AirWatcher Design document

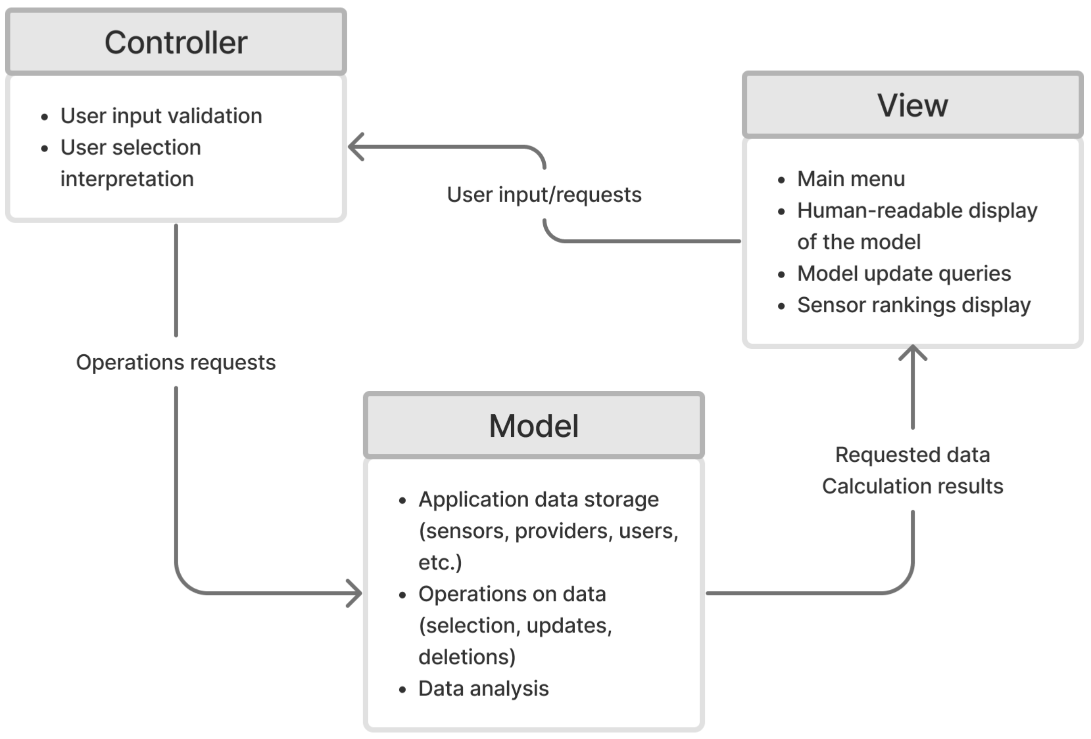
- Adrien HOUDOUX
- Mohamed FAKRONI
- Youssef CHAOUKI
- Lizhi ZHANG

