CS 876 - Streaming Data Systems
Implementation of Concept Drift Detection in AIR

Yash Koushik Kocherla
IMT2020033

***Abstract*—This project addresses the challenge of concept drift in dynamic data streams by developing a benchmark setting for implementing distributed concept drift detection algorithms. An Asynchronous Iterative Routing (AIR) dataflow is implemented, including a generator vertex that simulates concept drift, a drift detection vertex utilizing various algorithms, and a collector vertex. Through this framework, the project aims to implement different drift detection algorithms under varying throughputs and drift frequencies. The outcomes of this project will contribute to the development of more robust and efficient solutions for handling concept drift in real-world applications.**

This project makes the following contributions:

1) **A novel implementation of Concept Drift Detection in AIR.**
2) **An implementation of ADWIN detection algorithm.**
3) **An implementation of CUSUM detection algorithm.**

1. Problem Definition:

1.1. Problem:

We aim to solve the problem of efficiently processing and analyzing the drift in concept of the streaming data under consideration by implementing concept drift detection algorithms in AIR [5]. In real-world applications, data streams are dynamic and subject to changes over time. Concept drift, where the statistical properties of the data change, is a common challenge. There is a need to assess the performance of existing distributed concept drift detection algorithms in AIR systems.

Detecting concept drift in streaming data is crucial for real-time applications like fraud detection, stock analysis, and network monitoring. Traditional algorithms often lack scalability and efficiency for large data streams. AIR systems offer a promising platform for distributed, efficient concept drift detection by leveraging their scalability, fault tolerance, and adaptability. By addressing this challenge, we can significantly improve the performance and effectiveness of streaming systems in real-world scenarios.

1.2. Concept Drift:

Concept drift, in data analysis terms, refers to the phenomenon where the underlying relationships and patterns in your data change over time. Imagine it as a moving target - your data model, built on past observations, might no longer be accurate if the target itself is shifting. This can occur due to various factors like seasonal changes, evolving user behavior, or unforeseen events. It's crucial to detect and adapt to concept drift to maintain the accuracy and effectiveness of your data analysis and predictions [4] [3].

1.3. Asynchronous Iterative Routing:

AIR engine is designed from scratch in C++ using the Message Passing Interface (MPI), pthreads for multi-threading, and is directly deployed on top of a common HPC workload manager such as SLURM. AIR implements a light-weight, dynamic sharding protocol (referred to as "Asynchronous Iterative Routing"), which facilitates a direct and asynchronous communication among all client nodes and thereby completely avoids the overhead induced by the control flow with a master node that may otherwise form a performance bottleneck. [6]

2. Approach:

In this project, We make use of the Asynchronous Iterative Routing dataflow. We use the following nodes in our dataflow:

1) Generator
2) ADWIN concept drift detector
3) CUSUM concept drift detector
4) Collector

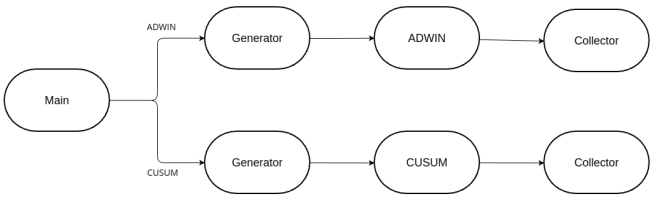

Figure 1. Dataflow

2.1. Generator

The main objective of the generator is to generate a stream of events (here, a list of 5 items with designated weights) with concept drift. It does so by:

1) Defining trending and remaining items: The node defines two sets of items: trending and remaining. It starts with a predefined collection of items and uses random selection to choose a smaller subset as trending. The remaining items are then used later for introducing concept drift.
2) Generating bag of items: Next, the node generates bags of items. Each bag contains 5 items randomly chosen from the currently trending set. These bags are then sorted for consistency and serialized along with their timestamps into messages that can be processed by downstream nodes.
3) Controlling the generating and drift rate: To ensure realistic data generation, the function carefully controls the rate and impact of drift. It achieves this by:
    - Enforcing a throughput limit: The function sleeps for a specific time between each bag generation based on a desired throughput, ensuring the data stream doesn't overwhelm the system.
    - Introducing concept drift: At specific intervals (determined the drift rate parameter), the function replaces one trending item with a previously unused item from the remaining set. This gradual replacement of items simulates the phenomenon of concept drift, where the underlying distribution of data changes over time.

