Efficient Datastream Sampling on Apache Flink

MARTHA VLACHOU-KONCHYLAKI
Efficient Data Stream Sampling on Apache Flink

MARTHA VLACHOU-KONCHYLAKI
marthavk@kth.se

School of Computer Science and Communication (CSC)
Master's programme in Machine Learning
Supervisor: Jens Lagergren
Examiner: Johan Hästad
Project Provider: SICS SWEDISH ICT
March 7, 2016
Abstract

Sampling is considered to be a core component of data analysis making it possible to provide a synopsis of possibly large amounts of data by maintaining only subsets or multisubsets of it. In the context of data streaming, an emerging processing paradigm where data is assumed to be unbounded, sampling offers great potential since it can establish a representative bounded view of infinite data streams to any streaming operations. This further unlocks several benefits such as sustainable continuous execution on managed memory, trend sensitivity control and adaptive processing tailored to the operations that consume data streams.

The main aim of this thesis is to conduct an experimental study in order to categorize existing sampling techniques over a selection of properties derived from common streaming use cases. For that purpose we designed and implemented a testing framework that allows for configurable sampling policies under different processing scenarios along with a library of different samplers implemented as operators. We build on Apache Flink, a distributed stream processing system to provide this testbed and all component implementations of this study. Furthermore, we show in our experimental analysis that there is no optimal sampling technique for all operations. Instead, there are different demands across usage scenarios such as online aggregations and incremental machine learning. In principle, we show that each sampling policy trades off bias, sensitivity and concept drift adaptation, properties that can be potentially predefined by different operators.

We believe that this study serves as the starting point towards automated adaptive sampling selection for sustainable continuous analytics pipelines that can react to stream changes and thus offer the right data needed at each time, for any possible operation.
Effektiv dataströmsampling med Apache Flink

Referat

Sampling anses vara en central del av dataanalys som gör det möjligt att ge en sammanfattning av stora mängder data genom att upprätthålla endast delmängder eller multidelmängder av den fullständiga datamängden. I kontexten av strömmande data, en typ av dataprocessning där data antas vara obegränsad, är sampling användbart eftersom sampling kan skapa begränsade beskrivningar som är representativa för oändliga dataströmmar. Sampling ger flera fördelar såsom körningar som använder en begränsad mängd minne, kontroll över trendkänslighet, och adaptiv behandling anpassad till användarfallet.

Huvudsyftet med denna avhandling är att genomföra en experimentell studie för att kategorisera befintliga samplingsmetoder för ett urval av egenskaper som härrör från vanliga användarfall. För detta ändamål har vi utformat och implementerat ett testsystem som tillåter en konfigurerbar samplingspolicy under olika bearbetningsscenarier tillsammans med ett bibliotek av olika samplingsmetoder. Vi bygger på Apache Flink, ett distribuerat strömbearbetningssystem för att skapa denna testbädd. Vi visar i vår experimentella analys att det inte finns någon optimal samplingsmetod för alla användarfall. I stället finns det gemensamma scenarion, så som online-aggregering och inkrementell maskininlärning, som kräver olika samplingsmetoder. Vi visar att varje samplingspolicy är en avvägning mellan bias, känslighet, och anpassning till konceptdrift, egenskaper som kan potentiellt vara fördefinierade i olika Flink-operatorer.

Vi tror att denna studie fungerar som utgångspunkt för adaptivt val av samplingsmetod för hållbara kontinuerliga analyspipelines som kan reagera på strömmande förändringar och därmed erbjuda rätt data vid varje tidpunkt, för alla användarfall.
Acknowledgments

I would like to thank all the people who contributed, even the slightest, in the realization of this work.

This includes, first of all, my supervisor from SICS, Paris Carbone, for his continuous feedback and motivation. I would barely make it without his help and guidance.

Furthermore, I would like to thank all the people at SICS who helped me through the process, Faye, Daniel, Vassia, Atie and Guyla, by making this a fun experience and for all the delightful moments we shared together.

A huge thank you to my friends Makis, Irene, Dimitris, Evita and Eleni who put up with me in the most stressful periods of this process.

Finally, I owe my best of gratitude to my parents, George and Elina; without their unconditional love and support I would hardly be the person I am today. Thank you for believing in me.
Contents

1 Introduction 1
  1.1 Motivation ......................................................... 1
  1.2 Scientific Question ............................................. 3
  1.3 Contribution ..................................................... 3
  1.4 Methodology .................................................... 3

2 Background 5
  2.1 Datastreams ..................................................... 5
  2.2 Sampling ........................................................ 6
  2.3 Change Detection ............................................... 6
    2.3.1 The Page Hinkley Test .................................... 7
  2.4 Apache Flink .................................................... 7
    2.4.1 The datastream model in Apache Flink ....................... 8
    2.4.2 Operators .................................................. 10
  2.5 Related Work ................................................... 11
    2.5.1 Traditional Sampling Schemes ............................. 11
    2.5.2 Modern Approaches ...................................... 12

3 System Analysis 17
  3.1 Testbed Overview .............................................. 17
  3.2 Implemented Sampling Policies ................................. 19
    3.2.1 FiFo Sampler ............................................. 19
    3.2.2 Reservoir Sampler ...................................... 20
    3.2.3 Biased Reservoir Sampler ................................ 21
    3.2.4 Windowed Sampling ...................................... 21
    3.2.5 The Greedy Sampler ..................................... 23
  3.3 Implementation ............................................... 26
    3.3.1 Sampling as a stream operator ............................ 26

4 Experimental Driven Analysis 33
  4.1 Evaluation Synopsis .......................................... 33
  4.2 The Datasets .................................................. 33
  4.3 Experiments ................................................... 35
Chapter 1

Introduction

In today’s world where information is dominant, it is more and more common for applications to deal with the unbounded data concept, rather than process bounded, non-evolving, batch datasets. Even in the static case, the amount of data to be processed and queried is so large that only one-pass algorithms are easy to afford. What is more, in the datastream model, data points arrive constantly in unpredictable rates, which is why sampling the datastream seems a natural choice when we search for reducing processing cost and overhead.

An additional characteristic of the dynamic datastream model, compared to the stationary batch analysis, is that the distribution of the data points frequently evolves over time. Consider weather measurements, for example, or Twitter trends concerning particular artists. There is high probability that the temperature or humidity changes over different seasons, as well as the artist’s reputation over time. A streaming application should consider each particular case and deal with concept drift for qualitative knowledge mining and model training.

1.1 Motivation

In the datastream model, items arrive fast and in a way that the end of the stream, if there is any, as well as the total size of the arriving elements, is unknown. In the modern world, such applications consist of IP traffic data, financial data (stocks or currency trades), dedicated network systems (meteorological data, satellite images, earth geodetics), sensor networks (seismic, movement, geolocation) or web data (emails, social network graph streams, tweets, browser clicks). The amount of arriving data is quite extensive to store in a database, therefore every query or operation on the stream must be done in real time, in a finite number of passes - optimally one. In those online applications where data evolve over time, it is a challenge to incorporate new data into learning models.
Sampling is considered a core component of data analysis, more specifically in such streams where keeping an accurate and representative sample of the data is imperative, when space and computational constraints are demanding. The problem of online sampling has been researched several times [36] [8] [7] [23] but approaches are limited by specific goals and no consideration of data parallelism whatsoever.

The main motivation of this thesis stems from a set of use cases which demand a more sophisticated datastream manipulation. More specifically, our aim is to use sampling in order to handle scenarios such as:

- **Depleted clustering resources**: In situations where resources are limited, resampling the datastream with a predefined sampling rate may serve as a stability factor for further tasks.

- **High-speed datastreams**: In such datastreams instead, one can use a sample to slow down the rate of stream.

  In both cases sampling the datastream, in the fashion suggested in this thesis, ensures a stable emission rate, decoupled of the datastream speed.

- **Machine Learning Models**: Use a sampled datastream to train classifiers or run clustering algorithms. The goal is to decrease the computational cost and learning time emerging from including all data points in the learning process, especially in algorithms whose performance time is proportional to the training size.

- **Trend Adaptation**: In the real time scenario, where the data distribution evolves over time, the sampling algorithm should adapt to stream evolution.

- **Sensitivity Control**: The sampling scheme should be able to adjust the sensitivity of the sample to noisy datastreams or outliers.

- **Stateful jobs**: Approximate the history of the datastream in applications where data distribution tends to shift over time.

