Efficient Data Stream Anomaly Detection

Anomaly Detection Using Z-Score with Various Window Sizes.

Introduction

This project aims to detect anomalies in a data stream using a Z-Score-based algorithm. The algorithm adapts to seasonal variations and concept drift by applying a sliding window technique to maintain efficiency. Different window sizes were tested, and it was found that a window size of 30 produced the best results.

Objectives

1. Algorithm Selection: Implement a suitable anomaly detection algorithm, adaptable to seasonal variations.
    - Given the need for efficiency and adaptability, Z-Score with a sliding window would be lightweight and sufficient for this project.
2. Data Stream Simulation: Emulate a data stream incorporating seasonal elements and random noise.
    - We can simulate a data stream by generating a sequence of values that combine regular patterns, noise, and occasional anomalies.
3. Anomaly Detection: Create a real-time mechanism to flag anomalies.
    - I implemented a real-time detection method using the Z-Score algorithm. The Z-Score can help identify values that are too far from the mean.
4. Optimization: Ensure the algorithm is optimized for both speed and efficiency.
    - the sliding window approach is efficient because it maintains only a small portion of the data in memory, avoiding performance bottlenecks.
    - While experimenting with different window sizes 30 was found to provide the best balance between sensitivity and accuracy.
5. Visualization: Provide a real-time visualization tool to display the data stream and detected anomalies.
    - I created a simple real-time plot using matplotlib that provides real-time detection for anomalies.

Code Documentation

Step 1: Generate Data Stream The data stream is generated using a sinusoidal function to simulate seasonal variations, along with random noise and injected anomalies.

```
# Step 1: Generate the data stream
def generate_data_stream(length=1000, noise_level=0.2, seasonal_period=50, anomaly_ratio=0.05):
    """
    Simulates a data stream with regular patterns, noise, and anomalies.

    Args:
        length (int): The total length of the data stream.
        noise_level (float): The standard deviation of random noise to be added to the data stream.
        seasonal_period (int): The period of the seasonal component (sinusoidal pattern).
        anomaly_ratio (float): The proportion of data points that should be anomalies.

    Returns:
        np.ndarray: Simulated data stream with seasonal variations, noise, and injected anomalies.
    """
    time = np.arange(length)  # Time points
    seasonal = np.sin(2 * np.pi * time / seasonal_period)  # Seasonal sinusoidal component
    noise = np.random.normal(0, noise_level, length)  # Random noise
    data_stream = seasonal + nois (parameter) anomaly_ratio: float

    # Introduce anomalies by addi anomaly_ratio (float): The proportion of data points that should be anomalies.
    num_anomalies = int(length * anomaly_ratio)
    anomaly_indices = np.random.choice(length, num_anomalies, replace=False)
    data_stream[anomaly_indices] += np.random.uniform(5, 10, num_anomalies)  # Anomalies are outliers

    return data_stream

# Generate the data stream with default parameters
stream = generate_data_stream()
```

Step 2: Validate Data Stream The generated data stream is validated to ensure it is free from NaN or infinite values.

```
# Step 2: Validate the data stream
def validate_data_stream(stream):
    """
    Validates the data stream to ensure it's an appropriate input for anomaly detection.

    Args:
        stream (np.ndarray): The data stream to validate.

    Raises:
        ValueError: If the data stream is not a valid NumPy array or contains invalid values (NaN or inf).
    """
    if not isinstance(stream, np.ndarray) or np.isnan(stream).any() or np.isinf(stream).any():
        raise ValueError("Invalid data stream")

# Validate the generated stream
validate_data_stream(stream)
```

