Streaming Predictive Analytics on Apache Flink

DEGREE PROJECT IN DISTRIBUTED SYSTEMS AND SERVICES AT KTH INFORMATION AND COMMUNICATION TECHNOLOGY

FOTEINI BELIGIANNI
Streaming Predictive Analytics on Apache Flink

Examiner: Vladimir Vlassov
Author: Foteini Beligianni
Supervisors: Seif Haridi Paris Carbone

A thesis submitted for the degree of Master of Science in Distributed Systems and Services

Stockholm, 15.07.2015
Acknowledgments

First and foremost, I would like to express my sincere gratitude to Paris Carbone, my academic supervisor, who provided me with feedback and support throughout this master thesis project.

Next, big thanks to Martha, Vasia, Gyula, Daniel and Atier for all the valuable comments and discussions throughout this last months, but also for all the nice moments in the office throughout this journey.

Last, another big thanks to Nikolas, a true friend that supported me more than I could ever ask to, during this two year EIT Digital Master program.
To my parents Yiorgo and Katerina, for everything they have done for me and to my uncle Yianni and aunt Mata, for their unconditional support.
Abstract

Data analysis and predictive analytics today are driven by large scale distributed deployments of complex pipelines, guiding data cleaning, model training and evaluation. A wide range of systems and tools provide the basic abstractions for building such complex pipelines for offline data processing, however, there is an increasing demand for providing support for incremental models over unbounded streaming data. In this work, we focus on the problem of modelling such a pipeline framework and providing algorithms that build on top of basic abstractions, fundamental to stream processing. We design a streaming machine learning pipeline as a series of stages such as model building, concept drift detection and continuous evaluation. We build our prototype on Apache Flink, a distributed data processing system with streaming capabilities along with a state-of-the-art implementation of a variation of Vertical Hoeffding Tree (VHT), a distributed decision tree classification algorithm as a proof of concept.

Furthermore, we compare our version of VHT with the current state-of-the-art implementations on distributed data processing systems in terms of performance and accuracy. Our experimental results on real-world data sets show significant performance benefits of our pipeline while maintaining low classification error. We believe that this pipeline framework can offer a good baseline for a full-fledged implementation of various streaming algorithms that can work in parallel.
Sammanfattning

Dataanalys och predictive analytics drivs idag av storskaliga distribuerade distributioner av komplexa pipelines, guiding data cleaning, model training och utvärdering. Ett brett utbud av system och verktyg ger endast grundläggande abstractions (struktur) för att bygga sådana komplexa pipelines för databehandling i off-line läge, men det finns en ökande efterfrågan att tillhandahålla stöd för stegvis modell över unbounded streaming data. I detta arbete fokuserar vi på problemet med modellering som ramverket för pipeline och ger algoritmer som bygger på grundläggande abstraktioner för stream processing. Vi konstruerar en streaming maskininlärnings pipeline som innehåller steg som model building, concept drift detection och kontinuerlig utvärdering. Vi bygger vår prototyp på Apache Flink, ett distribuerat databehandlingssystem med strömnings kapacitet tillsammans med den bästa tillgängliga implementation av en Vertical Hoeffding Tree (VHT) variant och ett distribuerat beslutsträd algoritm som koncepttest.

Dessutom jämför vi vår version av VHT med den senaste tekniken inom distributered data processing systems i termer av prestanda och precision. Vårt experimentella resultaten visar betydande fördelarna med vår pipeline och samtidigt bibehållen låg klassificerat felet. Vi anser att detta ramverk kan erbjuda en bra utgångspunkt vid genomförandet av olika streaming algoritmer som kan arbeta parallellt.
## Contents

List of Figures .......................... xi
List of Listings .......................... xii
List of Algorithms ........................ xii
Acronyms ................................ xi

1 Introduction ............................ 1
   1.1 Problem .......................... 2
   1.2 Purpose .......................... 2
   1.3 Contributions ..................... 2
   1.4 Methodology ....................... 3
   1.5 Delimitations ..................... 4
   1.6 Outline .......................... 4