- **Memory Resources**: In the unbounded datastream scenario, the need to develop sustainable and accurate applications is imperative. Keeping an accurate, yet bounded sample of the datastream, with managed, preallocated memory, can significantly improve the sustainability of a streaming application.

Our goal is to create a general-use stream-sampling component which acts in a distributed fashion, keeping an adequate approximation of the data distribution, also reflecting stream evolution. The sampling component is subject to a set of parameters which define the sampling policy, the extent to which the output datastream is adapted to change and the speed of the sampled stream. In a higher level, the parametric sampling scheme serves as a sampling optimization component within a data processing framework.
1.2 Scientific Question

There are three main scientific questions addressed in this thesis:

- What are the properties of existing stream sampling algorithms and how do these properties apply to different use cases? (i.e. how accurate are the samples, how fast do they adapt to concept drift)

- Can an adaptive streaming algorithm deal with concept drift or other constraints while also maintaining its overall desirable properties?

- How can we define a sampling optimization component which adapts its sampling policy to different scenarios?

In detail, as far as the sample’s adaptability to change is concerned, the sampler should be able to manage the trade-off of fitting stream evolution as fast as possible but also representing the long-term stream concept. User policies such as data eviction or time windowed sampling should be also supported. However, the biggest challenge and impact of the sampling technique is efficiently dealing with high ingestion rate i.e. hard space and IO constraints profiting from the scalability of a distributed solution while also preserving accuracy.

This sampling technique should also improve accuracy of incremental or historic learners when space and computational resources are limited since accurate samples, instead of the whole datastream, can be used for faster training but equally accurate models.

1.3 Contribution

In this thesis, in accordance to the scientific question posed in 1.2, we create a distributed version of a set of existing sampling techniques and implement a novel sampling policy handling concept shift scenarios, alongside a use-case complete evaluation framework for aiding the categorization and evaluation of sampling techniques under different workloads, types of data and consumer purposes. We also perform an experimental analysis of the above components, evaluating the performance of each sampler on different tasks, contributing towards the design of a sampling optimization plan for distributed streams.

1.4 Methodology

This is a quantitative study and the methodology is an experimentally-driven comparison of different stream sampling algorithms based on the following metrics:
• The similarity of a sample to the overall stream distribution in terms of distribution parameters (e.g., Bhattacharyya Distance).

• The accuracy of a sample when used in specific tasks (e.g., Classification)

• Adaptation to stream evolution (concept drift) by measuring the online similarity of a sample to the actual changing distribution over evolving streams.
Chapter 2

Background

In this chapter we will examine the basic concepts of datastreams and sampling, and describe the main architecture and pipeline of a Flink program (job), as well as how operators work in this framework. Finally, we cite a collection of traditional sampling schemes and related work in terms of novel datastream sampling approaches. The sampling schemes used for our implementation will be explained in Chapter 3 along with their implementation details and parameters.

2.1 Datastreams

Muthukrishnan [29] describes the datastream as a representation of input data that are coming at a very high rate and defines three types of datastream models:

- Time Series Model
- Cash Register Model
- Turnstile Model

Each model reflects a map operation which maps every incoming element to a point in \( \mathbb{R} \). For any of the above streaming models we assume that all data points are subjects to a common data structure and arrive causally ordered, through a FIFO channel, at unknown rates. We also assume that the stream is unbounded, in the sense that the amount of the arriving data points is infinite.

The time series model implies that each incoming element, \( a_i \), is mapped to itself, while the cash register model maps each item to an increment. For the time series model, one can consider datastream applications such as customer clicks, stock market indexes, sensor network data e.t.c. The cash register model, on the other hand, imitates the behavior of a cash register where only increments to data points are accepted. In that case, the incoming datastream consists of tuples \( (a_i, I_i) \) where \( I_i \) is the increment which corresponds to \( a_i \). Finally, the turnstile model includes the case of \( I_i \) being a negative quantity. Another model, similar to the turnstile
model, is suitable for geometric problems and handles the case of insertions and deletions of data points in a discrete geometric space [25] [18]. These datastreams are defined as dynamic datastreams.

2.2 Sampling

Sampling in the datastream concept is the procedure of selecting at random a representative subset of the datastream [26] [29]. This procedure is controlled by random variables, it is performed on a set of streaming data points and creates a subset of the original set, usually noticeably smaller, which represents, in some, if not in every feature, the original dataset. According to Strinivasan [34], the sampling hypothesis is defined as follows:

A system equipped with a sampling method constructs in lesser time, a theory that is no worse in predictive accuracy to that obtained from the same algorithm without a sampling method.

The sampled subset should be, at any time, a representation of the stream seen so far that is keeping the properties of the stream according to some rules, defined usually by the type of the application. The sample can be uniform or biased [24] and can be further used to querying the whole stream. The results of the queries on the sample will be an approximation compared to the whole stream but, according to the error introduced by each sampling method, they will be within certain confidence bounds. In the same sense, the sample can be used to train learning models where the cost of training with numerous data is high. In a k-nn classifier, for example, where every item must be compared to all of its neighbors, a sample may provide the same accuracy but with a lower processing cost than using the entire stream [3]. Similarly, ensemble methods such as bagging and boosting proceed by resampling the dataset the times needed to build the respective weak models to combine into the ensemble model.

2.3 Change Detection

In real-world scenarios, the underlying logic producing the data stream often changes with time. There is, quite often, a dynamic process controlling the underlying hidden variables and, as a result, the observable data points. Consequently, there exists a probability of change in the hidden variable, thus a change in the observable data points [19]. The name of this change of concept is deduced by the rate of the drift. More specifically [32]:

The term Concept Shift refers to abrupt changes, while the term Concept Drift is associated to gradual changes in the target concept.
The detection of that change of concept, especially when that change occurs in a low rate, is a non-trivial problem. In fact, there exist a number of approaches for this problem, the most significant being the ones outlined in [32]. Those are namely the Statistical Process Control (SPC) [20], which tracks the probability of error exceeding a certain threshold, the Adaptive Windowing method (ADWIN) [11], which uses a sliding window to detect drift in the datastream, the Fixed Cumulative Windows Model (FCWM) [33] which compares histograms between a past and a current window, and, finally, the Page Hinkley Test (PHT) [30] [28], which detects changes in a Gaussian signal.

2.3.1 The Page Hinkley Test

The latter, PHT, is the method we chose to use since we target at monitoring change in data points drawn from a Gaussian distribution. PHT only detects an increase of the mean of the underlying Gaussian distribution and is used mainly to detect an increase in a model error, e.g. classification error [28]. The test is carried out as follows [30]:

1. Set a cumulative variable

\[ U_T = \sum_{t=1}^{T} (x_t - \bar{x}_T - \delta) \tag{2.1} \]

where:

- \( x_t \) is the current mean
- \( \bar{x}_T = \frac{1}{T} \sum_{t=1}^{T} x_t \)
- \( \delta \) is a parameter tuning the size of the changes that are detected.

2. Track difference between the minimum of \( U_T \) and \( U_t \):

\[ PH_T = U_T - \min(U_t, t = 1 \ldots T) \]

3. A change is detected if this difference exceeds a certain threshold:

\[ PH_T > \lambda. \]

2.4 Apache Flink

For the sampling task all implemented algorithms perform as stream operators in the Apache Flink framework. Apache Flink [2] is an open-source, parallel data processing engine, equipped with a Java and Scala API, supporting both batch and unbounded data stream processing. Flink’s main flow architecture consists of transformations (such as map, reduce etc.) on batch (DataSet) or streaming (DataStream) data. Such transformations are being performed by the respective operators. Flink’s core runs on a streaming engine, thus, even batch processing scenarios are treated as special stream processing cases. Still, in this study we
consider the general streaming concept, developing operators transforming solely 
DataStream objects.

### 2.4.1 The datastream model in Apache Flink

Figure [2.1](image) represents the main architecture of the Flink system. The upper layer 
depicts all supported libraries, split in the two main sets: the batch and streaming 
APIs. Below the API layer lies the Optimizer. In runtime, a Flink program will be 
translated to a Direct Acyclic Graph (DAG) of operator-nodes and stream-edges 
and will be optimized through the optimizer layer. The DAG is then sent to the 
runtime environment which is responsible for the actual execution.

A streaming Flink program imitates the procedure depicted in [2.2](image) The main 
pipeline consists of three basic stages: the SourceFunction, the StreamOperators 
and finally the SinkFunction.

