COBBLESTONE ENERGY

RESEACH PROJECT

DATA STREAM ANOMALY DETECTION

Autor

José Luis Rico Ramos

INDEX

1. Introduction and project requirements

The first thing I did was to read several times the problem description and the constraints, to get the functional and non-functional requirements of the project, to be clear about what the system should and should not do, as well as how it should do it.

Functional requirements (What does the system do?):

- The system shall generate the data stream with regular patterns, random noise and seasonal elements.
- The system shall capture real-time anomalies in the data stream.
- The system must have error handling.
- The system must not handle incorrect data input.
- System must display both flow data and anomaly capture.

Non-functional requirements (How does the system do it?):

- System must be implemented in Python 3.x.
- The code must be well documented and explained.
- The system must make minimal use of external libraries.
  The anomaly selection algorithm must be optimised for time and resource efficiency.

Once the requirements statement is finalised (which should be validated by the customer or product manager in a real case) and the problem is well understood, we can continue with the next points of the project process.

2. Data flow simulation

For the data flow simulation, I wanted to be as faithful as possible to reality and simulate as if they were financial or metric transactions. Thus, using a variety of trigonometric functions, such as sine, in this case using various frequencies, amplitudes and centres.

In this way with the variety of amplitudes we simulate these seasonal elements, as well as adding noise and anomalies to each of the values.

```
"""
Simulates a data stream with a variable sinusoidal pattern, random noise, and
occasional anomalies.

:param length: Number of data points in the stream
:param anomaly_prob: Probability of generating an anomaly
:param noise_level: Standard deviation of the added Gaussian noise
:return: Generator that yields simulated data points
"""
def data_stream_simulation(length=1000, anomaly_prob=0.01, noise_level=0.1):

    # Validate input parameters
    if not isinstance(length, int) or length <= 0:
        raise ValueError("length must be a positive integer")
    if not (0 <= anomaly_prob <= 1):
        raise ValueError("anomaly_prob must be between 0 and 1")
    if not isinstance(noise_level, (int, float)) or noise_level < 0:
        raise ValueError("noise_level must be a non-negative number")

    # First, we create a static data stream with np.arange, then convert it into a
sinusoidal function
    # with various wave frequencies, simulating stock market movements. After that,
noise and
    # anomalies in the form of spikes will be added to the graph.

    t = np.arange(0, length, 0.1)  # Time variable for the main wave without addons

    # Set the parameters for wave frequency, wave center, and wave amplitude

    frequency = 1  # Regular frequency of the main wave
    center_wave = np.sin(0.05 * t) * 2  # Low-frequency wave for the center,
amplitude of 2
    amplitude_wave = np.abs(np.sin(0.1 * t) + np.random.normal(0, 0.1, len(t))) *
3  # Amplitude between 0 and 3

    # In the for loop, we will create a point of the sinusoidal wave with its noise
and anomaly based on the
    # anomaly probability for each call to the generator
```

```
    for i in range(len(t)):
        center = center_wave[i]  # Center of the current sinusoidal wave
        amplitude = amplitude_wave[i]  # Amplitude of the current sinusoidal wave
        seasonal_pattern = center + amplitude * np.sin(frequency * t[i])
        noise = np.random.normal(0, noise_level)

        anomaly = 0
        if np.random.rand() < anomaly_prob:
            anomaly = np.random.uniform(-5, 5)  # Large anomaly spike

        yield seasonal_pattern + noise + anomaly
```

Let's now analyse the time and memory complexity:

- **Time Complexity:** As you can see, I use generators (yield in this case), so the complexity in time in the best case will be O (1), when contemplating only for that is executed depending on the number of times that the generator is called, since both the sine function, as the abs function, etc.... are O (1) as well. While in the worst case it will be O(n), being n the number of elements to be generated, which will depend on the length parameter.

- **Memory Efficiency:** For this function, it has been optimised with the use of the generators, since we do not store in memory the resulting array, but we generate it dynamically depending on the calls of the generators. Even so, the array t has to be stored in memory, so it would be O(n), where again n is the length of the list determined by the length parameter.

3. Anomaly detection algorithm

To do this, we do research using the internet and artificial intelligence, which is not a problem in my case because, apart from being a computer engineer, I also have a Bachelor's Degree in business administration and finance, where I have studied a lot of statistics and probability and I also studied several degrees in financial trading, increasing my knowledge in the study of information and methods of measuring deviations.

After a lot of research and thanks to the 360º knowledge I have about business and engineering, I managed to develop an algorithm that detects the anomalies adapting to the drift and sensational variations of our already declared data flow.

To do so, I based myself on 3 well-known fundamental theories and algorithms:

- **Z score:** This is a very common standardisation method, which is also often used to detect outliers. This method works by measuring the number of times a value is shifted from the mean using the standard deviation.