2 Theoretical Background ................. 6
   2.1 Parallel Data Processing ............ 6
      2.1.1 Data Parallel Processing Models .. 6
   2.2 Data Streams ....................... 7
      2.2.1 Data Stream Definitions ........... 7
   2.3 Machine Learning Algorithms ......... 8
      2.3.1 Machine Learning Definitions ..... 9
      2.3.2 Decision Trees ................... 9
      2.3.3 Streaming Machine Learning ....... 12
   2.4 Apache Flink ....................... 16
   2.5 Related Work ....................... 18

3 Incremental Machine Learning Pipelines on Apache Flink .... 20
   3.1 Modeling Machine Learning Pipelines on Apache Flink Streaming .... 20
      3.1.1 Document Classification Machine Learning Pipeline Example .... 23
   3.2 Machine Learning Pipeline Implementation .................. 24
      3.2.1 Transformer ..................... 25
      3.2.2 Learner ......................... 25
      3.2.3 Evaluator ...................... 25
      3.2.4 ChangeDetector .................. 27

4 Algorithm Implementation ............... 29
   4.1 Hoeffding Tree on Apache Flink .......... 29
      4.1.1 Centralized Model of VHT Algorithm ............... 29
Appendix

A  Data Sets and Data Generators  A.1
A.1  Real Data Sets  A.1
A.1.1  Forest Covertype Data Set  A.1
A.2  Synthetically Generated Data Sets  A.1
A.2.1  HIGGS Data Set  A.1
A.2.2  Waveform-21 Data Set  A.2
A.2.3  LED Display Domain Data Set  A.2

B  UML Class Diagrams  A.3
B.1  Package Diagram  A.3
B.2  Class Diagrams  A.5
B.2.1  Classification Package Class Diagram  A.5
B.2.2  Metrics Class Diagram  A.6
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Decision Tree Induction Process</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>Types of Data Parallelism</td>
<td>12</td>
</tr>
<tr>
<td>4</td>
<td>Hoeffding Tree algorithm pipelined data flow</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Vertical Hoeffding Tree data flow in Apache SAMOA system</td>
<td>15</td>
</tr>
<tr>
<td>6</td>
<td>Apache Flink component stack[1]</td>
<td>17</td>
</tr>
<tr>
<td>7</td>
<td>Transformer abstraction for Machine Learning pipelines</td>
<td>21</td>
</tr>
<tr>
<td>8</td>
<td>Learner abstraction for Machine Learning pipelines</td>
<td>21</td>
</tr>
<tr>
<td>9</td>
<td>Evaluator abstraction for Machine Learning pipelines</td>
<td>22</td>
</tr>
<tr>
<td>10</td>
<td>Change Detector abstraction for Machine Learning pipelines</td>
<td>22</td>
</tr>
<tr>
<td>11</td>
<td>Machine Learning Pipeline</td>
<td>23</td>
</tr>
<tr>
<td>12</td>
<td>Document classification Machine Learning (ML) Pipeline</td>
<td>24</td>
</tr>
<tr>
<td>13</td>
<td>Package common class diagram, containing Learner and Transformer traits</td>
<td>24</td>
</tr>
<tr>
<td>14</td>
<td>Evaluator package class diagram in incremental-ml library on Apache Flink</td>
<td>27</td>
</tr>
<tr>
<td>15</td>
<td>ChangeDetector package class diagram</td>
<td>28</td>
</tr>
<tr>
<td>16</td>
<td>Data flow of native VHT implementation on Apache Flink</td>
<td>30</td>
</tr>
<tr>
<td>17</td>
<td>Data flow of Replicated model VHT implementation on Apache Flink</td>
<td>32</td>
</tr>
<tr>
<td>18</td>
<td>AttributeObserver package class diagram</td>
<td>35</td>
</tr>
<tr>
<td>19</td>
<td>Gaussian approximation for deciding on splitting a numerical attribute on sp5 splitting value</td>
<td>38</td>
</tr>
<tr>
<td>20</td>
<td>Apache SAMOA's high level architecture</td>
<td>40</td>
</tr>
<tr>
<td>21</td>
<td>Evaluation Pipeline of VHT algorithm variations in Apache Flink</td>
<td>43</td>
</tr>
<tr>
<td>22</td>
<td>Prequential classification error of Flink's native VHT SAMOA's VHT and RmVHT algorithm for UCI-Forest Covertype data set. Flink's native VHT has data source with parallelism equal to 1.</td>
<td>46</td>
</tr>
<tr>
<td>23</td>
<td>Average delay in seconds between node splits of VHT classifier, for UCI-Forest Covertype data set on Apache Flink and Apache SAMOA.</td>
<td>46</td>
</tr>
<tr>
<td>24</td>
<td>Average number of instances seen in each leaf before it splits on Apache Flink and Apache SAMOA.</td>
<td>47</td>
</tr>
<tr>
<td>25</td>
<td>Prequential classification error of Flink's native VHT, SAMOA's VHT and RmVHT algorithm for UCI-Forest Covertype data set. Flink's native VHT has data source with parallelism equal to 8.</td>
<td>48</td>
</tr>
<tr>
<td>26</td>
<td>Average number of instances seen in each leaf before it splits on Apache Flink VHT for different model parallelism.</td>
<td>49</td>
</tr>
<tr>
<td>27</td>
<td>Prequential classification error of Flink's native VHT, SAMOA's VHT and RmVHT algorithm for UCI-HIGGS data set.</td>
<td>49</td>
</tr>
<tr>
<td>28</td>
<td>Average delay in seconds between node splits of VHT classifier, for UCI-HIGGS data set on Apache Flink and Apache SAMOA.</td>
<td>50</td>
</tr>
</tbody>
</table>
29 Average number of instances seen in each leaf before it splits on Apache Flink and Apache SAMOA. .......................................................... 51
30 Average number of instances seen in each leaf before it splits on Apache Flink VHT for different model parallelism. ......................... 51
31 Classification error of Vertical Hoeffding Tree (VHT) and Replicated Model Vertical Hoeffding Tree (RmVHT) classifier, for Waveform 21-attribute data set on Apache Flink and Apache SAMOA. ........ 52
32 Classification error of VHT and RmVHT classifier, for Led data set on Apache Flink and Apache SAMOA. ........................................ 52
33 Package Diagram for incremental ML scala library. .................... A.3
34 Class diagram of classification package, containing VHT and DecisionTreeModel classes among others. ................................. A.5
35 Metrics package class diagram. .................................................... A.6