- A source is generating the actual stream. The source may listen to a particular 
socket or generate a stream from an input file. In this word count example 
the source reads the file where the text is stored and creates a stream of one 
element, the whole text.

- The datastream which is produced by the source is then, being transformed 
by a number of operators according to the needs of the application. A 
flatMap function, for example, would split the text into words and transform 
the datastream into a stream of words.
• After all transformations have been done the output of each execution is being pushed to a **sink** which consumes the transformed stream according to a predefined function (print to the console, update a database etc).

**A simple word count example**

The main execution code for a simple word count example can be seen in the listing 2.1. The source creates the initial datastream and the `flatMap` operator (Listing 2.2) splits the text and produces a stream of tuples, containing, each one, a word and one increment. A partitioning policy, `groupBy`, is used to group tuples by their “word” field so that tuples containing the same word will end up in the same processing unit.

```java
public static void main(String[] args) throws Exception {
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();
    //Create a text source
    DataStream<String> text = env.fromElements(WordCountData.WORDS);
    text.flatMap(new WordMapper()) // Split the text by space characters into words and map each word to an increment of 1
        .groupBy(0) //Group stream by word. Tuples containing the same word will be sent to the same processing unit
        .reduce(new WordReducer()) //Count the words
        .print(); //sink
    env.execute("Word Counter");
}
```

**Listing 2.1. main function for the word count example**

The `reduce` operation is performed by the `WordReducer` on the grouped datastream and acts as an aggregator, counting the words for each group (Listing 2.3). The datastream is then consumed by a sink, which, in this case, is a `print` operation. The job graph of this simple word count example can been seen in figure 2.2.

```java
class WordMapper implements FlatMapFunction<String, Tuple2<String, Integer>> {
    @Override
    public void flatMap(String value, Collector<Tuple2<String, Integer>> out) {
        String[] words = value.split(" ");
        for (String word : words) {
            out.collect(new Tuple2<String, Integer>(word, 1));
        }
    }
}
```

**Listing 2.2. flatMap operator for the word count example**

![Job Graph](image-url)
class WordReducer implements ReduceFunction<Tuple2<String, Integer>> {
  @Override
  public Tuple2<String, Integer> reduce(Tuple2<String, Integer> value1,
    Tuple2<String, Integer> value2) throws Exception {
    return new Tuple2<String, Integer>(value1.f0, value1.f1+value2.f1);
  }
}

Listing 2.3. reduce operator for the word count example

2.4.2 Operators

Each transformation on the datastream can be considered as an operation. Operators, are the procedures by which the datastream is transformed. According to Hirzel et al. [22]:

A stream operator consumes data items from incoming streams and produces data items on outgoing streams.

In the Apache Flink Streaming system, operators are stateful with various selectivities, running in parallel instances. Using Hirzel’s definition again [22]:

The selectivity of an operator is its data rate measured in output data items per input data items.

Map operations, for example, have a selectivity of 1.0 while the selectivity of flatMap operation can be defined by the user. Some fundamental transformations supported by a Flink program are described below:

map: In a map operation every output is the outcome of a certain function given the input element.

flatMap: flatMap is a more abstract version of the map operation. It supports triggering one, multiple or no outputs on the arrival of an input element.

filter: filter is a boolean operator which refines the datastream by blocking incoming elements according to some user defined function.
reduce: A reduce operator combines repeatedly incoming elements and produces a reduced output value of the same type.

2.5 Related Work

The datastream sampling task has been researched over the years, with Vitter [36] providing the first algorithm for uniform sampling in unbounded data streams. The concept of sampling has evolved since then and a variety of algorithms have emerged addressing several problems [24]. In the following section we present several traditional and modern sampling schemes so as to better outline the variety of approaches intended for solving the datastream sampling task.

2.5.1 Traditional Sampling Schemes

Bernoulli Sampling

The simplest form of sampling in datastreams is the Bernoulli Sampling [31]. It is the less sophisticated method relying mostly in a simple accept/reject (A/R) operation. At each arriving element a coin is flipped with predefined success probability \( p_i \). In success of the coin flip, the element is included in the sample, otherwise it is ignored. This is a simple operation, easily implemented and, provided that the length of the stream is known, it produces a uniform sample of the incoming stream. However, the major drawback is that the size of the sample is uncertain.

Reservoir Sampling

Reservoir Sampling, introduced by Vitter [36], provides a simple way to fill a preallocated buffer, called a reservoir, with uniformly sampled elements from the datastream. The main feature of a reservoir is that, at each point in time, it contains a uniform sample from the entire datastream progress. The idea behind the algorithm is that each arriving data point is being sampled with a probability proportional to its order in the stream and replacing an item in the reservoir uniformly at random, ensuring the uniformity of the sample at any point in time. More details on the reservoir sampling scheme along with its implementation details will be discussed in Chapter 3.

Windowed Sampling

In the presence of window constraints, active items belong in a certain window, defined with reference either in time (e.g. 1 day window), or in the order of the arriving items (e.g. 1000 data points). Babcock [8] has suggested two approaches for sampling with replacement under such constraints. In both approaches the main idea relies on a chain of future replacements for every item in the sample. Upon their expiration, items are evicted and their descendant takes their place. Both
algorithms, namely *chain sampling* and *priority sampling*, will be further discussed in Chapter 3.

**Min-Wise Sampling**

Min-Wise Sampling [14] is an algorithm producing a uniform sample of size $N$ in an online, one-pass fashion. The main idea is that each incoming element is assigned a number inside the interval $[0, 1]$ uniformly at random. At each time only the $N$ elements with the lower assigned values are kept in the sample.

**AMS Sampling**

The AMS Sampling [6] is a more sophisticated algorithm for calculating frequency moments than keeping counters for each value. AMS Sampling picks a random item on the stream and counts all upcoming elements having the same value as the picked item. The $k$-th frequency moment is, then, computed as a function of the length of the stream, the count of all upcoming elements (i.e. elements arriving after the randomly selected item) and the order of the moment. In order to calculate the expected value and get an $(1 \pm \varepsilon)$-estimate of $F_k$ with probability $1 - \delta$ one should run $O\left(\frac{n^{1-1/k}}{\varepsilon^2}\right)$ parallel instances of the described procedure and calculate average $A$ and median $M$ over $O\left(\frac{1}{\delta}\right)$ averages.

**2.5.2 Modern Approaches**

The following approaches compose a collection of more sophisticated, task-oriented sampling approaches, regarded as the state-of-the-art when it comes to datastream sampling.

**Count-Min Sketch [15]**

A Count-Min Sketch, although not a sampling algorithm, produces a summary of the streaming progress in an array of predefined size. The array holds increments of the, repeatedly hashed, incoming data points using a set of random hash functions. The rows of the array correspond to a unique random hash function, producing an integer output, while the columns represent the output range (identical for all functions) of the functions. Each item is hashed, upon arrival, with every hash function and, for each row of the array, the respective column is increased by 1. The constructed array as well as its building procedure are also displayed in figure 2.3.

**Adaptive-Size Reservoir Sampling [5]**

This study addresses the problem of dynamic reservoir size. In order to achieve uniformity in the samples the authors introduce the uniformity confidence measure.
In particular:

\[ UC = \frac{\text{the number of different samples of the same size possible with the algorithm}}{\text{the number of different samples of the same size possible statistically}} \]  

The algorithm (ARS) is based on a simple idea: use uniform, random sampling (reservoir sampling) until the size of the sample changes. If the reservoir decreases, then evict elements from the buffer uniformly at random. If the reservoir capacity increases, then, given a UC threshold, the algorithm searches for the minimum number of elements to arrive in order to fill the new sample, so that UC does surpass this threshold. Then randomly evict a number of items from the sample and fill the rest of the buffer with arriving items using uniform random sampling.

In a further study [4], where the stream consists of a set of substreams, there is a reservoir allocated for each of those substreams, responsible for keeping a uniform sample of the elements belonging to each substream. The size of each reservoir relates to the length of the substream seen so far. In that case, the ARS algorithm is used for keeping the separate adaptive reservoirs.

