Open-source Technologies and Stream Mining joint Project Documentation

Table of contents

Smart City Air Quality Monitoring with Real-Time Stream Analytics (SCAir-IoT)

Abstract

lorem ipsum...

Keywords: *Anomaly detection, Forecasting, Stream mining, Open-source technologies*

Authors

Albazzal, Houmam
Balogh, Máté
Földvári, Ádám
Lahmar, Abderraouf
Nagy, Zsuzsanna

Mentored by

Loubna Seddiki

1 Introduction

2 Background and Literature Review

In stream mining, we are limited to a portion of data and make decisions real-time in memory. As *Wares, Isaacs, and Elyan (2019)* highlight, in traditional machine learning contexts, data is referred to as batch data which can be loaded into memory in its entirety. According to the authors,

> "this is of stark contrast to stream mining, where data streams produce elements in a sequential, continuous fashion, and may also be impermanent, or transient, in nature ... This means stream data may only be available for a short time."

The authors refer to *Babcock et al. (2002)*, highlighting that

> "once an element from a data stream has been processed, it is discarded or archived. It cannot be retrieved easily unless it is explicitly stored in memory, which is small relative to the size of data streams."

3 Dataset

The dataset is the *UCI Air Quality* dataset *Vito, S. (2008)* which includes responses of gas sensor devices deployed in an Italian city. Besides these device readings, each gas measurement has a counterpart feature which denotes the gas concentration recorded by a co-located certified analyzer. Additionally, readings related to temperature along with absolute and relative humidity are included in the dataset.

The records span 1 year from March 2004 to February 2025, and present hourly aggregated measurements. Missing values are denoted with the value of `-200`.

4 System architecture

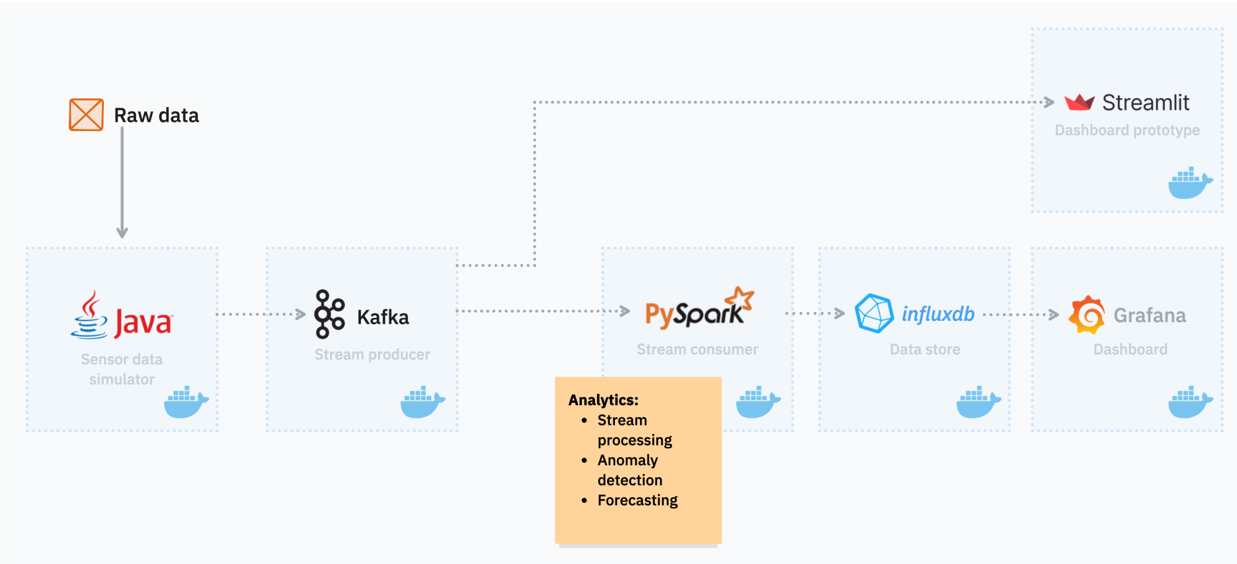

Figure 1: High-level view of the architecture with the utilized open-source technologies denoted for each component.

The stream mining pipeline includes components of *simulator*, *producer*, *consumer*, *data store*, and *dashboards* by which the raw `csv` dataset file was ingested. We utilized open-source technologies of *Java*, *Kafka*, *PySpark*, *Influxdb*, *Streamlit*, and *Grafana* provisioned in a containerized environment via *Docker*. The responsibility of each component is summarized as follows:

Raw data: Static file containing sensor measurements related to air quality.

Sensor data simulator: Reads the raw data file and simulates flow of sensor data.

Kafka producer: Streamlines the simulated sensor data into Kafka topics in their datetime order.

PySpark consumer: Listens to the streamlined data and creates mini batches to call analytical functions—such as Anomaly Detection and Forecasting—on this windowed data.

Influxdb: Data is then persisted in the database including the online predictions and the original incoming data.

Streamlit: Streamlit is used to create dashboard prototypes without the necessity of a database storage and connection. Kafka messages are directly consumed by this component to display simple line charts and anomaly detection related information and alerts.