List of Listings

1 Scala Word Count example in Apache Flink Streaming API ........... 18
2 Transformer Base Trait ............................................................... 25
3 Learner Base Trait ................................................................. 25
4 Evaluator Base Trait .............................................................. 26
5 ChangeDetector Base Trait ....................................................... 27
6 AttributeObserver Base Trait .................................................... 35
7 Scala code example of split and select operators in Flink ............ 42
8 Scala code of the pipeline used for evaluating VHT ................. 43

List of Algorithms

1 Hoeffding Tree[2] ................................................................. 14
2 Prequential Evaluator Algorithm pseudo code ....................... 33
3 Page Hinkley Test algorithm [3] .............................................. 34
4 Identifying potential split points of a numerical attribute. ........ 39
5 Gaussian approximation for evaluating potential splitting points with respect to entropy. ...................................................... 39

List of Acronyms

API Application Programming Interface .................................... 2
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
<td>17</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
<td>42</td>
</tr>
<tr>
<td>DSPE</td>
<td>Data Stream Processing Engines</td>
<td>40</td>
</tr>
<tr>
<td>DT</td>
<td>Decision Tree</td>
<td>8</td>
</tr>
<tr>
<td>HDFS</td>
<td>Hadoop Distributed File System</td>
<td>7</td>
</tr>
<tr>
<td>HT</td>
<td>Hoeffding Tree</td>
<td>5</td>
</tr>
<tr>
<td>LS</td>
<td>Local Statistic</td>
<td>15</td>
</tr>
<tr>
<td>MA</td>
<td>Model Aggregator</td>
<td>15</td>
</tr>
<tr>
<td>ML</td>
<td>Machine Learning</td>
<td>xi</td>
</tr>
<tr>
<td>MPI</td>
<td>Message Passing Interface</td>
<td>6</td>
</tr>
<tr>
<td>PHT</td>
<td>Page Hinkley Test</td>
<td>33</td>
</tr>
<tr>
<td>PI</td>
<td>Processing Item</td>
<td>41</td>
</tr>
<tr>
<td>RDD</td>
<td>Resilient Distributed Dataset</td>
<td>18</td>
</tr>
<tr>
<td>RmVHT</td>
<td>Replicated Model Vertical Hoeffding Tree</td>
<td>xii</td>
</tr>
<tr>
<td>SAMOA</td>
<td>Scalable Advanced Massive Online Analysis</td>
<td>40</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
<td>A.3</td>
</tr>
<tr>
<td>VHT</td>
<td>Vertical Hoeffding Tree</td>
<td>xii</td>
</tr>
</tbody>
</table>
1 Introduction