**Biased Reservoir Sampling [3]**

The biased reservoir sampling scheme [3] relies on the traditional reservoir scheme, but also introducing the concept of stream evolution resulting in the creation of a biased (towards recent elements) sample. The biased sample \( S(t) \) is created through a bias function \( f(r,t) \) which provides a relationship between the \( r \)-th and the \( t \)-th data point \( (r \leq t) \), proportional to the probability \( p(r,t) \) of the \( r \)-th point belonging in the sample when the \( t \)-th point arrives. Aggarwal [3] makes use of the properties of the class of memory-less bias functions using \( f(r,t) = e^{-\lambda(t-r)} \) as a bias function where \( \lambda \) stands for the bias rate.

When the space constraints are soft and much smaller than \( 1/\lambda \) then each new item is added to the reservoir deterministically. However, with probability equal to the proportion of the occupied slots in the reservoir \( (F(t)) \) the new item will replace one already sampled item while with complementary probability it will be appended to
the reservoir increasing its size. The use of $F(t)$ as a policy for adding or replacing new elements in the reservoir is introducing the exponential bias in the algorithm. The first items of the stream will quickly fill up the reservoir while as more items arrive the probability of the new items replacing old samples will increase. In an application with strong space constraints the same procedure is followed with a slight modification: the arriving item is not sampled deterministically but with a probability $p_{in} = n \cdot \lambda$.

**Sampling in Dynamic Streams** [18]

This sampling technique addresses the problem of maintaining a random sample of a set of items belonging in a discrete geometric space from a datastream containing insertions and deletions of points in space [18]. If $[U] = \{0, \ldots, U-1\}$ is a finite universe where the streamed insertions and deletions are applied and $M$ is the maximum multiplicity of an item in this universe the goal defined by Frahling et al. is to build a data structure which allows for operations on a vector $x : [U] \rightarrow [M]$. They introduce two minimal data structures: Unique and Distinct element data structures which support **Update** and **Report** operations. These minimal data structures along with a set of hash functions are used to support the two higher operations on the stream: **Update** and **Sample**.

**Sampling from Distributed Streams** [16]

In this paper Cormode et al. address the problem of producing a random sample from $k$ distributed streams minimizing communication cost and element processing time. They propose a set of algorithms to perform random sampling with and without replacement (WR and WoR respectively) for infinite, sequence based or time based windows (IS, SS and TS respectively). The main architecture of the distributed sampling problem is also depicted in 2.4 and consists of a set of $k$ sites $S_k$ and one centralized coordinator $C$ who communicates with each site. Every element arriving in each site contains one apparent timestamp and a hidden index and the coordinator must form a uniform sample from all arriving elements.

The main body of each variation (namely ISWoR, ISWR, SSWoR, SSWR, TSWoR and TSWR) uses binary Bernoulli sampling (a variation of Bernoulli sampling discussed in section 2.5.1) to assign one binary string to each incoming element. Then the prefix of this string is used and, according to the sampling probability, all elements with prefix $0^j$ are included in the sample. This produces a uniform sample with each element having the probability $p = 2^{-j}$ of being included in the sample. In particular:

- The **coordinator** keeps the final sample and ensures that every site is aware of the minimal binary value at each time. It also broadcasts signals when each new round is starting, i.e. the sampling probability is reduced by half.
Each site is responsible of when to send an incoming element to the coordinator.

**ASP-Trees [21]**

Adaptive Spatial Partitioning trees are used for sampling multidimensional datastreams. In this algorithm $d$-dimensional datastream input items are considered and an ASP, $2^d$-ary tree is used as the main storing unit. Each $2^d$-ary split on one node splits the space into $2^d$ parts. If, for example, the data-stream contains points in space (thus two-dimensional points) the first split on the root node will define 4 sub-planes (4 boxes). Each leaf node of the tree contains a counter with a certain capacity which is keeping track of elements falling into the box it represents. More specifically, as points arrive the leaf-counter corresponding to the box they belong increases until it reaches its maximum capacity. When that is reached, the node splits, again, into $2^d$ leaves which will define a sub-partition of the parent box. The algorithm supports refine and unrefine operations which subdivide and merge boxes respectively. Figure 2.5 depicts the data-structures used by the ASP algorithm.
Figure 2.5. ASP-tree data structure representation (figure from [21]).
Chapter 3

System Analysis

In this chapter, we intend to explain the core logic of the proposed system, which is the topic of this thesis. To begin with, there are two composite parts on which we operate on: In a technical level, we introduce all sampler components as operators in a distributed streaming environment. Then, we create a parametrized testbed which uses the samplers for evaluation purposes. In this system, the sampling policy along with the input, task and evaluation, are parametrized, plug-in components. Figure 3.1 depicts this pipeline as well as the components composing it which are subsequently decomposed in this chapter.

3.1 Testbed Overview

The testing framework in figure 3.1 builds a sampler evaluation pipeline based on user defined parameters. Each component is parametrized so as to fit testing requirements. Consequently, the source generator is subject to training set parameters, the samplers to sampling policies and the evaluation to a user defined evaluation metric. In particular, the testing framework assesses the quality of each sampler for a number of tasks such as concept drift adaptation, incremental classification, aggregates etc. For each experimental program execution, an input configuration is defining the following fundamental parameters, along with a set of secondary settings such as source generator, classification and concept drift detection parameters:

- Buffer size: the size of the actual number of sampled records that are being stored. The buffer size can affect the sampling quality: larger buffer sizes produce a more accurate representation of the stream.

- Sampling algorithm: The logic which is used for updating the buffer, thus keeping the sample.

- Emission Rate: The emission rate is the rate we randomly select a record from the sample and emit it to the operator’s output. The rate affects the actual ingestion rate we want to achieve in a streaming topology and ensures stability on the speed of the stream.
The evaluation testbed considers all parameters and builds a representative pipeline: generator - sampler - task - evaluator:

- The **generator** produces a synthetic datastream or streams an existing dataset from an input file. The generator component handles all input distribution parameters (distribution type and parameters such as mean and standard deviation for a Normal distribution) as well as concept drift (rate, probability, shift step). For example, the generator should be able to create data points drawn from a Normal evolving distribution with initial mean $= 0$ which will shift with step $= 0.1$ every $n$ generated instances. In the case of a file input, the generator source reads and parses the file, then streams it accordingly.

- The **sampler** operator is the component where the core sample creation logic of each stream sampler is plugged-in. This component is subject to the inner buffer size, the sampling method and the emission rate.

- The **evaluator** should assess the overall or online sampling quality on each instance. The evaluator component is responsible for evaluating the state of the inner buffer in terms of the parameters of the distribution, measuring the rate of adaptation of the sample to concept drift, computing the accuracy of an external component (e.g. classifier, aggregator) or assessing the overall representativity of the final sample against stream history.

The target properties of our sampling scheme are evaluated through the testing framework in order to assess and compare sampling quality for each policy.
3.2 Implemented Sampling Policies

In this section we present all the sampling policies implemented for our analysis. Those include a set of algorithms performing the sampling task but focusing on different scenarios (e.g. adaptation to stream evolution, time windows etc.). More specifically, we chose to compare two types of reservoir sampling methods, namely the Reservoir Sampler and the Biased Reservoir Sampler, two samplers operating on sliding windows (the Chain and Priority samplers), a novel policy dealing with more sophisticated scenarios such as concept drift (the Greedy Sampler) as well as a sampling policy designed for serving as a baseline (FiFo Sampler).

Before further analyzing each technique, we define a number of concepts, used subsequently to describe each procedure. In particular, for each sampler we define:

- \( a_i \): the \( i \)-th item to arrive
- \( \text{buffer}[N] \): the array which contains the sample of the datastream seen so far
- \( N \): the maximum capacity of the buffer
- \( \text{policy} \): the logic for updating the buffer
- \( w \): the time or index based active window

Since, in most of the sampling policies, the update of the sample is determined by a random variable, we also define the coin-toss scenario which imitates the flip of a biased coin. The bias is injected into the probability with which the toss is successful (the outcome is heads). Thus, when stating "we flip a coin with probability \( p \)" we refer to the scenario where the probability of a coin-toss success is \( p \).

3.2.1 FiFo Sampler

The FiFo Sampler is a trivial implementation of a sampler, in the sense that its update action consists of maintaining a buffer of the last \( N \) arriving elements, thus imitating the behavior of a queue. The procedure is deterministic, since there is no random variable controlling the sampling and the sampler serves basically as a baseline for comparison with the rest, more sophisticated policies.