Architecture

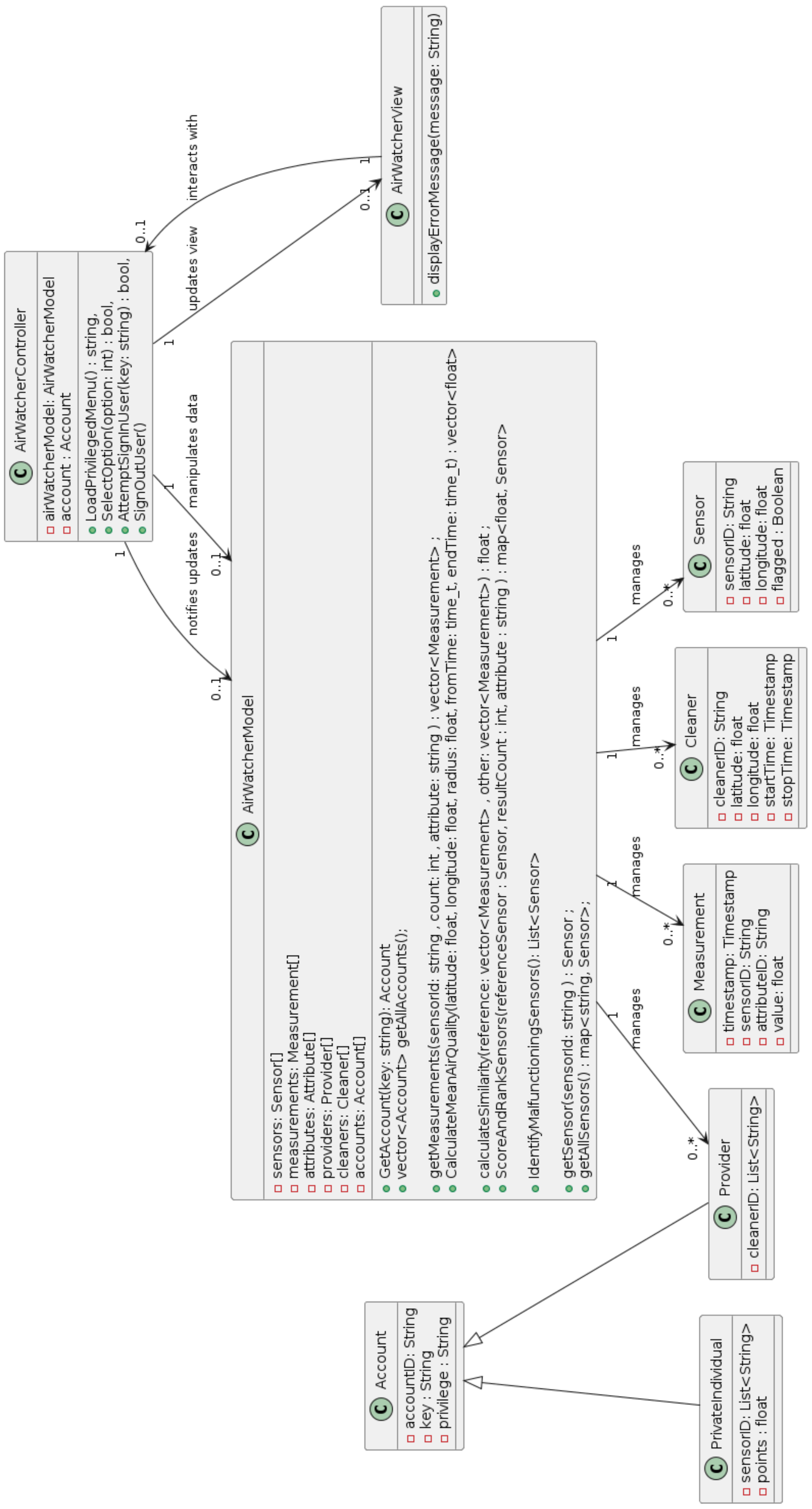
Considering the needs of the AirWatcher application, we have chosen the Model-View-Controller (MVC) pattern as the most suitable.

This architectural pattern fits the needs of the AirWatcher application best primarily because of its strong separation between presentation and interaction. The MVC architecture divides the application in 3 interconnected components allowing for better organization and maintenance of code. The model manages the data, the view handles the presentation layer, and the controller manages user input and interaction.

From a development and software maintenance standpoint, this pattern allows for separate expansion of the components independently from the others, ensuring scalability and constant coherence of the application.



