DataStream Anomaly Detection

Project Overview

This project focuses on developing a Python-based system for detecting anomalies in a continuous data stream, simulating real-time data that could represent metrics like system performance or financial transactions. The key objective is to identify unusual patterns, such as exceptionally high values or deviations from expected trends.

Approach:

The project utilizes the Double Exponential Moving Average (DEMA) for its anomaly detection mechanism. DEMA was chosen over other potential methods like **Isolation Forest** or **Z-score** for several reasons:

- ✓ **Responsiveness**: DEMA can adapt quickly to changes in data trends and seasonality, which are crucial for real-time systems.
- ✓ **Simplicity and Efficiency**: It offers a straightforward implementation that is computationally less intensive than the Isolation Forest, making it suitable for real-time processing.
- ✓ **Effectiveness in Trend and Seasonality**: Unlike the Z-score, which primarily focuses on deviations based on standard deviation and mean, DEMA effectively handles underlying trends and cyclical changes, providing more reliable anomaly detection in data with seasonal variations.

Addressing Objectives and Requirements:

1. Algorithm Selection

- **Chosen Algorithm**: **DEMA(**Double Exponential Moving Average**)** was implemented to track data trends and seasonal adjustments dynamically, providing a robust solution for real-time anomaly detection.

- **Rationale**: The algorithm's ability to quickly adapt to changes and its low computational overhead make it ideal for real-time applications where data characteristics may shift over time.

2. Data Stream Simulation

- **Implementation**: The simulated data stream includes components for trend, seasonality, and noise. It also injects anomalies at random intervals to test the detector's responsiveness.

- **Features**: This simulation allows for comprehensive testing of the anomaly detection mechanism under various scenarios that mimic real operational environments.

```python
class DataStream:
    """
    Simulates a real-time data stream by adding trend, seasonality, noise, and anomalies.

    """
    def __init__(self, size: int, noise: float, cycle: int):
        if size <= 0 or noise < 0 or cycle <= 0:
            raise ValueError("Size, noise, and cycle must be valid integers!")
        self.size = size
        self.noise = noise
        self.cycle = cycle
        self.index = 0

    def next_point(self) -> tuple:
        """
        Generates the next data point in the stream with applied trend, seasonality, and potential anomalies.

        """
        if self.index < self.size:
            flow = np.arange(self.index, self.index + 1)
            trend = np.linspace(10, 100, self.size)[self.index]
            seasonal = 15 * np.sin(2 * np.pi * flow / self.cycle)
            noise = self.noise * np.random.randn(1)

            # Generate anomalies with a certain probability
            if np.random.rand() > 0.98:
                value = trend + seasonal + noise + 50  # significant spike
            elif np.random.rand() > 0.96:
                value = trend + seasonal + noise - 50  # significant drop
            else:
                value = trend + seasonal + noise  # usual behavior

            self.index += 1
            return flow, value
        else:
            return None, None
```

3. Anomaly Detection

- **Real-time Processing**: Anomalies are detected in real-time as data is streamed, using the computed **DEMA** to identify significant deviations.

- **Dynamic Thresholding**: The detection mechanism adjusts thresholds based on recent data volatility, which accommodates concept drift and reduces false positives.

```python
class AnomalyDetector:
    """
    Detects anomalies using Double Exponential Moving Average (DEMA) based on dynamically adjusting thresholds.

    """
    def __init__(self, length: int, alpha: float, history: int, factor: float):
        if length <= 0 or alpha <= 0 or alpha > 1 or history <= 0 or factor <= 0:
            raise ValueError("All parameters must be positive, and alpha should be between 0 and 1!")
        self.length = length
        self.alpha = alpha
        self.history = history
        self.factor = factor
        self.ema = None
        self.dema = None
        self.past_data = deque(maxlen=history)

    def get_dema(self, point: float) -> float:
        """
        Computes Double Exponential Moving Average of the data points.

        """
        if self.ema is None:
            self.ema = point
        else:
            self.ema = self.alpha * point + (1 - self.alpha) * self.ema

        if self.dema is None:
            self.dema = self.ema
        else:
            self.dema = self.alpha * self.ema + (1 - self.alpha) * self.dema

        return self.dema
```

```python
def is_anomaly(self, point: float, time_flow: float) -> bool:
    """
    Checks if the latest data point is an anomaly by comparing it against the computed DEMA and a dynamic threshold.

    """
    dema = self.get_dema(point)
    self.past_data.append(point)

    # Dynamically adjust threshold based on historical data
    if len(self.past_data) == self.history:
        threshold = self.factor * np.std(self.past_data)
    else:
        threshold = 10.0  # Default threshold for early data points

    if abs(point - dema) > threshold:
        logging.info(f"Anomaly detected: {point} at time {time_flow} deviates significantly from expected DEMA {dema}.")
        return True
    return False
```