  The formula consists of:

  $$Z = (X - \mu)/\sigma$$

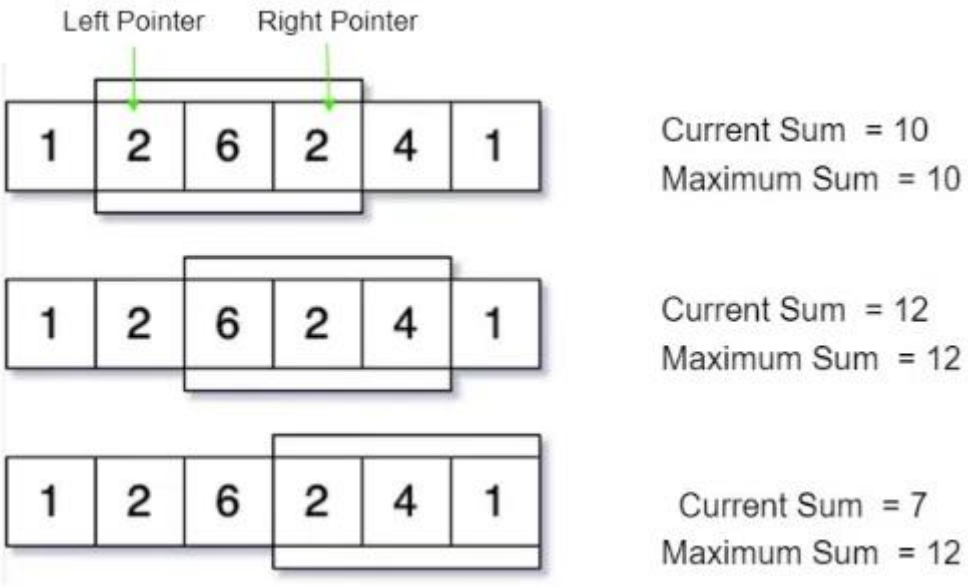
  Donde:
  - X: Value of element
  - μ: Population Mean
  - σ: Population Standard Deviation

  So, if the value for example of Z is 2.55, it means that the value is disparate to the mean 2.55 times the standard deviation, thus being able to obtain the anomalies.

- **Exponential Moving Average (EMA):** This is a technical indicator widely used in financial trading, which is based on the SMA (which is simply an average of values of an asset or index x days), but differs from the previous one in that the EMA gives more value to the days closer to the current date, i.e. it assigns weight depending on how new the values taken are.

- **Sliding Window Algorithm:** It is an algorithm that proposes a more efficient solution, both in time and memory, to problems involving operations with sub-arrays, which usually have a time complexity of O(n2), but thanks to this algorithm it is reduced in O(n).

  It is based on traversing the algorithms as if it were a deque (a data structure that is mainly a queue, but which can be pushed and popped both at the beginning and at the end of the queue). In this way, for example, imagine a problem to reach the largest sum of a subarray of length K, in a normal case we would have to pivot on an element i and go forward with another loop to go through all the array elements, which leaves us as I said with a time complexity of O(n2), however with the window algorithm, we take from k to k each of the elements moving from left to right using the deque, so we will end up with a time of O(n).



So, using these three algorithms I have designed one that will help me to find the different anomalies.

The heuristic is mainly based on the Z score, however, as with the Exponential Moving Average, we will only keep X number of last values that will have more weight to constantly recalculate the μ: Population Mean and σ: Population Standard Deviation, always being the X value a high number so that it is representative, making use of the window algorithm as we have explained previously.

In this way it is not necessary to recalculate μ and σ, each time with more and more values as time progresses, because in an assumption of infinite time and infinite values the complexity would tend to infinity and the performance would be worse and worse.

(Basically, if the number of values is 1000, it would be necessary to calculate each time that values with 1001,1002,1003... Which over time becomes inefficient, that is why the static number is established using the window, which advances at the rate of creation of new values).

```
"""
Detects anomalies in a data stream using Z-score over a sliding window.

:param data_stream: Generator that yields data points
:param window_size: Number of data points in the sliding window
:param threshold: Z-score threshold for flagging anomalies
:return: Generator yielding (data_point, is_anomaly) tuples
"""
def z_score_anomaly_detection(data_stream, window_size=50, threshold=3):

    # Validate input parameters
    if not hasattr(data_stream, '__iter__'):
        raise ValueError("data_stream must be an iterable or generator")
    if not isinstance(window_size, int) or window_size <= 0:
        raise ValueError("window_size must be a positive integer")
    if not isinstance(threshold, (int, float)) or threshold <= 0:
        raise ValueError("threshold must be a positive number")
```

```
    # Declare local variables for the function, such as the sliding window, mean, and
variance

    window = deque(maxlen=window_size)
    mean = 0
    variance = 0

    for data_point in data_stream:

        # Validate that the data point is a number

        if not isinstance(data_point, (int, float)):
            raise ValueError("data_point must be a number")

        # When the window is full, calculate the mean and variance for each iteration
of the generator

        if len(window) == window_size:
            mean = np.mean(window)
            variance = np.var(window)

        # Add the new data value to the end of the window, implicitly removing the
first one

        window.append(data_point)

        # If the window is not full, we cannot calculate the mean and variance
        if len(window) < window_size:
            yield data_point, False
        else: # Once the window is full, we can start detecting anomalies using the z-
score function
            std_dev = np.sqrt(variance)
            z_score = (data_point - mean) / std_dev if std_dev != 0 else 0
            is_anomaly = bool(abs(z_score) > threshold)  # Ensure is_anomaly is a
boolean
            yield data_point, is_anomaly
```

As we can see the algorithm is optimised in both time and memory, let's analyse it:

- **Time Complexity:** The time complexity is O (1) in the best case, because when using the yield element, a generator each time the generator is called will execute the code until the yield, and until the number of elements in the window is not == to the maximum, it will not calculate μ and σ. And in the worst-case O (n), where n is the maximum number of elements in the window, because in each call to the generator it has to recalculate with the displacement of the window and the new value of both μ and σ.