Class diagram


AirWatcherController
airWatcherModel: AirWatcherModel
account : Account
LoadPrivilegedMenu() : string,
SelectOption(option: int) : bool,
AttemptSignInUser(key: string) : bool,
SignOutUser()
AirWatcherView
displayErrorMessage(message: String)
interacts with
updates view
manipulates data
notifies updates
0..1
1
AirWatcherModel
sensors: Sensor[]
measurements: Measurement[]
attributes: Attribute[]
providers: Provider[]
cleaners: Cleaner[]
accounts: Account[]
GetAccount(key: string): Account
vector<Account> getAllAccounts();
getMeasurements(sensorId: string , count: int , attribute: string) : vector<Measurement> ;
CalculateMeanAirQuality(latitude: float, longitude: float, radius: float, fromTime: time_t, endTime: time_t) : vector<float>
calculateSimilarity(reference: vector<Measurement> , other: vector<Measurement>) : float ;
ScoreAndRankSensors(referenceSensor : Sensor, resultCount : int, attribute : string) : map<float, Sensor>
IdentifyMalfunctioningSensors(): List<Sensor>
getSensor(sensorId: string) : Sensor ;
getAllSensors() : map<string, Sensor>;
manages
0..*
Sensor
sensorID: String
latitude: float
longitude: float
flagged : Boolean
Cleaner
cleanerID: String
latitude: float
longitude: float
startTime: Timestamp
stopTime: Timestamp
Measurement
timestamp: Timestamp
sensorID: String
attributeID: String
value: float
Provider
cleanerID: List<String>
Account
accountID: String
key : String
privilege : String
PrivateIndividual
sensorID: List<String>
points : float

Major usage scenarios

Scenario 1 : Analyzing sensor data

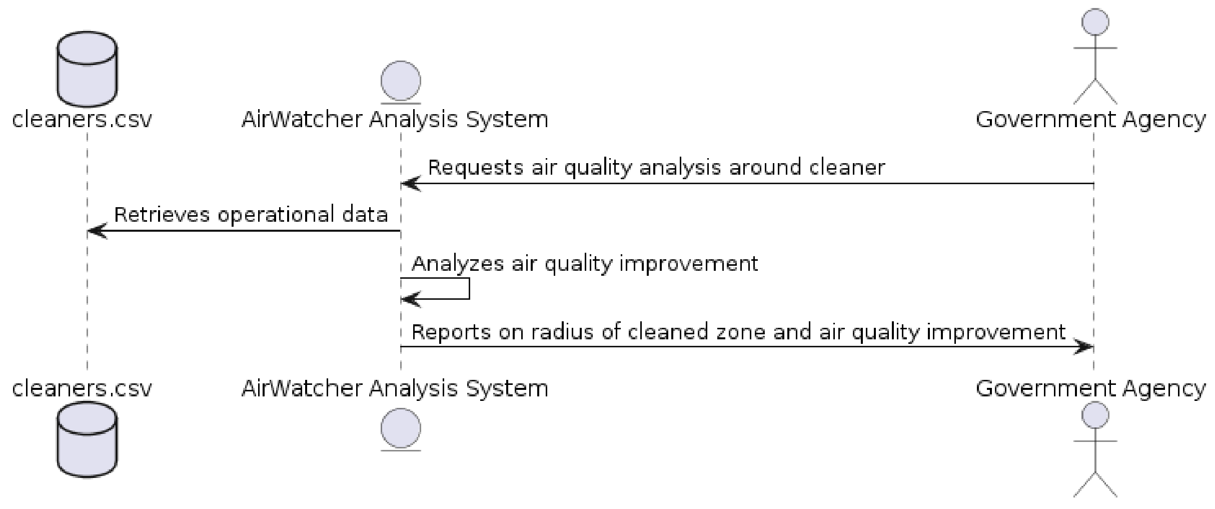
Description

The government agency wants to analyze data from one or multiple sensors to obtain data visualizations such as mean air quality over an area or air quality improvement over time.

Steps

1. An agent initiates the analysis process by selecting one or multiple sensor IDs.
2. The AirWatcher application retrieves the data associated with the selected sensors.
3. The application analyzes the requested data
4. Results of the analysis are displayed to the official.