Over the last years the size of the available data sets has increased enormously, due to the continuous growth of the Internet and social media (Facebook, Twitter, etc.), as well as the high rate of adoption of devices which are capable of creating information (e.g. smartphones). Facebook clusters for example handle 500 TB of new data every day[4] and Twitter has 500M tweets sent every day. Only in United Stated the digital universe is expected to grow by 25% each year till 2020, reaching by then the enormous amount of 6.6 zettabytes or around 7 billion terabytes[5].

The volume of those structured or unstructured data sets, referred with the "buzz" term Big Data, is so high that existing data mining methods are incapable of dealing with that. Data Mining is the process of extracting useful information from complex big data sets, with the use of machine learning techniques; like classification, pattern recognition, clustering to identify existing valuable and understandable patterns/ models in the data sets[6]. By learning a model that can be used to identify the patterns of a data set, predictive data mining techniques can be used to do predictions for previously unseen data.

Predictive data mining techniques are broadly used nowadays in recommendation services (e.g shopping recommendations, friends recommendation in social media, etc.), drift detection of the original concept and more. Nevertheless, the velocity and variety of big data sets arise a number of challenges, which are not taken into consideration by the traditional methods[7]. Some of the most important challenges is the need for on-line processing models which will incorporate newly generated training data to the predictive model with the possible latency increasing in the same moment the ingestion rate of the algorithms. Data mining methods assume that all data can be stored in memory, whereas there are space and time restrictions due to the high generating speed of data[7].

Incremental machine learning and streaming analytics (or "data stream mining") are becoming an emerging field for research and business intelligence applications today. The so-called incremental or on-line machine learning algorithms vary from traditional algorithms that regard data sets with fixed size for off-line processing, instead, focusing on continuously updating their underlying model as data arrives. Some of the benefits of incremental approaches are among others adaptability to data trends, bounded memory requirements, exploratory data analysis on fresh data, low-latency predictions for critical decision making and automated eviction of old data.

Nowadays, every human computer interaction produces information, thus creating a new need for analyzing data fast enough in order to add more value to an
1 INTRODUCTION

enterprise or product. Real-time or nearly real-time information makes a company much more agile than its competitors, therefore there exist an emerging industrial need for Real-Time Predictive Analytics. Instead of relying on batch processing for extracting information and pattern from the available data, resulting on delays between the creation and exploitation of the data, streaming large-scale data processing engines, can be used to incrementally update the extracted models and patterns.

1.1 Problem

Despite the great benefits of incremental machine learning, a number of big challenges arise as well; with the main of those to be the redesign of the existing learning algorithms in the streaming model. Adding to those challenges, a distributed execution of incremental machine learning algorithms arises the problem of distributing the algorithm in an efficient way so as the learning process scale-out without sacrificing the accuracy of the learned model.

Apache Flink is an open source platform for large-scale data processing[8]. This master thesis builds its contributions on Apache Flink, by porting in the abstractions used in Batch ML algorithms which could be also used in the Apache Flink Streaming Application Programming Interface (API) and extending the existing pipeline with additional abstractions that are needed in incremental machine learning.

1.2 Purpose

This master thesis presents the abstraction which were defined for modeling an ML pipeline with Apache Flink Streaming API. The pipeline will target in facilitating the implementation of ML algorithms on Apache Flink Streaming API.

