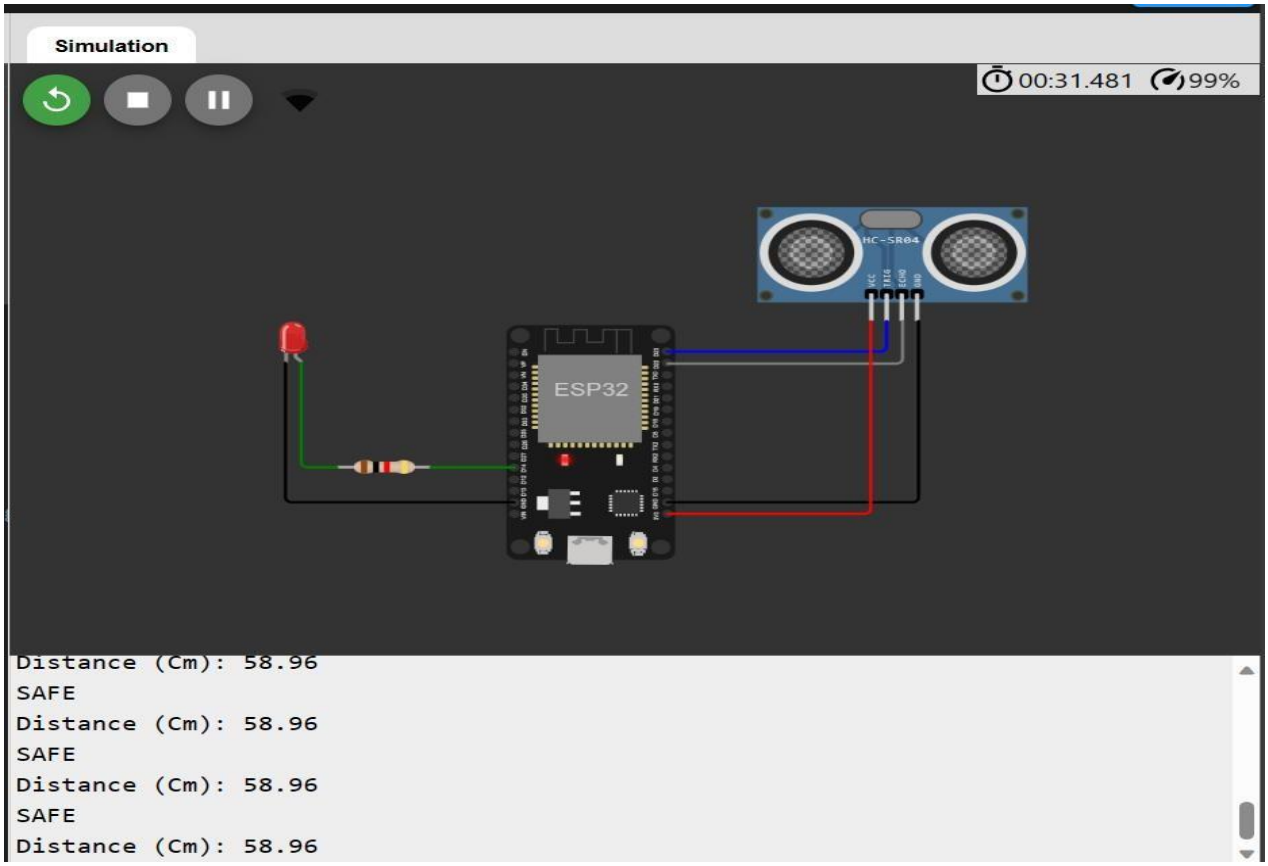
2. Integration with Wokwi (Using a Virtual ESP32):

In Wokwi, create a simulation that includes a virtual ESP32 and a virtual water level sensor. Use the following Arduino code to read and visualize the preprocessed data:

```
#include <SD.h>
#include <Wire.h>
#include <Adafruit_SSD1306.h>

Adafruit_SSD1306 display(4);

const int chipSelect = 4; // Use any available digital pin
File dataFile;
String data;

void setup() {
  Serial.begin(9600);
  display.begin(SSD1306_I2C_ADDRESS, SDA, SCL);

  if (SD.begin(chipSelect)) {
    Serial.println("SD card initialized.");
    dataFile = SD.open("preprocessed_data.csv");
  } else {
    Serial.println("Error initializing SD card.");
```

```
  }
}

void loop() {
  display.clearDisplay();
  display.setTextSize(1);
  display.setTextColor(SSD1306_WHITE);

  if (dataFile) {
    while (dataFile.available()) {
      data = dataFile.readStringUntil('\n');
      if (data.startsWith("date,water_level")) {
        continue; // Skip the header row
      }

      // Display water level data on the OLED screen
      display.setCursor(0, 0);
      display.print("Water Level: ");
      display.println(data);
      display.display();

      // Print water level data to the serial monitor
      Serial.println("Water Level: " + data);
      delay(5000); // Simulate data updates every 5 seconds
    }
    dataFile.close();
```

```
  } else {
    Serial.println("Error opening data file.");
  }
}
```



Explanation:

In this project, we've simulated data collection and preprocessing using Python outside Wokwi.