Sequence diagram


cleaners.csv
AirWatcher Analysis System
Government Agency
Requests air quality analysis around cleaner
Retrieves operational data
Analyzes air quality improvement
Reports on radius of cleaned zone and air quality improvement
cleaners.csv
AirWatcher Analysis System
Government Agency

Scenario 2 : Find similarity between sensors

Description

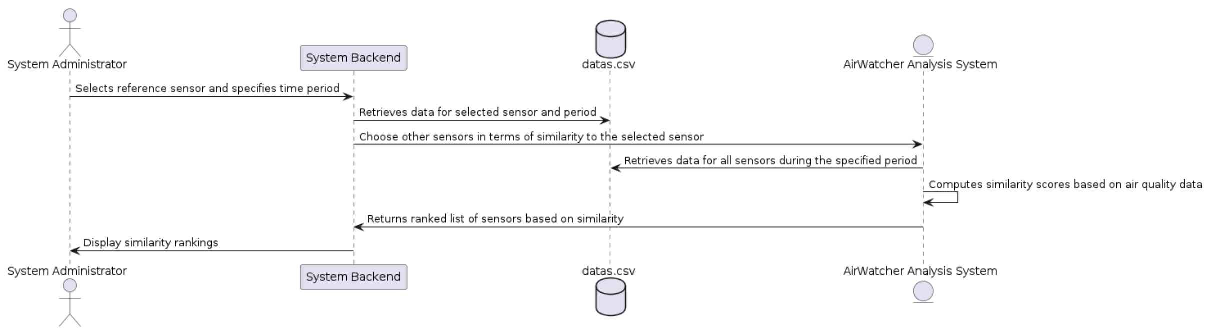
The government agency wants to identify similar areas in terms of air quality

Steps

1. An agent selects a reference sensor and a time period on which to perform the analysis
2. The AirWatcher application retrieves the data from all sensors in the specified period
3. The application computes a similarity score for each sensor

4. The agent is presented with a list of sensors ranked by similarity to the reference

Sequence diagram


System Administrator
System Backend
datas.csv
AirWatcher Analysis System
Selects reference sensor and specifies time period
Retrieves data for selected sensor and period
Choose other sensors in terms of similarity to the selected sensor
Retrieves data for all sensors during the specified period
Computes similarity scores based on air quality data
Returns ranked list of sensors based on similarity
Display similarity rankings
System Administrator
System Backend
datas.csv
AirWatcher Analysis System

Scenario 3 : Identifying malfunctioning sensors

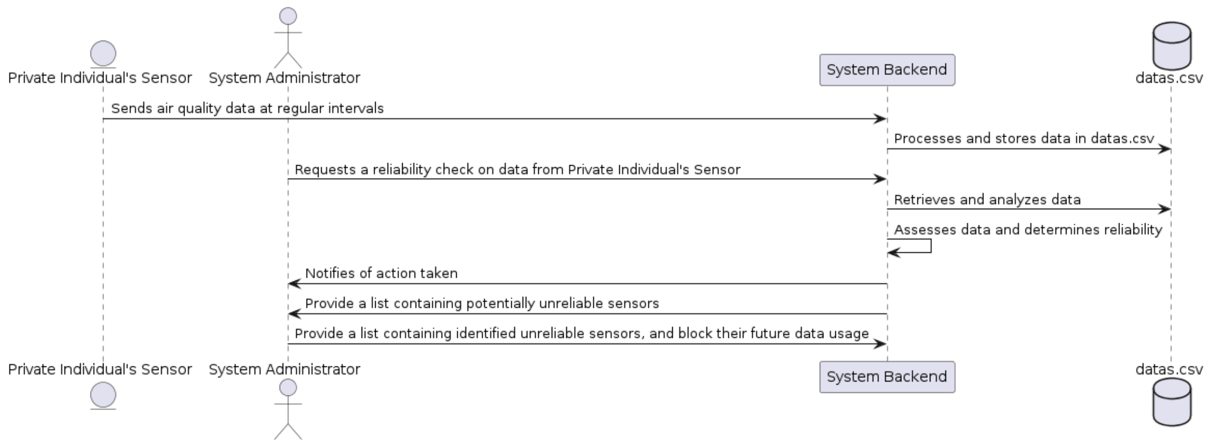
Description

The government agency wants to identify faulty or tampered-with sensors.