Grafana: Dashboard visualization component that periodically fetches the database for new data to show the latest insights in real-time.

5 Modeling and Predictions

5.1 Overview

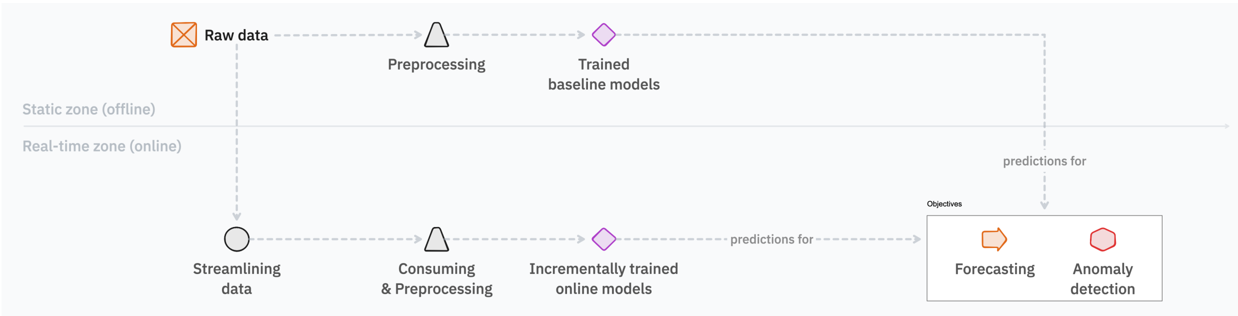

Figure 2: High-level view of the offline and online machine learning pipeline.

5.2 Anomaly detection

For *Anomaly Detection*, different techniques were utilized ranging from recognizing sensor readings with missing measurement to detecting local and global outliers, in each sensor seperately.

Both, local and global approaches were developed with the general novelty, outlier detection methodology in mind, where the steps are:

1. Transform the data, to better reflect properties of interest
2. Obtain a novelty function
3. Apply peak picking algorithm on the novelty function

To detect local patterns in form of sudden changes (peaks and valleys), an 8-sample window was utilized on the streamlined data. As part of the data transformation, values of `-200` were replaced with `NA`. The most recent element in the window was selected as a test sample to make predictions for, while the rest of the in-window samples (at most 7) were designated as historical training points. Derivative operation was applied on the window data to turn a measurement value x_T into the deviation from x_{T-1}. This resulted in the novelty function, where `NA` values were first imputed with the window median but were transformed back to `NA` once the difference values were obtained. During the "training", the in-window estimator calulates statistics $(IQR, Q1, Q3)$ from the in-window historical samples to perform a traditional outlier detection test using the $Q1 - 1.5 \times IQR$ and $Q3 + 1.5 \times IQR$ as thresholds to flag the test sample as outlier.

To account for out-of-window global patterns and to implement a detector that is applicable in stream mining systems where we cannot always rely on storing every incoming data point, an online detector was implemented using the *T-digest* algorithm. This method uses the same peak picking strategy using the traditional outlier thresholds, yet the quartiles along with the *Interquartile Range* get iteratively updated by each new sample. The algorithm maintains a centroid-based representation about these statistics using the incoming data and updates its state through merges.

5.3 Offline forecasting

The offline forecasting module represents a traditional Supervised Machine Learning pipeline where models are trained on historical datasets and deployed for inference on streaming data. This approach prioritizes model complexity and accuracy over immediate adaptability, assuming that the underlying statistical properties of the air quality data remain relatively stable over short periods.

5.3.1 Model Training and Architecture

The foundation of this module is established in **test_Modelling.ipynb**. The training process begins by loading the UCI Air Quality dataset and performing rigorous preprocessing, including datetime indexing and handling missing values (mapping sensor error codes like **-200** to **NaN**). The core of the strategy relies on **supervised feature engineering**, where the time-series problem is transformed into a regression problem. The system constructs a rich feature set comprising **lagged values** (1, 2, 3, 6, 12, 24 hours) to capture immediate and daily dependencies, **rolling statistics** (mean and standard deviation over 3, 6, 12-hour windows) to smooth volatility, and **temporal embeddings** (hour of day, day of week, month).

The primary algorithm utilized is the **Histogram-based Gradient Boosting Regressor** (HGBR). This algorithm is chosen for its efficiency with large datasets and native handling of missing values (NaNs), which are common in sensor networks. Separate models are trained for specific forecast horizons (H+1, H+2, and H+3 hours ahead). These trained models, along with their metadata, are serialized using **joblib** and stored in the **artifacts/** directory, ensuring that the inference engine has access to pre-validated statistical patterns.

5.3.2 Inference Pipeline and State Management

The inference logic is encapsulated in **offline_forcasting/offline_forecasting.py** within the **OfflineForecaster** class. This class acts as a singleton **Sliding Window Listener** that consumes raw sensor data arriving from the Simulator via Kafka. A critical challenge in offline-to-online deployment is **feature consistency**; the inference engine must reconstruct the exact feature set used during training. To achieve this, the forecaster maintains in-memory buffers (**deques**) for every sensor topic.