**Algorithm 1: FiFo Sampling Algorithm**

```plaintext
foreach arriving item \( a_i \) do
    if \( \text{buffer} \) has reached maximum capacity then
        remove oldest element from \( \text{buffer} \)
    append \( a_i \) to the \( \text{buffer} \)
```

19
3.2.2 Reservoir Sampler

Reservoir Sampling [36] is the simplest form of uniform sampling for datastreams. The idea is keeping a reservoir (a dynamic array or similar data structure) which, at any moment, contains a uniform sample of size $n$ from the datastream seen so far. In particular, the first $k$ items of the stream are sampled with probability $p = 1$ while all subsequent elements are sampled with probability $p = \frac{k}{i}$ where $i$ stands for their order in the stream. If an item is chosen to be sampled while the reservoir is full, it replaces one element in the reservoir uniformly at random. Figure 3.2 illustrates an example of keeping a reservoir of size 3 on a stream of $n$ items.
Algorithm 2: Reservoir Sampling Algorithm

```plaintext
foreach arriving item \( a_i \) do
   if buffer has reached maximum capacity then
      \( p \leftarrow N/i \)
      heads \( \leftarrow \) flip a coin with heads probability \( p \)
      if heads then
         \( j \leftarrow \) random index in the buffer
         buffer \( [j] \leftarrow a_i \)
      else
         append \( a_i \) to the buffer
```

3.2.3 Biased Reservoir Sampler

The Biased Reservoir Sampler, as also explained in 2.5.2, is a variation of the traditional Reservoir Sampling scheme, this time fitting stream evolution concepts. The idea of this sampling technique is prioritizing new arriving elements, so that the reservoir containing the sample is biased towards recent events. In this implementation, a new arriving element is always added in the sample and the probability of the element replacing an already sampled item, or being appended in the reservoir, depends on the proportion of occupied slots in the memory. Thus, since new elements are always added in the sample, the proportion of more recent items, at each moment, in the sample is clearly higher than this of a traditional reservoir sampler. Algorithm 3 reflects the implementation of the Biased Reservoir Sampler used in this thesis while figure 3.2.3 depicts an example of the algorithm’s behavior.

Algorithm 3: Biased Reservoir Sampling Algorithm

```plaintext
foreach arriving item \( a_i \) do
   \( p \leftarrow m/N \) where \( m \) is the actual size of the buffer
   heads \( \leftarrow \) flip a coin with heads probability \( p \)
   if heads then
      \( j \leftarrow \) random index in the buffer
      buffer \( [j] \leftarrow a_i \)
   else
      append \( a_i \) to the buffer
```

3.2.4 Windowed Sampling

In the sliding window problem, we consider active items all recent items contained in windows defined by timestamps (time-based windows) or item ranges (sequence-based windows). The challenge of windowed sampling resides in the fact that sampled items may expire and must be replaced by other, active datastream elements. Babcock [8] suggests two approaches of sampling from time-based and sequence-based windows.

---

21
Similarly to the reservoir sampler, this biased variation also uses a coin toss, this time following the rule $p = \frac{m}{N}$, where $m$ is the number of items included in the buffer, to determine if the item will be appended or replace an existing item in the buffer. In that case, $a_1$ arrives with replacing probability equals to 0 thus it is appended in the sample. In a similar fashion, $a_2$ is appended in the buffer after a coin toss failure (tails) with success probability $p = \frac{1}{3}$. $a_3$ manages a heads outcome with probability $p = \frac{2}{3}$, thus it replaces an existing item in the buffer etc.

### Chain Sampler

The trivial case for the chain-sample algorithm maintains a sample of size 1 in a sequence-based window. The idea is that for every sampled item (e.g. $a_i$) one must also choose the index of the element that will replace it when it expires i.e. pick the index of the replacing element uniformly at random from the range $i + 1 \cdots i + n$. It should, then, store the picked element when it arrives and likewise pick the next index which will replace the second element. A chain of elements is, thereby, created dealing with the problem of replacement samples after their expiration. In order to create a sample of size $k$ with replacement we must store $k$ chains. Algorithm 4 and figure 3.4 describe the implementation of maintaining a chained sample of $N$ active

<table>
<thead>
<tr>
<th>$a_i$</th>
<th>probability</th>
<th>coin flip</th>
<th>buffer (max size = 3)</th>
</tr>
</thead>
<tbody>
<tr>
<td>$a_1$</td>
<td>0</td>
<td>tails</td>
<td>$a_1$ - -</td>
</tr>
<tr>
<td>$a_2$</td>
<td>$\frac{1}{3}$</td>
<td>tails</td>
<td>$a_1$ $a_2$ -</td>
</tr>
<tr>
<td>$a_3$</td>
<td>$\frac{1}{3}$</td>
<td>heads</td>
<td>$a_1$ $a_2$ -</td>
</tr>
<tr>
<td>$a_4$</td>
<td>$\frac{1}{3}$</td>
<td>heads</td>
<td>$a_4$ $a_3$ -</td>
</tr>
<tr>
<td>$a_5$</td>
<td>$\frac{1}{3}$</td>
<td>tails</td>
<td>$a_4$ $a_3$ $a_5$</td>
</tr>
<tr>
<td>$a_6$</td>
<td>1</td>
<td>heads</td>
<td>$a_6$ $a_3$ $a_5$</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Figure 3.3. Biased Reservoir Sampling Example** of keeping a sample of 3 items. Similarly to the reservoir sampler, this biased variation also uses a coin toss, this time following the rule $p = \frac{m}{N}$, where $m$ is the number of items included in the buffer, to determine if the item will be appended or replace an existing item in the buffer. In that case, $a_1$ arrives with replacing probability equals to 0 thus it is appended in the sample. In a similar fashion, $a_2$ is appended in the buffer after a coin toss failure (tails) with success probability $p = \frac{1}{3}$. $a_3$ manages a heads outcome with probability $p = \frac{2}{3}$, thus it replaces an existing item in the buffer etc.
elements in a sequence-based window.

**Algorithm 4: Chain Sampling Algorithm**

```plaintext
foreach arriving item \( a_i \) do
    if \( i \) has been selected as a replacement index then
        store(\( a_i \))
        update() //remove expired items and replace them with chained ones
    if buffer has reached maximum capacity then
        \( p \leftarrow n/N \) where \( n \) is the actual size of the buffer
        heads \( \leftarrow \) flip a coin with heads probability \( p \)
        if heads then
            \( j \leftarrow \) random index in the buffer
            buffer \([j]\) \( \leftarrow a_i \)
            \( k \leftarrow \) select future replacement index for \( a_i \) in \([i+1, i+w]\)
            add \( k \) to the chain for \( a_i \)
        else
            append \( a_i \) to the buffer
            \( k \leftarrow \) select future replacement index for \( a_i \) in \([i+1, i+w]\)
            add \( k \) to the chain for \( a_i \)
```

**Priority Sampler**

For maintaining a sample of size 1 in a time-based window the priority-sample algorithm assigns a random number \( p_i \in [0, 1] \) defined as priority to each item and samples the item with the higher priority included in the window. For a sample of size \( k \) with replacement the algorithm assigns \( k \) priorities for each item choosing, for the \( i \)th position in the sample, the active item with the higher \( i \)th priority.

**Algorithm 5: Priority Sampling Algorithm**

```plaintext
initialize: assign all priorities of buffer to \(-1\)
foreach arriving item \( a_t \) do
    update() //remove expired items and replace them with chained ones
    priorities[\( N \)] = createPriorities() //vector of \( N \) random priorities in \([0,1]\)
    assign priorities to \( a_t \)
    foreach \( j \) in buffer do
        chain \( a_t \) in position \( j \) such that the chain preserves an older to newer order in time and higher to lower priority order
```