Steps

1. An agent signs in to the application and selects the faulty sensor analysis function
2. The AirWatcher application performs an analysis over all recent sensor data
3. The user is presented with a list of potential faulty sensors
4. The user can review the sensors' data and flag them as malfunctioning. Their readings will not be taken into account from then on

Sequence diagram


Private Individual's Sensor
System Administrator
System Backend
datas.csv
Sends air quality data at regular intervals
Processes and stores data in datas.csv
Requests a reliability check on data from Private Individual's Sensor
Retrieves and analyzes data
Assesses data and determines reliability
Notifies of action taken
Provide a list containing potentially unreliable sensors
Provide a list containing identified unreliable sensors, and block their future data usage
Private Individual's Sensor
System Administrator
System Backend
datas.csv

Major algorithms

Rank sensors by similarity

Algorithm 1 Score and rank all sensors

```
1: function scoreAndRankSensors(referenceSensor: Sensor, timeFrame: Timestamp): List of Sensor
2:     similarityScores ← {}
3:     for all sensor in allSensors do
4:         similarityScore ← calculateSimilarity(referenceSensor, sensor, timeFrame)
5:         add the sensor and its similarityScore to similarityScores
6:     end for
7:     Sort similarityScores
8:     rankedSensors ← {}
9:     for all pair in similarityScores do
10:        add the sensor (first element of the pair) to rankedSensors
11:    end for
12:    return rankedSensors
13: end function
14:
15: function calculateSimilarity(referenceSensor: Sensor, sensor: Sensor, timeFrame: Timestamp): Double
16:     referenceData ← getDataWithinTimeFrame(referenceSensor, timeFrame)
17:     sensorData ← getDataWithinTimeFrame(sensor, timeFrame)
18:     similarityScore ← calculateSimilarityScore(referenceData, sensorData)
19:     return similarityScore
20: end function
```