1.3 Contributions

In scope of this master thesis the following contributions were achieved:

1. A literature study of existing systems and programming models which allow the implementation of incremental ML algorithms. The most important systems are presented in Section 2.5

2. Part of the implementation of an adapter for integrating Apache SAMOA streaming machine learning framework on Apache Flink Streaming
3. The design of the basic abstractions needed to model incremental ML pipelines on a descriptive manner

4. The implementation of our ML pipeline prototype along with an implementation of a scalable decision tree algorithm

5. The evaluation of the performance of the implemented prototypes and algorithms, against the scalable decision tree algorithm implemented on Apache SAMOA. The evaluation showed that our solution scaled well and maintains an acceptable accuracy over time

6. The implementation will be a contribution to the open source project Apache Flink[8], thus the implementation can be freely accessed

1.4 Methodology

The quantitative research method was used for this master thesis[9]; experiments and testing over different platforms of the incremental machine learning algorithms were conducted for the evaluation of the algorithms.

The philosophical assumption of this degree project was positivism[9], as the evaluation of the designed framework and the implemented algorithm was based on the experimental results and it was unbiased by the opinions of the researcher.

Moreover, the experimental research method was used in order to investigate how changes between the different versions of the algorithms affect the results.

The factors affecting the experimental results are:

- Implementation of the algorithms
- Datasets used: The size as well as the nature of the datasets that are used in incremental classification algorithms are highly important.
- Parallelism: Number of available slots for parallel processing of the data

To effectively compare the different versions of the algorithm a set of metrics was identified and used. Specifically, the variables considered for the evaluation of the experiments were:

- Prequential Error: The prequential classification error of the algorithm
- Execution time of the algorithms in different platforms and with respect to different data sets.

The validity of the measurements from our algorithms have been ensured by cross-checking the implementation of the algorithms. Moreover, the results were
1 INTRODUCTION

compared to the results of the same algorithms executed on Apache SAMOA, a different platform.

In order to avoid possible oscillation due to the random seed used by the data generators, the experiments were run multiple times for each data set, which revealed a low variance between different runs, thus confirming the reliability of the experiments.

The programming languages used in scope of this degree project as well as the testbed have been described in this report. Consequently, the replicability of our results can be ensured. Additionally, all datasets used in the experiments are publicly available in the web but also described in the appendix A, as well as the paper describing the implemented algorithm[2]. Taking all these into consideration we can safely assume that our results could be replicated by other research.

1.5 Delimitations

One of the issues that could affect the study of an incremental machine learning algorithm is the arrival rate of the observed data points. As stated in [2], when the arrival rate is higher than the maximum processing rate of an application, the quantity of unused data grows without bounds as time progresses. As a result more data points may be needed for higher accuracy.

Sampling the stream of incoming events could be one solution to high arrival rate, as a sampling technique could provide data points to the algorithm in a normal rate. The effect of the arrival rate of the input stream is not considered for the evaluation of the implemented algorithms.

1.6 Outline

Section 2 presents all necessary theory that will introduce the reader on the base concepts and theory needed to understand the systems used in this thesis. Moreover, related work on the area is presented.

Section 3 presents our model for ML pipelines on Apache Flink Streaming API. Moreover the prototyping of the pipeline model is presented in detail.

Section 4 presents all the algorithms which were implemented in scope of this master thesis, in order to evaluate the modeled pipeline.

In Section 4.1 the details of the VHT, the ML algorithm implemented with the pipeline modeled by this master thesis is presented. Moreover, one more variation
of the Hoeffding Tree (HT) algorithm is presented and the main differences with the first implementation are highlighted.

Section 6, presents the evaluation pipeline of the two implemented variations of HT algorithm, as well as the experimental setup for the evaluation. In addition, the experimental results of the evaluation are presented also in the same section.

Lastly, in Section 7, the conclusion is given and potential future work is discussed.
2 Theoretical Background

2.1 Parallel Data Processing

Nowadays, the size of the available data sets has increased enormously, hence there exist an essential need for processing massive volume of data in an efficient manner. Parallel data processing enables the processing of large data sets in parallel, thus increasing the throughput of the algorithms.