- **Memory Efficiency:** This will be O (n), where n is again the number of maximum elements of the window, as the complete values of the array that is returned are not stored in memory as it is a generator as we have mentioned before, however, we do need to store all the values of the window dynamically to be able to recalculate μ and σ.

4. Conclusion and graphic review

In conclusion, a review of the functional and non-functional requirements associated with the objectives set for the project shows that they have all been more than met:

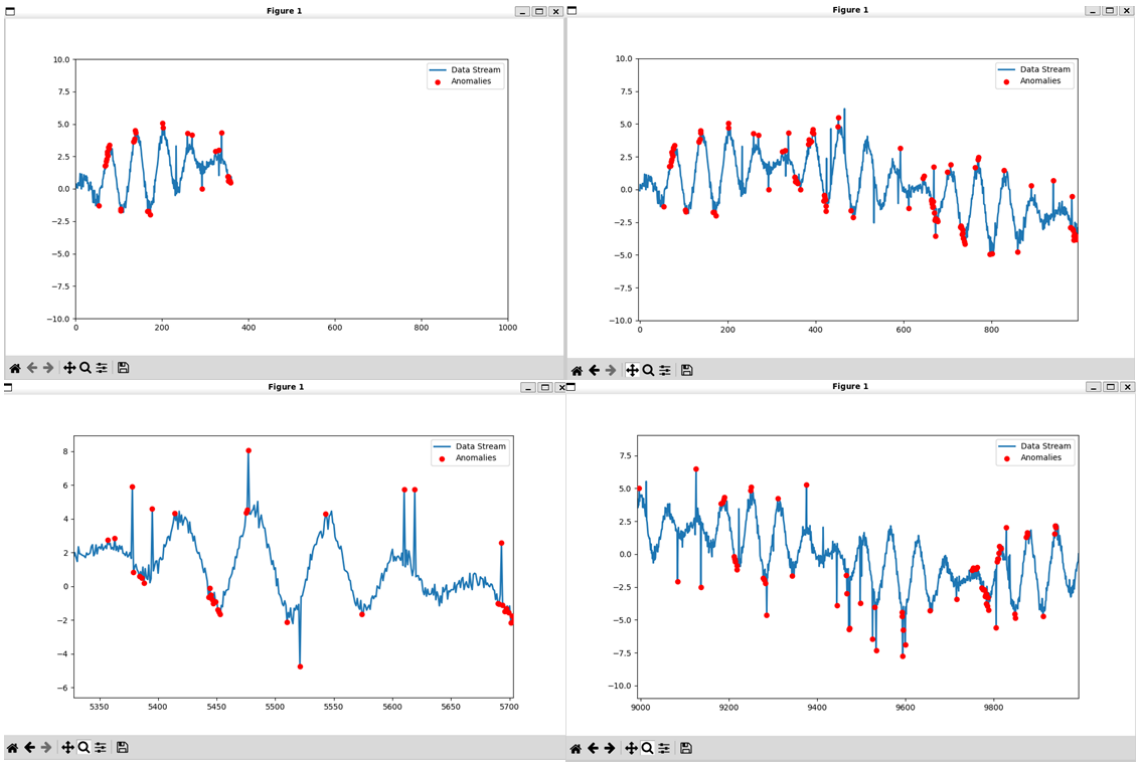
Functional requirements (What does the system do?):

- The system shall generate the data stream with regular patterns, random noise and seasonal elements. ✅
- The system shall capture real-time anomalies in the data stream. ✅
- The system must have error handling. ✅
- The system must not handle incorrect data input. ✅
- System must display both flow data and anomaly capture. ✅

Non-functional requirements (How does the system do it?):

- System must be implemented in Python 3.8.10. ✅
- The code must be well documented and explained. ✅
- The system must make minimal use of external libraries. ✅
  The anomaly selection algorithm must be optimised for time and resource efficiency. ✅

I would like to add, to avoid future problems with the execution of the GUI interface, that as I am on Windows 10, where the WSL does not have a natively integrated GUI, I have used an X server, where you can see how the points are generated in real time and the anomalies are highlighted in red.



As you can see the similarity with a pattern of financial transfers is very high, in addition to the fact that the algorithm is very well optimised for the detection of anomalies, in almost all the extremes the peaks where to buy and sell are detected correctly.

In conclusion, it has been a very cool and interesting project, which I enjoyed doing. If you liked it, I would love to talk to you about the project or the opportunities for recruitment.

Thank you very much for reading this far.