Calculate mean air quality

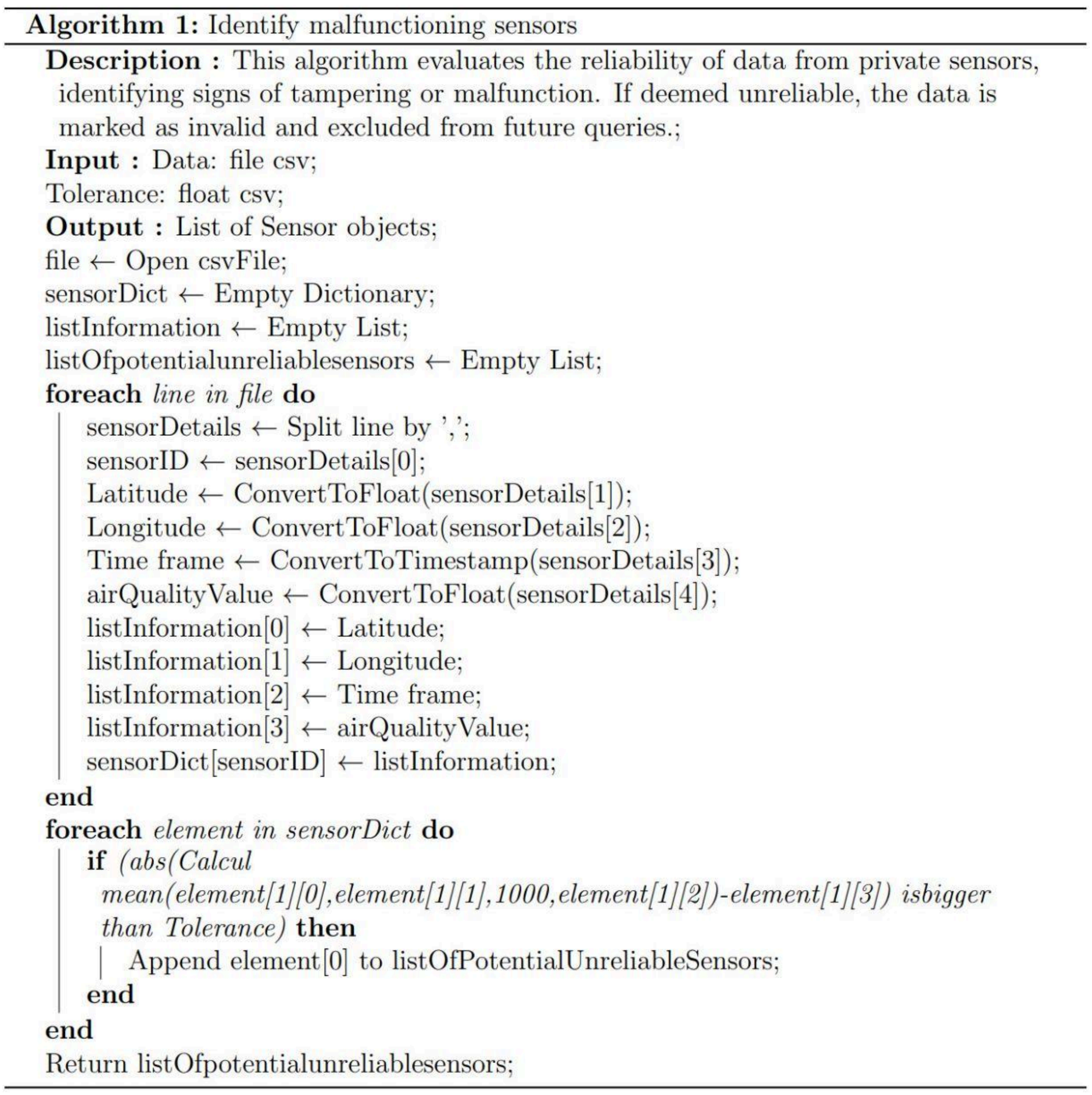
```
Algorithm 1: Calculate Mean Air Quality
Description : Calculates the mean air quality within a circular area specified by
  the user, using the time bounds provided to determine the period for calculation

Input : Latitude: float;
Longitude: float;
Radius: float;
Time frame: Timestamp;

Output : Mean air quality : float

list of air quality measurements ← [ ];
sum ← 0;
count ← 0;
foreach air quality measurement in the Time Frame do
    if the measurement is within a radius of Radius around (Longitude, Latitude)
      then
        Add the measurement to the list of measurements;
    end
end
if the list of measurements is not empty then
    foreach measurement in the list of measurements do
        sum ← sum + air quality of the measurement;
        count ← count + 1;
    end
    mean ← sum / count;
end
else
    No air quality measurements available for the specified period and location;
    Initialize mean to a default value or return an error;
end
Return mean;
```

Identify malfunctioning sensors



Algorithm 1: Identify malfunctioning sensors

```
Description : This algorithm evaluates the reliability of data from private sensors,
  identifying signs of tampering or malfunction. If deemed unreliable, the data is
  marked as invalid and excluded from future queries.;
Input : Data: file csv;
Tolerance: float csv;
Output : List of Sensor objects;
file ← Open csvFile;
sensorDict ← Empty Dictionary;
listInformation ← Empty List;
listOfpotentialunreliablesensors ← Empty List;
foreach line in file do
    sensorDetails ← Split line by ',';
    sensorID ← sensorDetails[0];
    Latitude ← ConvertToFloat(sensorDetails[1]);
    Longitude ← ConvertToFloat(sensorDetails[2]);
    Time frame ← ConvertToTimestamp(sensorDetails[3]);
    airQualityValue ← ConvertToFloat(sensorDetails[4]);
    listInformation[0] ← Latitude;
    listInformation[1] ← Longitude;
    listInformation[2] ← Time frame;
    listInformation[3] ← airQualityValue;
    sensorDict[sensorID] ← listInformation;
end
foreach element in sensorDict do
    if (abs(Calcul
      mean(element[1][0],element[1][1],1000,element[1][2])-element[1][3]) isbigger
      than Tolerance) then
        Append element[0] to listOfPotentialUnreliableSensors;
    end
end
Return listOfpotentialunreliablesensors;
```