As data streams in, the **OfflineForecaster** aligns the history across different sensors. Once the buffer accumulates enough samples to calculate the required lags and rolling windows (left-padding with **NaNs** during cold starts), it constructs a single-row feature vector. This vector is passed to the loaded HGBR pipelines. To ensure robustness, the system respects the **feature_names_in_** attribute of the saved models, preventing column mismatch errors. Predictions are generated both **on-demand** (triggered by window updates) and via a **periodic background thread** (defaulting to a 10-second cadence). Finally, the forecasted values are persisted to **InfluxDB** via the **dbWriter**, allowing for immediate visualization and comparison against actual incoming values.

Suggestions for Future Improvements

- **Automated Retraining Pipeline:** Currently, models are static. Implementing a "Champion/Challenger" system that periodically retrains models on the most recent week of data and swaps them in production would mitigate long-term model rot.
- **Deep Learning Architectures:** Replacing Gradient Boosting with LSTM (Long Short-Term Memory) or Transformer-based architectures (like Temporal Fusion Transformers) could better capture long-range dependencies, provided the inference latency remains acceptable.
- **Feature Store Integration:** Decoupling feature calculation from the application logic using a Feature Store (e.g., Feast) would ensure stronger consistency between the training notebook and the inference script.

5.4 Online forecasting

The **Online Forecasting** module, implemented in **online_forcasting/online_forecasting.py**, addresses the limitations of static models by implementing **Incremental Learning**. Unlike the offline approach, this system does not rely on pre-existing artifacts. Instead, it initializes fresh models that learn continuously from the data stream, allowing the system to adapt rapidly to **concept drift** (e.g., sudden changes in sensor calibration or environmental conditions).

5.4.1 Theoretical Context and Algorithm

Online learning differs fundamentally from batch learning by performing a **"predict-then-update"** cycle. The system utilizes the **Stochastic Gradient Descent (SGD) Regressor**, a linear model optimized for streaming. While linear models are generally less complex than tree-based ensembles, SGD is computationally inexpensive and capable of updating weights sample-by-sample (**partial_fit**).

The workflow is strictly sequential:

1. **Extract Features (X_t)**: Current lags and covariates are computed.
2. **Predict (y_{t+1}):** The current model predicts the next step.
3. **Wait:** The system waits for the actual observation of y_{t+1}.
4. **Update:** Once the actual value arrives, the model calculates the error (loss) and updates its weights via backpropagation to minimize future errors.

Crucially, because SGD is sensitive to feature scaling, the pipeline includes an incremental **StandardScaler** that also updates its mean and variance estimates on the fly, ensuring that the gradient descent converges correctly even as the statistical properties of the raw data shift.

5.4.2 Adaptive Pipeline and Robustness

The **OnlineForecaster** manages its own bounded buffers (defaulting to 200 points) to construct features similar to the offline model but optimized for speed: short-term lags (1, 2, 3, 6 hours) and covariates (Temperature and Humidity interactions). To prevent the model from making erratic predictions during the initial "warm-up" phase or during sensor malfunctions, several **robustness mechanisms** are implemented.

The system maintains streaming **Quantile Sketches** (using T-Digest concepts) and calculates **Z-scores** on a short sliding window. If an incoming value is detected as a statistical anomaly (e.g., $> 3.5\sigma$), the system flags it. Furthermore, predictions are **clamped** to a dynamic confidence band derived from recent rolling statistics. If the model produces a prediction that is physically implausible or diverges wildly from the recent trend, the system falls back to a simple rolling mean. This hybrid approach ensures that the forecaster remains stable even when the SGD model is under-fitted or facing outlier data. The resulting one-step-ahead forecasts are written to **InfluxDB**, tagged specifically as **online_pred**.

5.4.3 Suggestions for Future Improvements

- **Adaptive Learning Rates:** Implementing mechanisms to dynamic adjust the SGD learning rate (**eta0**) based on the volatility of the error signal. If drift is high, the learning rate should increase to adapt faster; if stable, it should decrease to converge.
- **Non-Linear Online Models:** Moving beyond linear SGD to non-linear online algorithms, such as **Hoeffding Adaptive Trees** or Kernel-based Recursive Least Squares (KRLS), to capture complex relationships without the full overhead of batch retraining.
- **Ensembling:** Creating a weighted ensemble of several online models with different learning rates (e.g., one "fast" learner and one "slow" learner) to balance stability and adaptability.

6 Experiments and testing

TODO: add outputs of anomaly detection

7 References

Wares, S., Isaacs, J. and Elyan, E. (2019). Data stream mining: methods and challenges for handling concept drift. SN Applied Sciences, 1(11), 1412.

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. (2002, June). Models and issues in data stream systems. In Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems (pp. 1–16).

Vito, S. (2008) Air Quality Dataset. UCI Machine Learning Repository. Available at: https://archive.ics.uci.edu/ml/datasets/Air+Quality (Accessed: 25 November 2025).

TODO: reference *T-digest*