### 3.2.5 The Greedy Sampler

In this policy the idea of Concept Drift adaptation is introduced in a greedy fashion, hence the name of the *Greedy* Sampler. We used the concept drift detection
Table 3.4. Chain Sampling Example of keeping a sample with replacement of 3 items in a window of 3. $a_1$ is added to the buffer, since it is not yet full, with probability $p = 1$. A random index (3) is then selected from the window which will contain all active items when $a_1$ expires ([2, 4]). Item $a_2$ is also sampled and a random index (5) is similarly selected for its future replacement. When $a_2$ arrives it is stored in memory in order to replace $a_1$ when it expires. Item $a_3$ is also sampled and a future replacement is randomly selected. At the next item arrival, $a_4$, item $a_1$ is not longer active thus it is replaced with its chained item, $a_3$. $a_4$ is also stored in memory since it was selected as a future replacement by item $a_3$ and then sampled with probability $p = 3/4$ replacing randomly an item in the buffer discarding all of its chain. Again, a future replacement index is selected from the window which will be active upon its expiration.
Figure 3.5. *Priority Sampling Example* of keeping a sample of 3 elements in a window of 1000ms. Priority sampling also uses the chain logic for future replacements. Before any item arrives, the positions of the buffer are initialized with the value $-1.0$ (or any value below 0). When $a_0$ arrives a vector with random priorities is assigned to the arriving element and, in that case, all 3 tuples consisting of the item and its respective priority are added in the buffer. After 300ms $a_{300}$ arrives and is also being assigned a vector of three priorities $(0.6, 0.1, 0.3)$. The first and third position of the buffer are replaced by the respective tuples since their priority is higher but for the second position the tuple $\langle a_{300}, 0.1 \rangle$ is chained below $\langle a_0, 0.8 \rangle$ in the buffer. Similarly all tuples for $a_{600}$ are placed in the chain with descending priority and increasing timestamp (older to newer). Upon expiration of an item its position in the buffer is replaced with the next one in chain.
algorithm introduced in [2.3] in order to detect positive variations of the mean of the distribution. The Greedy Sampler adopts the behavior of the Reservoir Sampler, that is keeping a uniform sample of all streamed items, also utilizing a drift detection component which triggers the sampler in the presence of drift. In case of a change detection in the distribution the sampler evicts a proportion of its internal buffer and restarts the sampling procedure.

\textbf{Algorithm 6:} Greedy Sampling Algorithm

\textbf{parameters:} $e$ the eviction rate, $CDD$ the concept drift detector

\textbf{foreach} arriving item $a_i$ \textbf{do}

\hspace{1em} add $a_i$ to $CDD$

\hspace{1em} \textbf{if} drift is detected \textbf{then}

\hspace{2em} evict $e$ proportion of the buffer uniformly at random

\hspace{1em} \textbf{else}

\hspace{2em} sample with the reservoir policy

3.3 Implementation

For the implementation of the testbed, a number of classes and interfaces were developed, applying to the structure portrayed in figure 3.1. An object implementing the source interface, namely the \texttt{SourceFunction}, produces the source stream, which, in the case where it derives from a random variable, it is followed by a \texttt{NumberGenerator} implementation. The diagrams in figures 3.6 and 3.7 represent the implementations of both the aforementioned interfaces respectively.

3.3.1 Sampling as a stream operator

As analyzed in section 2.4.2 operators are mainly transformations on the incoming streams and sampling can be perceived as such a transformation. Consequently, in order to reflect our optimization goals, all samplers were implemented as operators in the Apache Flink system. Generally, the sampler operator consists of three main units: a memory component (the buffer) which stores the sample of the stream, a sampling component which updates the buffer and an emission component which produces the output of the operator. A generic view of such an architecture is presented in figure 3.8.

In runtime, each operator is initialized with its sampling policy and an empty buffer. Upon arrival of each item the operator chooses whether or not to add or replace the item in their internal buffer. With a constant, user defined rate, the operator chooses at random elements from the buffer and emits them to the next component of the pipeline. Any sampling policy should implement the \texttt{SampleFunction} interface which is given as an input parameter for the \texttt{StreamSampler} operator.
The inner buffer of each operator is modeled by the Buffer class (figure 3.9) which contains related, trivial operations, such as generating random outputs, monitoring capacity constraints, random replacement of items etc.

The StreamSampler operator is responsible for reading the sampling policy (i.e. the sampling function) and performs two main actions:

- **update** the buffer state: Each parallel instance of a sampler policy should, at each item arrival, update its internal buffer (i.e. the sample) according to the sampling policy.

- **emit** items: A thread, decoupled from items’ arrival, is responsible for emitting random items drawn from the sample with replacement at an emission rate defined by the sampling policy.

The SampleFunction interface, from the other hand, is responsible for providing all information and logic needed for the operator. In particular, each sampling class acting as a sampler should implement methods which:
Figure 3.7. The NumberGenerator interface and its implementations.

Figure 3.8. The StreamSampler operator as implemented in Apache Flink.
Figure 3.9. The Buffer class.

Figure 3.10. The DistanceEvaluator class.

Figure 3.11. Class diagram of the StreamSampler operator.
• return the **contents** of the buffer

• produce a **random output** from the buffer

• return the **emission rate** that is the number of items per second which are generated from the buffer for the output stream

• contain the **sample** function which includes the core sample creation logic of the stream sampler

• **reset** the sampler

The datastream, after being sampled, may be transformed through several operators according to each task. In a distance evaluation task, for example, the output stream is connected with the source and evaluated with a **DistanceEvaluator** object (fig 3.10).
Figure 3.12. Implementations of the `SampleFunction` interface, defining each sampling policy.
Chapter 4

Experimental Driven Analysis

For our analysis, each sampling policy is tested for their ability to handle a collection of use cases under various constraints. The experimental analysis is aiming at assessing the behavior of each policy to a set of five different tasks, demanding dissimilar handling. We, therefore, assess the performance of each sampler to each scenario, building, ultimately, an evaluation table summarizing all results.

4.1 Evaluation Synopsis

Before analyzing the datasets as well as the tasks performed for the evaluation section, we present two tables summarizing the experimental analysis also acting as reference for abbreviations used in this section. Table 4.1 serves as a dictionary for all the sampler abbreviations while table 4.2 provides an overview of the dataset used for each experiment along with the samplers tested for the task.

<table>
<thead>
<tr>
<th>Full Name</th>
<th>Abbreviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reservoir Sampler</td>
<td>RS</td>
</tr>
<tr>
<td>Biased Reservoir Sampler</td>
<td>BS</td>
</tr>
<tr>
<td>Chain Sampler</td>
<td>CS</td>
</tr>
<tr>
<td>Priority Sampler</td>
<td>PS</td>
</tr>
<tr>
<td>FiFo Sampler</td>
<td>FS</td>
</tr>
<tr>
<td>Greedy Sampler</td>
<td>GS</td>
</tr>
</tbody>
</table>

Table 4.1. Sampling algorithm abbreviations

4.2 The Datasets

For the five groups of experiments we used three different datasets fitting to each use case.
<table>
<thead>
<tr>
<th>Experiment</th>
<th>Dataset / Source</th>
<th>Tested Samplers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitivity</td>
<td>Normal distribution with</td>
<td>RS, BS, CS, FS, FS</td>
</tr>
<tr>
<td></td>
<td>outliers</td>
<td></td>
</tr>
<tr>
<td>Concept Drift Adaptation</td>
<td>Normal distribution with</td>
<td>RS, BS, CS, FS, GS</td>
</tr>
<tr>
<td></td>
<td>outliers and one sudden</td>
<td></td>
</tr>
<tr>
<td></td>
<td>drift</td>
<td></td>
</tr>
<tr>
<td>Distribution Approximation</td>
<td>Airline Dataset</td>
<td>RS, BS, CS, PS, FS</td>
</tr>
<tr>
<td>Heavy Hitters</td>
<td>Airline Dataset</td>
<td>RS, BS, CS, PS, FS</td>
</tr>
<tr>
<td>Classification</td>
<td>RBF Dataset</td>
<td>RS, BS, CS, PS, FS</td>
</tr>
</tbody>
</table>

Table 4.2. Experiment synopsis

**Gaussian Distribution**

The first dataset is a synthetic Gaussian source which is generating 500K samples drawn from a Gaussian distribution. The distribution can be stable over time, with smooth or sudden drift with a 10% chance of outliers. Outliers are generated uniformly at random outside the interval:

\[
[Q_1 - 1.5 \cdot IQR, Q_3 + 1.5 \cdot IQR]
\]  \hspace{1cm} (4.1)

defined by the interquartile range of the distribution \(IQR\) \[^{37}\]. More specifically, the interquartile range in a normal distribution is defined as \[^{35}\]:

\[
IQR = Q_3 - Q_1
\]  \hspace{1cm} (4.2)

where the values of \(Q_1\) and \(Q_3\) can be calculated from the inverse cumulative distribution at the points 0.25 and 0.75 respectively.

- In the case of a **stable** distribution for each data point \(a_i\): \(a \sim \mathcal{N}(0,1)\)
- A Gaussian source simulating **sudden drift** evolves from \(\mathcal{N}(0,1)\) to \(\mathcal{N}(2,1)\) in one step.