4. Optimization

- **Efficiency**: The implementation is optimized to handle high data volumes efficiently, ensuring minimal latency in anomaly detection. E.g. Updating the plot instead of new plot every time significantly enhances efficiency, especially as the data stream grows

- **Scalability**: The system is designed to scale with increased data volume without significant degradation in performance. E.g. Dynamically adjusts the detection thresholds based on the recent data volatility using **deque.**

5. Visualization

- **Real-time Plotting**: An interactive plot updates in real-time to display the data stream and any detected anomalies, enhancing the user's ability to monitor the system's performance visually.

```python
class Plotter:
    """
    Handles the plotting of data and anomalies in real-time.

    """
    def __init__(self):
        plt.ion()
        self.fig, self.ax = plt.subplots(figsize=(12, 6))
        self.ax.set_title('Data Stream with Anomalies')
        self.ax.set_xlabel('Time')
        self.ax.set_ylabel('Data Points')
        self.times = []
        self.values = []
        self.anomalies = []

        # Initialize plot lines
        self.line, = self.ax.plot([], [], label='Normal Data', color='blue')
        self.anomaly_scatter = self.ax.scatter([], [], color='red', label='Anomaly')

    def update_plots(self, time_flow:list, value:list, anomaly: bool) -> None:
        """
        Updates the plot with new data points and marks anomalies.
        """
        self.times.append(time_flow[0])
        self.values.append(value[0])

        # Update the data for the line plot
        self.line.set_data(self.times, self.values)

        if anomaly:
            self.anomalies.append((time_flow[0], value[0]))

        if self.anomalies:  # Update scatter plot for anomalies
            x, y = zip(*self.anomalies)
            self.anomaly_scatter.set_offsets(list(zip(x, y)))

        # Adjust the plot limits dynamically
        self.ax.set_xlim(left=max(0, time_flow[0] - 50), right=time_flow[0] + 10)
        self.ax.set_ylim(min(self.values) - 10, max(self.values) + 10)

        self.ax.legend()
        self.fig.canvas.draw()
        self.fig.savefig(f'anomaly_detection_plot.png') # this image will also update in real time in case there are issues with the canva.draw()
```

6. Documentation and Code Quality

- **Code Comments and Documentation**: Extensive documentation and comments throughout the codebase facilitate understanding and maintenance.

- **Robust Error Handling**: Comprehensive error-handling strategies ensure the system's robustness and reliability.

Code Results:

config.json:

The config.json file contains key parameters that are crucial for tuning the anomaly detection system. These settings allow for flexibility and ease of adjustment without modifying the code directly. Here is a breakdown of each parameter:

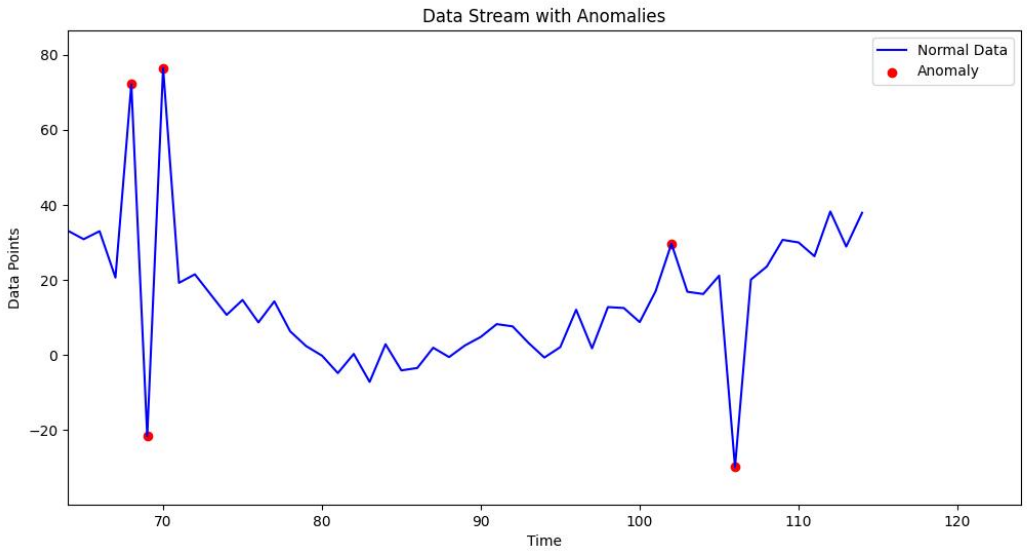
- **data_stream_size**: Specifies the total number of data points that the simulation will generate. This setting impacts how long the data stream runs during each session of the simulation.
- **noise_level**: Defines the standard deviation of the Gaussian noise added to the data stream. This simulates measurement errors or environmental noise, adding realism to the simulated data.
- **seasonality**: Sets the period of the sine wave used to model seasonality in the data stream. This parameter helps simulate periodic fluctuations, such as daily or weekly cycles, in the data.
- **window_size**: Determines the size of the window used for calculating the Double Exponential Moving Averages (DEMA). It defines how many recent data points the system should consider when calculating these averages.
- **ema_alpha**: The smoothing factor for the Exponential Moving Average calculations. This value is between 0 and 1 and controls the weight given to more recent data points. A higher value places more emphasis on recent changes in the data, making the average more responsive to new trends.
- **lookback**: Indicates the number of recent data points used for calculating the dynamic threshold for anomaly detection. This setting helps in determining how volatile the recent data has been and adjusts the sensitivity of anomaly detection accordingly.
- **threshold_factor**: A multiplier applied to the calculated standard deviation of the lookback period to set the dynamic threshold for detecting anomalies. This factor can be tuned to make the system more or less sensitive to changes that might be considered anomalous.
- **batch_size**: Specifies the number of data points processed in each batch during the simulation. This setting is particularly useful for managing processing load and responsiveness in real-time applications.

The following snippets show how the system processes a sample data stream and detects anomalies:

Running **main.py**:

```
2024-09-14 18:49:49,254 - INFO - Anomaly detected: 27.549994178406983 at time 4 deviates significantly from expected DEMA 14.536502163819943.
2024-09-14 18:49:50,242 - INFO - Anomaly detected: 34.202931581740614 at time 11 deviates significantly from expected DEMA 20.161641807563626.
2024-09-14 18:49:52,283 - INFO - Anomaly detected: 1.6937943882492803 at time 24 deviates significantly from expected DEMA 20.093838294256166.
2024-09-14 18:49:59,162 - INFO - Anomaly detected: 72.1511274065555 at time 68 deviates significantly from expected DEMA 31.169194323047225.
2024-09-14 18:49:59,318 - INFO - Anomaly detected: -21.592127993858963 at time 69 deviates significantly from expected DEMA 28.415572642293426.
2024-09-14 18:49:59,488 - INFO - Anomaly detected: 76.50669068878818 at time 70 deviates significantly from expected DEMA 31.394498642908594.
2024-09-14 18:50:04,729 - INFO - Anomaly detected: 29.699059672573263 at time 102 deviates significantly from expected DEMA 10.688740862845169.
2024-09-14 18:50:05,327 - INFO - Anomaly detected: -29.82547053717567 at time 106 deviates significantly from expected DEMA 11.61324359055135.
```

Plot:



Conclusion

The Cobblestone Anomaly Detection system provides a reliable and efficient tool for monitoring and detecting anomalies in real-time data streams, equipped with robust documentation, dynamic adaptability, and efficient processing capabilities.