This function acts as a crucial source of data for evaluating and comparing distributed concept drift detection algorithms in streaming systems. It provides a flexible and realistic environment by allowing fine-grained control over the data generation rate, drift frequency, and window size.

```
typedef struct EventCD
{
    long int event_time;
    char bag[15];
} EventCD;
      You, 1 second ago • Uncommitted changes
void Serialization::YSBserializeCD(EventCD *event, Message *message)
{
    char *b = message->buffer + message->size;
    memcpy(b, &event->event_time, 8);
    b += 8;
    memcpy(b, &event->bag, 15);
    message->size += sizeof(EventCD);
}
```

Figure 2. Serialization of Message in generator node

2.2. Adaptive Windowing (ADWIN)

Adaptive Windowing, famously also known as ADWIN is a drift detection algorithm. It works by maintaining buckets of recent data, calculating their mean and variance, and then comparing the variance to a threshold. If the threshold is exceeded, ADWIN triggers a drift event and resets its internal state to adapt to the new concept [1]. This allows ADWIN to effectively detect and react to changes in the underlying data distribution. Algorithm implemented in the node:

1) Data Structure used: ADWIN uses a series of buckets to store the recent data. Each bucket holds the sum and count of elements within a specific timeframe. The code defines a bucket size and continuously updates the buckets as new data arrives.

```
struct Bucket
{
    int size;
    double sum;
};
```

Figure 3. Bucket struct

2) Mean and Variance calculation: ADWIN periodically calculates the mean and variance of the data based on the information in the buckets. The mean estimates the current concept, while the variance assesses the likelihood of a change.
3) Drift Detection: ADWIN compares the current variance to a threshold calculated with a pre-defined delta value. If the variance significantly exceeds the threshold, the algorithm triggers a drift detection event, indicating a potential change in the underlying data distribution.
4) Window Management: Upon detecting a drift, ADWIN resets the buckets and starts accumulating data anew. This allows the algorithm to quickly adapt to the new concept without being influenced by outdated data.

The serialized message in the generator is deserialized using a deserializeCD method and then the result of this node is serialized using the method serializeAdwin.

2.3. Cumulative Sum (CUSUM)

Cumulative Sum, popularly known as CUSUM, is a family of algorithms designed to detect concept drifts in data streams. These algorithms work by maintaining two counters, one for positive and one for negative deviations from an expected value. When the deviations accumulate significantly in one direction, it suggests a potential shift in the underlying data distribution, indicating a concept drift [2].

1) Initialization: Define a reference value, against which deviations will be measured. Then we declare CUSUM+ and CUSUM- (for positive and negative deviations respectively). Set the thresholds

for these deviations which determine the level of deviation required to trigger a drift detection event.

2) Processing of Data Stream: As the datapoints (each bag of 5 items) is being deserialized, we calculate the deviation (in our algorithm the deviation measure used is Euclidean distance) from the reference value. Accordingly we will update the CUSUM+ and CUSUM-. If any of the above two, exceed the threshold, then a drift detection event is triggered.
3) Resetting CUSUMs: Upon drift detection, both CUSUM+ and CUSUM- are reset to 0. This allows the algorithm to adapt to the new concept and start accumulating deviations from the updated reference value.

```
typedef struct EventAdwin
{
    long int event_time;
    char drift[2];
} EventAdwin;

void Serialization::YSBdeserializeCD(Message *message, EventCD *event,
                                     int offset)
{       You, 2 days ago • First ADWIN version
    char *b = message->buffer + offset;
    memcpy(&event->event_time, b, 8);
    b += 8;
    memcpy(&event->bag, b, 15);
}

void Serialization::YSBserializeAdwin(EventAdwin *event, Message *message)
{
    char *b = message->buffer + message->size;
    memcpy(b, &event->event_time, 8);
    b += 8;
    memcpy(b, &event->drift, 2);
    message->size += sizeof(EventCD);
}

void Serialization::YSBdeserializeAdwin(Message *message, EventAdwin *event,
                                        int offset)
{
    char *b = message->buffer + offset;
    memcpy(&event->event_time, b, 8);
    b += 8;
    memcpy(&event->drift, b, 2);

}
```

Figure 4. Serialization of ADWIN and CUSUM

2.4. Collector

Finally at the end of our dataflow, the Collector is responsible for handling the incoming data streams from the drift detection nodes containing information about drift events. The key functions of this node are:

1) Recieving data streams: The code receives messages from other vertices containing EventAdwin structures, each representing an event with its timestamp, and a drift flag.
2) Processing data: For each received EventAdwin structure, the code checks for the presence of a drift flag. If a drift is detected, its timestamp is logged and printed in the terminal for further analysis.

3. Evaluation and Results:

1) Parameters for first evaluation:
   - Generator Throughput: 1000
   - Drift Detection Method: ADWIN
   - Parallel dataflows: 4

```
yash@kyk:~/Desktop/Clg/7th_Sem/SDS/Project/Release$ mpirun -np 4 ./AIR CD 1000 ADWIN 4

*****************AIR (c) 2020 Uni.lu*****************

AIR INSTANCE AT RANK 2/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 4/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 1/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 3/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
Drift detected at event-time: 3143977
-----------------------------------------------------
Drift detected at event-time: 3143987
-----------------------------------------------------
Drift detected at event-time: 3143977
-----------------------------------------------------
Drift detected at event-time: 3143977
-----------------------------------------------------
Drift detected at event-time: 3143987
-----------------------------------------------------
Drift detected at event-time: 3143987
```