**The Airlines Dataset**

The *airline dataset* \[^1\] consists of flight entries within the USA containing information about departure and arrival times, origin, destination, carrier etc of each flight. While the complete dataset consists of 120M entries we chose to stream a subset of 50.000 data points containing all flights in 2008. A representative entry of this dataset is listed below:

\[2008,1,1,2,10,737,\text{NW},336,274,\text{LAX},\text{DTW},1979,0,-3\]

where the labels of each field are, respectively:
The Random RBF Dataset

The randomRBF dataset is a synthetic dataset proposed in [10] which generates a hypersphere of normally distributed data points. In our case, 10M of 10-dimensional points were generated for the incremental classification task.

4.3 Experiments

In this section we analyze the tasks performed for the evaluation purposes (namely sensitivity, concept drift adaptation, distribution approximation, heavy hitters and classification), along with a comparison of the performance of each algorithm against each task. The results obtained from this experimental analysis will lead to the assessment of the evaluation table discussed in 4.4 and 4.5.

4.3.1 Sensitivity

The sensitivity task measures mainly the reaction of the buffer of each sampler to noisy data streams. For this task, a data stream drawn from a stable, normal distribution was generated, with an injected probability of generating outliers.

Parametrization:

- The input stream consists of 500000 data points generated from an one-dimensional normal distribution: \( \mathcal{N}(0,1) \) adding also a 10% probability of generating an outlier for each generated data point.

- In this initial experiment we did not evaluate the output data points rather than the buffer distribution after processing each incoming element. Thus there is no need of setting up the emission rate parameter. However, the buffer size was set to 5K.

- The evaluation metric used is the Bhattacharyya distance as described in [9] although in figures 4.1 - 4.5 no such distance is depicted, rather than the mean and standard deviation of the measured buffer distribution against the true mean and standard deviation of the source. The Bhattacharyya distance between two normal distributions follows the subsequent formula:

\[
D_B(p, q) = \frac{1}{4} \ln \left( \frac{1}{4} \left( \frac{\sigma_p^2}{\sigma_q^2} \sigma_q^2 + \sigma_q^2 + \sigma_q^2 \right) \right) + \frac{1}{4} \left( \frac{(\mu_p - \mu_q)^2}{\sigma_p^2 + \sigma_q^2} \right) \tag{4.3}
\]

where:
Figure 4.1. Reservoir Sampler sensitivity to outliers.

- $p$ and $q$ are the two measured normal distributions,
- $\mu_p$, $\mu_q$ their mean parameters, and
- $\sigma_p$, $\sigma_q$ their standard deviation parameters respectively.

Figures 4.1 - 4.5 represent the mean and standard deviation of the inner buffer against the source parameters for five sampling policies, namely the Reservoir Sampler, Biased ReservoirSampler, Chain Sampler, Priority Sampler and FiFo Sampler, in the presence of outliers. While there is not significant difference among the majority of the samplers, the Reservoir Sampler seems significantly more stable than the rest, as expected, while the Chain Sampler produces the second most stable buffer distribution. The main assumption one can derive from the results of this experiment is that, the more biased a sampler is towards recent events the more sensitive it is against the presence of outliers and noise.

4.3.2 Concept Drift Adaptation

In this evaluation task the adaptation of the buffer distribution to concept drift was tested against all policies and the buffer size. We should note here that although the Priority Sampler was also tested, the obtained average results were unstable and noisy which is why they don’t appear in our plots.

Parametrization:

- The input stream consists again of 500000 data points generated from an one-dimensional normal distribution with 10% presence of outliers, this time with a sudden drift from $\mathcal{N}(0, 1)$ to $\mathcal{N}(2, 1)$ half across the streaming process.
Figure 4.2. Biased Reservoir Sampler sensitivity to outliers.

Figure 4.3. Chain Sampler sensitivity to outliers.
Figure 4.4. Priority Sampler sensitivity to outliers.

Figure 4.5. FiFo Sampler sensitivity to outliers.
The inner buffer distribution was, again, evaluated, while the buffer size varied among 1K, 5K, 10K and 50K capacities.

The evaluation metric is the same used in 4.3.1 (i.e. the Bhattacharyya distance), this time also depicted in the respective figures 4.6 - 4.9. Figures 4.6 - 4.9 depict this distance from the source distribution in presence of concept drift for four different buffer sizes: 1K, 5K, 10K and 50K respectively. The Reservoir Sampling policy (RS) has the worst performance among all samplers as expected due to its uniformity feature. Yet, for small buffer sizes its biased variant (BS) adapts well to the generated drift unlike larger buffer sizes which make this sampler loose its adaptivity feature. The FiFo Sampler (FS) also performs well under this scenario as does the Greedy Sampler (GS) which is able to detect the drift and adjust its buffer accordingly. The windowed sampler, CS, has a stable, invariant to sampling size performance which is second worst among all samplers.

4.3.3 Distribution Approximation

The goal of this experiment is to evaluate the ability of the output datastream to count aggregates in one dimension and approximate this discrete distribution.
Parametrization:

- The input stream is the first 100000 data points from the airlines dataset 4.2.
- The buffer size was again varied among 1K, 5K, 10K and 50K capacities while this time the emission rate was set to 200 data points/s.
- The task is an aggregate counting task and will be described in detail in the following paragraph.
- Finally, the evaluation metric used is the accuracy of the output stream when approximating the distribution in one dimension.

The Task: The aggregates for one dimension (namely the Carrier field) were counted, normalized and averaged over 4 independent runs for each sampler. Then the absolute mean distance from the true distribution was calculated and used to compute the accuracy of each method. More specifically:

1. For each sampling policy a set of vectors was calculated

   \[ x_j = [n_1, n_2, \cdots, n_N], j \in \{1, 2, 3, 4\} \]  \hspace{1cm} (4.4)

   where \( n_i = \frac{c_i}{\sum c_i} \) is the normalized value of each counter \( c_i \).
2. A similar vector $y$ contains the respective values for the true distribution.

3. The error vector of each sampler against the true distribution is

$$e_j = \sum_i |y_i - x_{ij}|, j \in \{1, 2, 3, 4\}$$  \hspace{1cm} (4.5)

4. While its accuracy is calculated as $a = I - e$

The mean accuracy for each experiment is depicted in figure 4.10 grouped by buffer size. Again, the reservoir sampler performs worst than its competitors for all buffer sizes. In small buffer sizes, however, its biased variant outperforms the rest although there is high probability, judging by its overall performance in this experiment, that this measure is a statistical error. In general, the FiFo sampler appears to be the most accurate although in most of the cases its confidence bounds overlap with both Chain and Priority samplers.

### 4.3.4 Heavy Hitters

The heavy hitters task evaluates how well each sampling policy performs in a top-k task, that is retrieving the most popular classes in one dimension.
Figure 4.9. Distance from true distribution in conditions of concept drift, with buffer size = 50000 data points.

**Parametrization:** This experiment was conducted with the same parameters as in 4.3.3.

**The Task:** The field examined was, again, the Carrier field from the airline dataset and precision ($P$) was used as an evaluation metric. Following Manning’s definition of precision [27]:

$$
\text{Precision} = \frac{\#(\text{relevant items retrieved})}{\#(\text{items retrieved})} = \frac{tp}{tp + fp} \quad (4.6)
$$

Figures 4.11 represent the average precision at top 1, 2, 3, 4 and 5 (out of 20 different carriers in the dataset) for 3 different buffer sizes: 500, 1K and 5K. In this task the FiFo and Biased Samplers show the best overall performance while the Reservoir Sampler outperforms its competitors when the buffer size is large. In general, the top 3 Carriers are retrieved in almost every case, especially for large buffer sizes, larger than 2% of the stream size.

**4.3.5 Classification**

Parametrization:
Figure 4.10. Accuracy of output stream when counting aggregates in one dimension.

- The input dataset is the first 2000000 data points from the synthetic RBF dataset discussed in [12].
- The buffer size was again this time set to 10K and the emission rate was set to 600 data points/s.
- The task is an incremental Classification task, using a parallelized version of the Hoeffding Tree classification algorithm [17], along with its error calculation component.
- The error obtained from the aforementioned component, measured against the stream progression, serves as our evaluation metric.
Figure 4.11. Precision of each sampler for the top-k task.