Step 3: Z-Score Anomaly Detector A Z-Score anomaly detector is implemented to flag anomalies based on deviations from the sliding window's mean and standard deviation.

```python
# Step 3: Define the Z-Score anomaly detector class
class ZScoreAnomalyDetector:
    """
    Anomaly detector based on Z-Score, which identifies anomalies by comparing the latest value
    to the mean and standard deviation of a sliding window of previous values.
    Args:
        window_size (int): The size of the sliding window to calculate the mean and standard deviation.
        threshold (float): The Z-Score threshold above which a value is considered an anomaly.
    """
    def __init__(self, window_size=30, threshold=3):
        self.window_size = window_size  # Size of the sliding window
        self.threshold = threshold  # Z-Score threshold for anomaly detection
        self.data_window = deque(maxlen=window_size)  # Sliding window to store recent data points
    def detect(self, new_value):
        """
        Detects if the incoming value is an anomaly based on the Z-Score.
        Args:
            new_value (float): The latest data point in the stream.
        Returns:
            bool: True if the data point is an anomaly, False otherwise.
        """
        if len(self.data_window) < self.window_size:
            # Not enough data yet, just store the value and don't flag as anomaly
            self.data_window.append(new_value)
            return False
        # Calculate mean and standard deviation of the window
        mean = np.mean(self.data_window)
        std_dev = np.std(self.data_window)
        if std_dev == 0: # If standard deviation is zero, we can't compute Z-Score, so we return False
            return False
        # Compute Z-Score for the new value
        z_score = (new_value - mean) / std_dev
        # Update the sliding window with the new value
        self.data_window.append(new_value)
        # Return True if the Z-Score exceeds the threshold (anomaly detected)
        return abs(z_score) > self.threshold
# Initialize the Z-Score anomaly detector with a window size of 30 and threshold of 3
detector = ZScoreAnomalyDetector()
```

Step 4: Real-time Plotting The data stream is visualized in real-time using matplotlib, with anomalies highlighted in red.

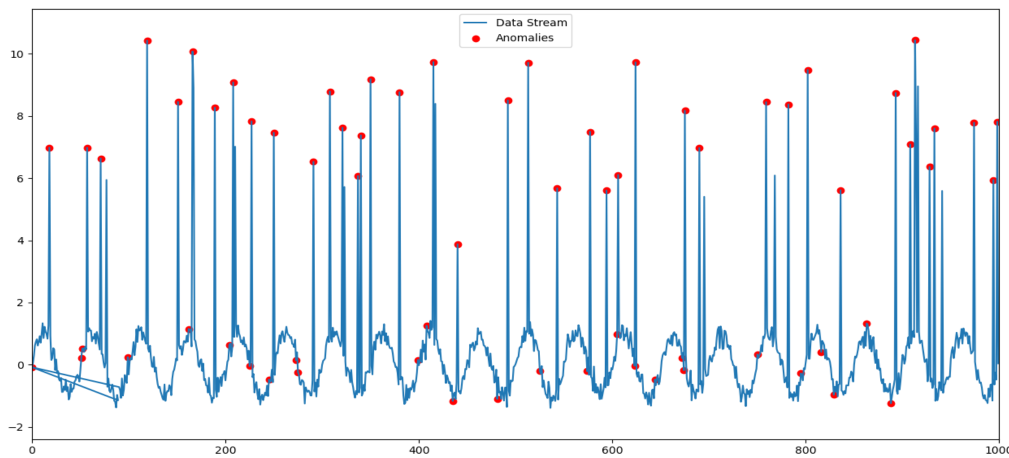
```python
# Step 4: Real-time plotting and anomaly detection
def real_time_plot(stream, detector):
    """
    Visualizes the data stream in real time, marking anomalies as red dots.

    Args:
        stream (np.ndarray): The data stream to visualize.
        detector (ZScoreAnomalyDetector): An instance of the Z-Score anomaly detector.
    """
    fig, ax = plt.subplots()
    xdata, ydata = [], []  # Data lists for plotting
    anomalies = []  # List to store anomaly coordinates

    # Set plot limits for better visualization
    ax.set_xlim(0, len(stream))
    ax.set_ylim(np.min(stream) - 1, np.max(stream) + 1)

    # Plot line for the data stream and scatter for anomalies
    line, = ax.plot([], [], label="Data Stream")
    anomaly_scatter = ax.scatter([], [], color='red', label="Anomalies")
```

```python
    def update(frame):
        """
        Update function for the animation, called for each new frame (data point).

        Args:
            frame (int): The current frame number in the animation.

        Returns:
            line, anomaly_scatter: Updated plot elements (data stream and anomalies).
        """
        value = stream[frame]  # Get the value from the stream for the current frame
        is_anomaly = detector.detect(value)  # Detect if the value is an anomaly

        # Append the new data point to the plotting lists
        xdata.append(frame)
        ydata.append(value)

        # Update the data stream line plot
        line.set_data(xdata, ydata)

        # If an anomaly is detected, add it to the scatter plot
        if is_anomaly:
            anomalies.append((frame, value))
            anomaly_scatter.set_offsets(anomalies)

        return line, anomaly_scatter

    # Create the animation that updates the plot in real-time
    ani = FuncAnimation(fig, update, frames=len(stream), blit=True, repeat=False)

    # Display legend and show the plot
    ax.legend()
    plt.show()

# Run the real-time plot function with the generated data stream and anomaly detector
real_time_plot(stream, detector)
```