Figure 5. Terminal output

Checking the correctness of the first drift detection, that is at timestamp 3143977:

```
3143985,11 3 4 6 7
3143984,11 3 4 6 7
3143983,12 3 4 6 7
3143982,12 3 4 6 7
3143981,10 12 3 6 7
3143980,10 12 3 6 7
3143979,10 12 13 3 6
3143978,10 12 13 3 6
3143977,10 13 14 3 6
```

Figure 6. generator output

2) Parameters for second evaluation:
   - Generator Throughput: 1000
   - Drift Detection Method: CUSUM
   - Parallel dataflows: 4

```
yash@kyk:~/Desktop/Clg/7th_Sem/SDS/Project/Release$ mpirun -np 4 ./AIR CD 1000 CUSUM 4

*****************AIR (c) 2020 Uni.lu*****************

AIR INSTANCE AT RANK 1/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 2/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 3/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
AIR INSTANCE AT RANK 4/4 | TP: 1000 | MSG/SEC/RANK: 2 | AGGR_WINDOW: 10000ms
Drift detected at event-time: 2880991
-----------------------------------------------------
Drift detected at event-time: 2880989
-----------------------------------------------------
Drift detected at event-time: 2880981
-----------------------------------------------------
Drift detected at event-time: 2880991
-----------------------------------------------------
Drift detected at event-time: 2880991
-----------------------------------------------------
Drift detected at event-time: 2880989
```

Figure 7. Terminal output

Checking the correctness of the first drift detection, that is at timestamp 2880991:

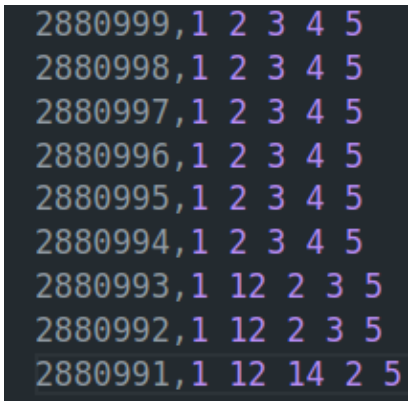

Figure 8. generator output

3) I have extensively experimented with various generator throughputs and dataflows. In general, by observation, CUSUM performed better than ADWIN in terms of detecting the concept drift early, i.e CUSUM was more sensitive the to kind of data that we were considering, than ADWIN.

4. Challenges Faced:

1) Drift Detection Algorithms in CPP: The implementation of drift detection algorithms posed a significant challenge as there were no existing libraries available in CPP for this purpose. Consequently, both drift detection algorithms had to be developed from scratch to meet the project requirements.
2) Segementation Faults!!!: Integrating the CPP-developed generator and drift detection algorithms into the AIR architecture presented difficulties, resulting in segmentation faults. Debugging and resolving these issues were crucial to ensuring the seamless functioning of the components within the system.
3) Serialization and Deserialization: The process of serialization and deserialization introduced confusion, specifically regarding the optimal points in the data flow to perform these operations. Clarifying the exact locations for serializing events and deserializing them in downstream nodes became a critical aspect of ensuring data consistency and proper communication within the architecture.

5. Future Aspects:

1) Implementation of Additional Drift Detection Algoritms: Extend the project scope by implementing other drift detection algorithms such as the PH-test (Page Hinkley test) and DDM (Dynamic Distance Measure). Following a similar methodology as discussed in this project, these algorithms can provide additional insights into concept drift detection.
2) In-Depth Analysis of Drift Detection Methods: Conduct a comprehensive analysis of various drift detection methods within the AIR framework. Evaluate their scalability and efficiency in real-world scenarios, considering factors like the volume of incoming data and the system's latency in detecting concept drifts.
3) Generator Enhancements: Enhance the data generation capabilities by developing a versatile generator. This generator should offer flexibility in generating data based on specific needs, including scenarios like incremental drift and sudden drift.
4) User-Friendly Interfaces: Develop user-friendly interfaces or dashboards to facilitate easy configuration and monitoring of the drift detection system. Providing intuitive tools for users to interact with and customize the drift detection parameters will enhance the usability and accessibility of the overall solution.

References

[1] https://riverml.xyz/dev/api/drift/ADWIN/.

[2] https://www.aporia.com/learn/data-drift/concept-drift-detection-methods/.

[3] Adriana Sayuri Iwashita and João Paulo Papa. An overview on concept drift learning. *IEEE Access*, 7:1532–1547, 2019.

[4] Jie Lu, Anjin Liu, Fan Dong, Feng Gu, João Gama, and Guangquan Zhang. Learning under concept drift: A review. *IEEE Transactions on Knowledge and Data Engineering*, 31:2346–2363, 2019.

[5] Vinu Venugopal, Martin Theobald, Samira Chaychi, and Amal Tawakuli. Official github repo for air. https://github.com/bda-uni-lu/AIR/tree/master.

[6] Vinu Venugopal, Martin Theobald, Samira Chaychi, and Amal Tawakuli. Air – a light-weight yet high-performance dataflow engine based on asynchronous iterative routing, 01 2020.