Efficient Data Stream Anomaly Detection

Abeer Megahed

Introduction

The process of identifying unusual patterns or behaviours within a data set. These irregularities in the data are considered *anomalies* or outliers; and could indicate an inconsistency or a malfunction in the system being monitored.

Methods used for anomaly detection includes:
1. **Statistical:** Searches for anomalies by analysing a time series' mean, variance, or distribution. (e.q. Z-score which measures the distance between a data point and the mean by standard deviation).
2. **Machine Learning:** By training a machine learning model to detect anomalies in time series dataset (e.q. Isolation forest & Support Vector Machines (SVMs)).
3. **Signal Processing:** Identifies changes in signal patterns generated by the monitored system (e.q. Fourier transformation).
4. **Hybrid:** Combining multiple techniques for accuracy improvement.

Algorithm Selection

The chosen method is a simple and efficient implementation using NumPy's vectorized operations.

This approach involves calculating the **moving average** using a convolution operation, which is optimized for performance in NumPy. The convolution efficiently computes the average of a fixed number of data points within a sliding window. This method is suitable for data streams where the computational cost is not a major concern and where the underlying data distribution is relatively stable.

While moving average may not be as effective as more sophisticated algorithms for handling concept drift and seasonal variations, it can be a good choice for basic anomaly detection tasks, especially when combined with other techniques or when the data stream is relatively small.

Data Stream Simulation

Generating a simulated data stream with seasonal patterns and random noise:

```
def generate_data_stream(length, seasonality, random_noise):
    time_series = np.zeros(length)
    for i in range(length):
        time_series[i] = np.sin(i / seasonality) + np.random.normal(0, random_noise)
    return time_series
```

Debug: [length = 10000, seasonality = 100, random_noise = 0.5]

```
Index: 0, Value: 0.3160385048099119
Index: 1, Value: 0.41309540868480366
Index: 2, Value: 1.4761674026577967
Index: 3, Value: -0.8245013151939063
Index: 4, Value: 0.5419592203868971
Index: 5, Value: -0.7481395444061951
Index: 6, Value: -0.05056495580773045
Index: 7, Value: 0.63834663528097
Index: 8, Value: 0.6619816440363916
Index: 9, Value: -0.20179504676879145
Index: 10, Value: 0.5090913939947022
```

Anomaly Detection

Detecting anomalies in the data stream using a simple moving average approach:

```
def detect_anomalies(data_stream, window_size, threshold):
    anomalies = []
    moving_average = np.convolve(data_stream, np.ones(window_size) / window_size, mode='valid')
    for i in range(len(moving_average)):
        if abs(data_stream[i + window_size - 1] - moving_average[i]) > threshold:
            anomalies.append(i + window_size - 1)
    return anomalies
```

Debug: [random_noise = 0.6, window_size = 50, threshold = 2]

```
Anomaly detected at index: 219
Anomaly detected at index: 239
Anomaly detected at index: 1085
Anomaly detected at index: 1650
Anomaly detected at index: 1958
Anomaly detected at index: 2923
Anomaly detected at index: 4322
Anomaly detected at index: 5127
Anomaly detected at index: 5140
Anomaly detected at index: 5762
Anomaly detected at index: 8354
Anomaly detected at index: 8714
Anomaly detected at index: 9101
```

Optimization

The current function uses a simple moving average approach to identify anomalies. While effective for basic scenarios, it can be computationally expensive for large data streams.

Optimization Strategies:

1. **Vectorization:**

Utilizing NumPy's vectorized operations to perform calculations on entire arrays instead of iterating over the data stream can significantly improve performance.

```
def detect_anomalies_vectorized(data_stream, window_size, threshold):

    moving_average = np.convolve(data_stream, np.ones(window_size) / window_size, mode='valid')
    anomaly_indices = np.where(np.abs(data_stream[window_size - 1:] - moving_average) >
threshold)[0]

    return anomaly_indices
```

2. **Incremental update:**

Reducing computational overhead by updating the moving average incrementally for each new data point instead of recalculating it from scratch.

```
def detect_anomalies_incremental(data_stream, window_size, threshold):

    moving_average = np.sum(data_stream[:window_size]) / window_size
    anomalies = []

    for i in range(window_size, len(data_stream)):
        moving_average = (moving_average * window_size - data_stream[i - window_size] +
data_stream[i]) / window_size
        if abs(data_stream[i] - moving_average) > threshold:
            anomalies.append(i)

    return anomalies
```

Visualization

Adjust the parameters [stream length, seasonality, random noise, window size, and threshold] to the desired values and run the visualization (Static or animated):

```python
if __name__ == '__main__':
    data_stream_length = 10000
    seasonality = 100
    random_noise = 0.5
    window_size = 50
    threshold = 2

    # Generate data stream
    data_stream = generate_data_stream(data_stream_length, seasonality, random_noise)
    # Detect anomalies
    #anomalies = detect_anomalies(data_stream, window_size, threshold)
    #anomalies = detect_anomalies_vectorized(data_stream, window_size, threshold)
    anomalies = detect_anomalies_incremental(data_stream, window_size, threshold)
    # Visualize data stream
    visualize_data(data_stream, anomalies)
    #visualize_data_animated(data_stream, anomalies)
```