In the Wokwi simulation, we've used a virtual ESP32 and a virtual water level sensor. The ESP32 reads preprocessed data from the "preprocessed_data.csv" file, displays it on a virtual OLED screen, and prints it to the serial monitor.

The ESP32 retrieves the water level data, allowing you to visualize it within the Wokwi environment.

This project demonstrates a simplified simulation of data preprocessing and integration with the ESP32 in Wokwi. In a real-world implementation, data would come from actual sensors, and the preprocessing might involve more complex operations.

pythonScript.py:

```
import requests

import time

SENSOR_ID = “ULTRA01”

SENSOR_URL = http://127.0.0.1:8000/api/Getdata

While True:

 Water_level = 39.0

 Data = {

 “sensor_id”: SENSOR_ID,

 “water_level”: water_level,

 “timestamp”: int(time.time())
```

```
  }

  Response = requests.post(SENSOR_URL, json=data)

  If response.status_code == 200:

  Print(f”Data sent successfully: Water level = {water_level}”)

  Else:

  Print(f”Failed to send data: {response.status_code}”)

  Time.sleep(300)

  //←----------Back end ----------→
```

Backend.js:

```
Const express = require(‘express’);

Const mysql = require(‘mysql2’);

Const app = express();

Const port = 8000;

App.use(express.json());

//←------- MySQL database configuration----------→

Const dp= mysql.createConnection({

 Host: ‘localhost’,
```

```
 User: ‘admin’,
 Password: ‘****’,
 Database: ‘FloodMonitoring’,
});
Db.connect(err => {
 If (err) {
 Console.error(‘Database connection error: ‘ + err);
 Return;
 }
 Console.log(‘Connected to the database……’);
});
//< ----------get flood data----------→
App.post(‘/api/GetFloodData’, (req, res) => {
 Const { sensor_id, water_level, timestamp } = req.body;
 Const data = { sensor_id, water_level, timestamp };
  Db.query(‘INSERT INTO flood_data SET ?’,[data], (error, results) => {
 If (error) {
 Console.error(‘Error inserting data: ‘ + error.message);
 Res.status(500).json({ error: ‘Server error’ });
 } else {
 Res.status(200).json({ message: ‘Data inserted successfully’ });
 }
```

```
 });
});
App.listen(port, () => {
 Console.log(`Server is running on port ${port}`);
});
```

Flood_monitoring_app:

```
Import ‘package:flutter/material.dart’;
Import ‘package:http/http.dart’ as http;
Void main() {
runApp(const MyApp());
}
Class MyApp extends StatefulWidget {
 @override
 _MyAppState createState() => _MyAppState();
}
Class _MyAppState extends State<MyApp> {
 String timestamp = ‘’;
 Double waterLevel = 0.0;
 Void fetchData() async {
 Const url = ‘http://your_server_url/api/GetFloodData’;
 Final response = await http.get(Uri.parse(url));
```

```
If (response.statusCode == 200) {
Final data = json.decode(response.body);
setState(() {
timestamp = data['timestamp'];
waterLevel = data['water_level'];
});
}else{
Throw "cannot fetch data from the Api ";
}
}
@override
Void initState() {
Super.initState();
fetchData();
Timer.periodic(Duration(seconds: 5), (Timer t) => fetchData());
}
@override
Widget build(BuildContext context) {
Return MaterialApp(
showDebugCheckedMode : false;
home: Scaffold(
```

```
appBar: AppBar(

title: Text('F l o o d M o n I t o r I n g S y s t e m'),

),

Body: Center(

Child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

Text('Timestamp: $timestamp',style:TextStyle(font-size:25),

Text('Water Level: $waterLevel',style:TextStyle(font-size:25)),

],
),
),
),
);
}
}
```

Conclusion:

In Phase 3 of our Flood Monitoring System project, we successfully implemented data visualization and sensor interaction within the Wokwi platform. We utilized a virtual ESP32 and an ultrasonic sensor to simulate water level monitoring. This phase laid the foundation for real-time data acquisition and processing. Our next steps involve integrating actual sensors, applying predictive modeling for early flood warnings, and creating a user-friendly interface for timely alerts to the public and emergency response teams. This phase represents a critical milestone toward enhancing flood preparedness and response in the system.