Figure 4.12(a) depicts the classification error of the algorithm for the entire data stream progress; In figure 4.12(a) one can observe that both the Reservoir, and Biased Reservoir Samplers produce a stream that can make the classification error converge to the optimal error. However, the rest of the samplers do not perform considerably worse. The most important deduction from this benchmark is that the Reservoir Sampling algorithm not only has the minimum error and the fastest convergence over the whole progress but also its overall error is always smaller than when using the original, unsampled dataset. This is mainly because resampling is influencing the weak learners of Hoeffding Tree classification algorithm [13].
Figure 4.12. Classification error against data point progress for all sampling policies.
4.4 Evaluation

The evaluation part consists of assessing an evaluation score for each sampler and task and construct a score-table with all assigned scores. The score assessment is achieved taking advantage of the evaluation metric used for all experiments. In particular:

1. For the outlier insensibility task, we computed the average distance of the inner buffer distribution from the true distribution, for each sampler, along the whole streaming progress.

2. For the concept drift detection scenario, since the drift happens at the 250000th data point, we examine the condition of each sampler’s inner distribution a few data points after the drift. Thus, we averaged, for each sampler, the measured distance at the 270000th data point against all tested buffer capacities. Figure 4.13 demonstrates the exact location of the data point along the streaming progress.

3. Aggregates score was assessed using the average accuracy of each sampler against the various buffer sizes.

4. The performance of each sampler on the heavy hitters task was measured with the average precision at 5 against all buffer sizes.

5. Finally, for the classification task, since all samplers have similar behavior, the evaluation metric used was the classification accuracy $(1 - \text{error})$ at the first quarter of the datastream progress, that is on the 500000th data point.

Table 4.3 summarizes the scores given for each sampler and task, emphasizing the optimal policy for each task. The used evaluation metrics, although not optimal in assigning the exact quality of each sampler, they provide a sensible comparison measure for the policies implemented against each task.

<table>
<thead>
<tr>
<th>Experiment</th>
<th>RS</th>
<th>BS</th>
<th>CS</th>
<th>PS</th>
<th>GS</th>
<th>FS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sensitivity</td>
<td>0.0261</td>
<td>0.0262</td>
<td>0.0262</td>
<td>0.0264</td>
<td>-</td>
<td>0.0267</td>
</tr>
<tr>
<td>Concept Drift Adaptation</td>
<td>0.3067</td>
<td>0.0726</td>
<td>0.2285</td>
<td>NA</td>
<td>0.0340</td>
<td>0.0596</td>
</tr>
<tr>
<td>Distribution Approximation</td>
<td>0.9887</td>
<td>0.9931</td>
<td>0.9906</td>
<td>0.9908</td>
<td>-</td>
<td>0.9910</td>
</tr>
<tr>
<td>Heavy Hitters</td>
<td>0.8667</td>
<td>0.8500</td>
<td>0.8700</td>
<td>0.8833</td>
<td>-</td>
<td>0.9000</td>
</tr>
<tr>
<td>Classification</td>
<td>0.6897</td>
<td>0.6715</td>
<td>0.6637</td>
<td>0.6598</td>
<td>0.6741</td>
<td></td>
</tr>
</tbody>
</table>

Table 4.3. Comparison of Sampling Algorithms
4.5 Discussion

Overall, there is no optimal policy for all the tasks. In each experiment each algorithm behaves differently due to a variety of reasons.

Consider, for example the first experimental task (§4.3.1). In that case, the Reservoir Sampling algorithm is the most stable and insensitive to outliers, since it keeps a uniform sample of the datastream. The Chain Sampling algorithm is the second most stable, yet the window size parameter plays a definitive role in that behavior. An increase in the size of the window signifies a more uniform sample in the inner buffer of the operator. Therefore, the insensitivity of the sampled output to outliers is rather a matter of uniformity of the inner buffer which makes biased algorithms incompetent for the task.

On the other hand, in the presence of concept shift, the uniformity feature of the Reservoir Sampler deprives the policy from adapting to new conditions. On the contrary, samplers with biased settings perform better in such cases, even if their performance is subject to window size, bias coefficient, or their inner buffer size. In detail:

- The Biased Reservoir Sampler adds items to the buffer deterministically. However, what is uncertain is which item is evicted from the buffer upon a new item’s arrival. Thus, at some point after the drift, the buffer will have been filled with such a percentage of items drawn from the new distribution, that the drift will be reflected in the distribution parameters. The time, or number of data points, needed so that the buffer reaches this proportion re-
reflects the buffer size.

- The *Greedy Sampler* checks, upon each item arrival, if the inner distribution of the buffer changes more than a certain threshold. Again, the size of the buffer acts as an inertia parameter for the change to be reflected.

- The *FiFo Sampler* takes as a sample the most recent data points, thus the distance of the inner buffer against the true distribution is directly connected with the size of the buffer. In fact, the distance will be minimized after the arrival of $N$ data points, where $N$ is the size of the buffer.

In modern applications, where stream evolution is a common concept, a uniform sample would rarely provide an accurate approximation of the datastream. Yet, the combination of an evolution detection component with a uniform sampling scheme (as done in the Greedy Sampler operator) may provide an adequate solution for the problem.

When it comes to the distribution approximation task (§4.3.3), the variance of each dimension, as well as the selectivity of its items, plays a major role in the approximation accuracy. For example, imagine variables with high selectivity, such as an airport with low traffic in this specific example. There is a chance that this variable is not included in the output datastream, lowering the approximation accuracy. In the same task, almost all samplers perform with high accuracy except the Reservoir Sampler. This is mainly because of the underlying drift in the distribution of the data points. They now reflect an underlying process rather than being generated from a stable Gaussian distribution. However, the reason why the Reservoir Sampler is performing poorly in this experiment is one additional point: it keeps a uniform sample of the whole datastream, thus, along the whole stream progress there is an even probability of emitting an item from any point along the sampled datastream progress. With that fashion, the data points that arrive earlier in the stream have better chance of being emitted multiple times, making the output stream biased towards past events. The heavy hitters task (§4.3.4) is quite similar in the sense that an aggregation of the output stream is, again, queried, this time against the $k$ most popular labels. For the same reason, the optimal approach here is the FiFo Sampler, a highly biased towards recent events sampler.

The classification experiment shows highly accurate results for all of the samplers. Nevertheless, the dataset used may not be optimal for testing the different features of the samplers. The same dataset containing concept drift could be, for example, a better testing use case in order to observe different behaviors for each policy.

Finally, despite windowed samplers not behaving optimally on any of the conducted experiments, their performance is not significantly worse in most of the tasks. Especially when it comes to concept shift adaptation or noisy datastreams, the size of the window acts as a delay parameter on the perception of the drift, or a smoothing.
factor respectively. However, we should point out that their computational complexity prevents fast execution, emerging the demand of an improved variation, or a different windowed policy. Figure 4.14 depicts this delay and poor performance of both Chain and Priority samplers.

Figure 4.14. Time comparison for one execution of the Classification task
Chapter 5

Future Work

The scope for that thesis was to perform an experimental analysis of different sampling policies performing different tasks, evaluating their performance on each task. Still, the tested policies along with the tested tasks are solely a subset of the potential algorithms and experiments that can be performed. We strongly believe that the implementation of the experimental testbed is helping towards testing more policies such as the ones cited in 2.5.2. One can further test each policy with additional types of learning models, such as kNN classifiers, SVM models or clustering tasks etc. Even for the classification task performed in 4.3.5 one can use different datasets which result in dissimilar accuracies despite using the same classifier [12].

Considering the detection change component, the tuning of the parameters of the Page Hinkley Test plays a major role in detecting the change in the distribution without triggering false alarms. The parameters have been set so that the shift of the tested distribution can be detected. In real-world problems, however, such detectors are generally being used to detect concept drift in the result of a decision policy, e.g. a change in the classification error rather than the distribution itself [32]. What is more, in such scenarios the change is rarely sudden. Instead, it takes the form of an evolving drift which makes its detection fairly more challenging.

Finally, the present study is only a small step towards sampling optimization. The testing of more sampling methods at further tasks can enrich table 4.3 and constitute a step towards the next milestone: building a sampling optimization plan deriving from the aforementioned evaluation. In such a system, one can perform a static analysis defining the characteristics of the stream along with the current use case, as well as a runtime analysis detecting runtime concepts such as concept drift or outliers, and choose, ultimately, the optimal sampling method according to the characteristics of the stream or the running task. We strongly believe that such a sampling optimization system with high accuracy and managed memory and computational constraints will facilitate datastream manipulation and lower model training costs.
Bibliography