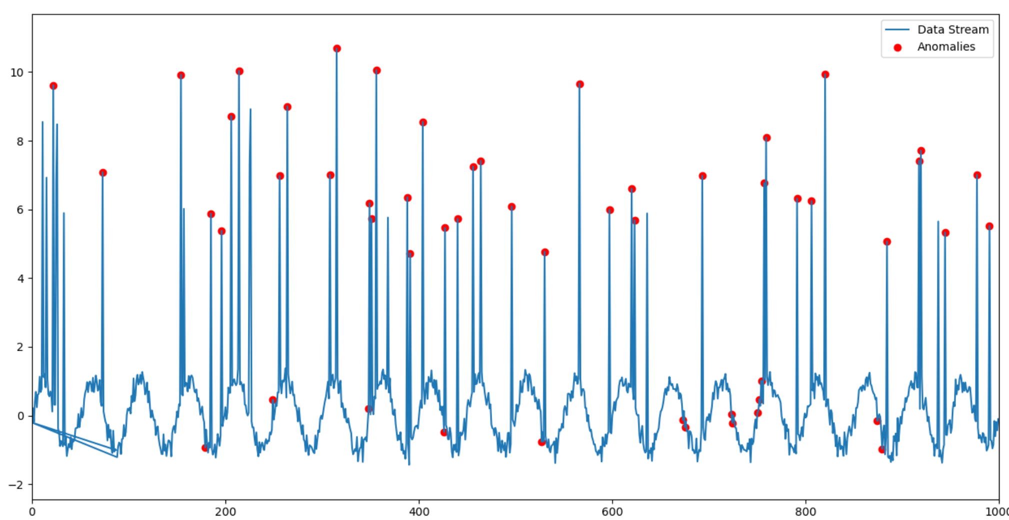
Results

The following images show the anomaly detection results for different window sizes. Window size 30 was found to provide the best balance between sensitivity and accuracy.