Unit tests

Function 1: rankSensorsBySimilarity

Description:

- What it should do: Rank all sensors in terms of similarity to a selected sensor based on data generated during a specified period.
- Context (Pre-conditions):
    - The function requires sensor data from the specified period.
    - The function needs a selected sensor to compare against.

- Result (Post-conditions):
    - The function returns a ranked list of sensors based on their similarity to the selected sensor.

Use Cases:

	Input	Expected Output
1. Base Case - No Sensors Found	No sensors available in the database.	An empty list.
2. Single Sensor Available	Only one sensor is available in the database.	A list containing the single sensor with a similarity degree of 100%.
3. General Case - Multiple Sensors Available	Multiple sensors available in the database with varying degrees of similarity.	An ordered list of sensors based on their similarity degree, from most similar to least similar.
4. No Similar Sensors Found	No sensor is similar to the given sensor.	An empty list.
5. No Sensors Available	No sensors available in the database.	An empty list.
6. All Sensors Have Same Similarity	All available sensors have the same similarity degree with the given sensor.	A list containing all sensors, with identical similarity degrees for each.

Function 2: calculateMeanAirQuality

Description:

- What it should do: Calculate the mean air quality for a specified area and time period.
- Context (Pre-conditions):
    - The function requires a list of air quality measurements within the specified area and time period.
- Result (Post-conditions):

- The function returns the mean air quality value for the specified area and time period.

Use Cases:

	Input	Expected Output
1. No Measurements Available	No air quality measurements available within the specified time frame and location.	Return a default value or raise an error indicating no measurements available.
2. Single Measurement Available	Only one air quality measurement available within the specified time frame and location.	Return the air quality value of the single measurement as the mean.
3. Multiple Measurements Available	Multiple air quality measurements available within the specified time frame and location.	Calculate the mean air quality value based on all available measurements and return it.
4. Maximum Number of Measurements	Maximum number of air quality measurements available within the specified time frame and location.	Ensure the algorithm can handle the maximum number of measurements without performance issues or errors.
5. No Measurements within Radius	No air quality measurements available within the specified radius of the given location.	Return a default value or raise an error indicating no measurements available within the specified radius.
6. Invalid Time Frame	Invalid time frame provided (e.g., end time before start time).	Proper error handling and error message indicating the invalid time frame.
7. Invalid Location	Invalid location provided (e.g., latitude or longitude out of range).	Proper error handling and error message indicating the invalid location.

Function 3: identifyMalfunctioningSensors

Description:

- What it should do: Identify malfunctioning sensors based on their data and predefined thresholds.
- Context (Pre-conditions):
    - The function requires sensor data and predefined thresholds for identifying malfunctioning sensors.
- Result (Post-conditions):
    - The function returns a list of sensors flagged as malfunctioning.

Use Cases:

	Input	Expected Output
1. No Malfunctioning Sensors	All sensors are functioning properly.	Return an empty list indicating no malfunctioning sensors.
2. Single Malfunctioning Sensor	One sensor is malfunctioning.	Return a list containing the ID of the malfunctioning sensor.
3. Multiple Malfunctioning Sensors	Multiple sensors are malfunctioning.	Return a list containing the IDs of all malfunctioning sensors.
4. No Sensors Available	No sensors are available.	Return an empty list indicating no sensors available.
5. Large Number of Sensors	Large number of sensors, some of which are malfunctioning.	Verify the function's performance with a large number of sensors and ensure all malfunctioning sensors are correctly identified.