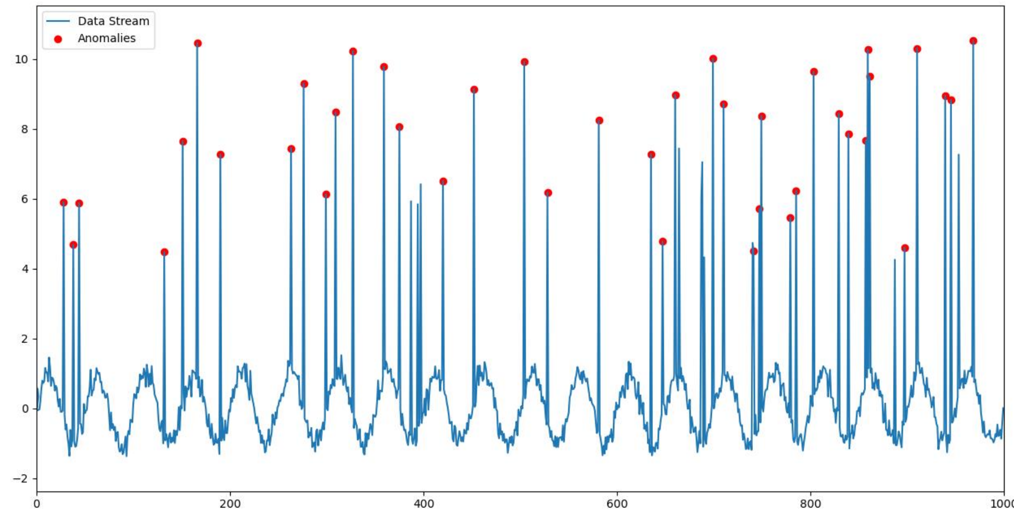
- **Window Size = 10**



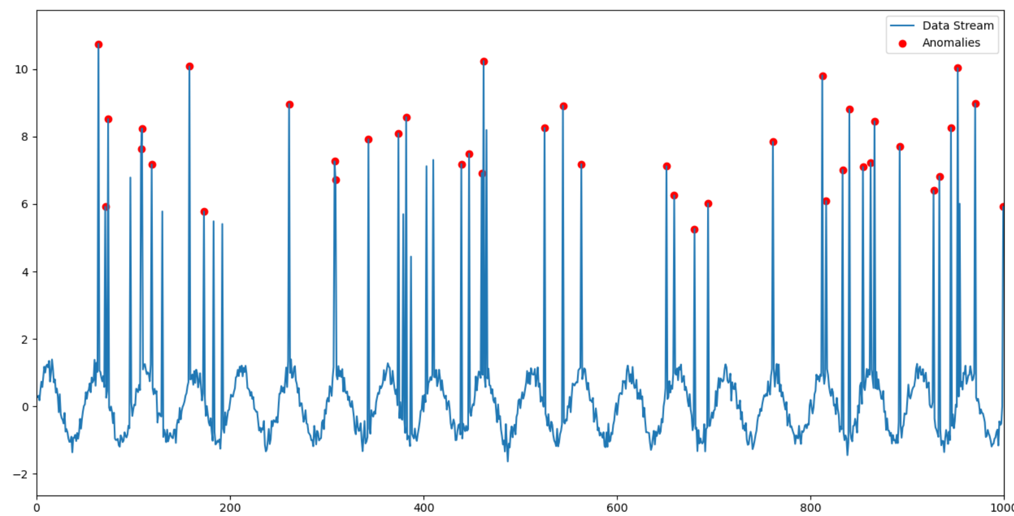
- **Window Size = 20**



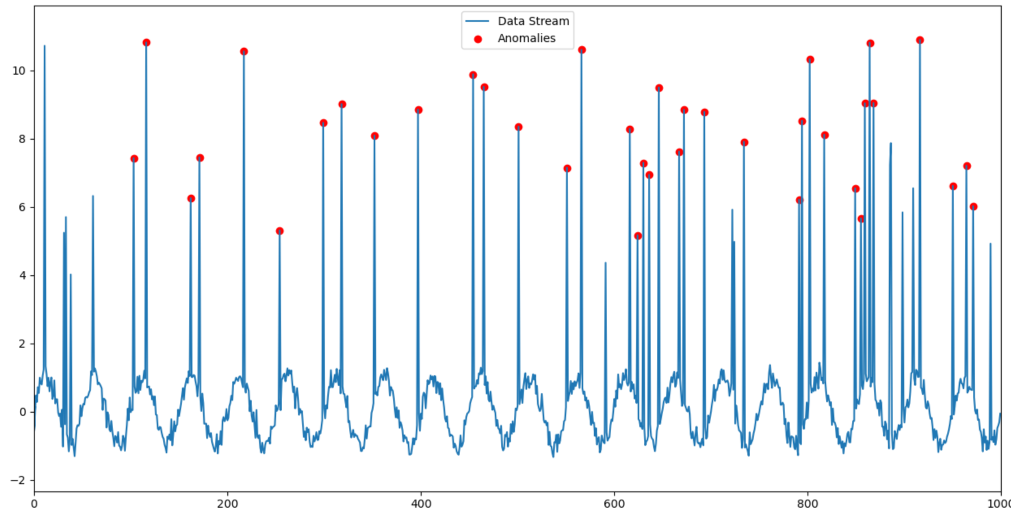
❖ Window Size = 30



❖ Window Size = 40



❖ Window Size = 50



❖ Window Size = 100