In literature there are encountered mainly three categories of parallelization[10, 11, 12]:

1. *Data Parallelism*: The same operation is performed in parallel on subsets of the data set. Data Parallel algorithms are parallelized with respect to the data

2. *Task Parallelism*: Different operations are performed in parallel on the same data set

3. *Hybrid Parallelism*: A pipeline of tasks executed concurrently

2.1.1 Data Parallel Processing Models

Message Passing Interface (MPI) [13], is one of the first widely used standards for parallel processing of data. MPI is a language-independent message passing interface standard which can be used for developing parallel applications that distribute the work load in numerous machines of a cluster. An MPI program is loaded and executed in every processor of the cluster facilitating the communication of the collaborating processes through functions like "MPI_Bcast", "MPI_Reduce", etc [14]. One of the drawbacks of the MPI standard is that it does not offer any fault tolerance mechanisms.

MapReduce is a programming model, first proposed in 2004 by Google[15], which among others offers a fault tolerance mechanism, thus allowing people with small experience in parallel computing to implement efficient applications. In MapReduce the user expresses a problem with the use of two functions[15]:

1. *Map*: Takes the input in the form of key/value pairs \((K,V)\), process them and produces intermediate results in the form of \((K,V)\) pairs.

2. *Reduce*: Receives all intermediate results that have the same key and merge those together in order to produce a final result per key.
There exist numerous frameworks which implement the MapReduce programming model, and all of them facilitate the use of distributed file systems like the Hadoop Distributed File System (HDFS)[16]. Except from being a broadly used programming model for big data processing, MapReduce forms the base idea of many big data batch and stream processing engines, such as Apache Hadoop, Apache Flink, Apache Spark, Apache Storm[8, 17, 18].

2.2 Data Streams

The space and time assumptions of the traditional data mining techniques makes them inappropriate for mining big data which cannot fit on the main memory with low processing latency and unbounded processing restrictions. Incremental machine learning algorithms are based on data streams, which have much different characteristics than the data sets.

2.2.1 Data Stream Definitions

**Data Stream.** A *Data Stream* is a continuous partitioned and partially ordered stream of data elements, which can be potentially infinite. Each data element of the stream follows a predefined schema[19].

**Data Element.** A *Data Element* is the smallest unit of data which can be processed by a data streaming application, as well as the smallest unit that can be sent over network. A *Data Element* can be any object of a given expected schema.

**Data Stream Source.** A source that produces a possibly infinite amount of data elements, which can not be loaded into the memory, thus making batch processing unfeasible [7].

**Data Stream Sink.** A *Data Stream Sink* is an operator representing the end of a stream. *Data Stream Sinks* have only an input data stream and no output data stream. In MapReduce programming model, an operator writing to a file or database can be considered as a Sink operator.
**Data Stream Operator.** A *Data Stream Operator* is a function which applies a user-defined transformation in the incoming stream of data elements and emits the transformed data to the outgoing streams [12].

### 2.3 Machine Learning Algorithms

Over the last years ML algorithms are used for a wide range of services and systems, such as recommendation systems, spam detection and more.

Machine learning algorithms use a data set as an input and try to *learn* a model in order to make data-driven predictions for other unseen data. Thus with the use of ML algorithms, computers can learn concepts/models, without being statically programmed.

There exist mainly two ways of categorizing an ML algorithm, depending either on the format of the input data set or on the format of the output of the learned model [20, 21].

With respect to the type of information that a machine learning algorithm needs in order to *learn* a model, ML algorithms can be categorized as follows:

- **Supervised:** For each observation of the input data set $x_1, x_2, \ldots x_n$ an associated response value $y_1, y_2, \ldots y_n$ is also provided to the algorithm. In that way, the algorithm *learns* a model, which tries to relate the response to the input observations. Examples of supervised ML algorithms are classification and regression algorithms.

- **Unsupervised:** For each observation of the input data set $x_1, x_2, \ldots x_n$ no associated response value is provided to the algorithm. In that way, the algorithm *learns* a model, which tries to find relations between observations. Clustering algorithms are an example of unsupervised ML algorithms.

Depending on the prediction output of a ML algorithm, there exist a number of categories the main of which are the following:

- **Classification:** Classification algorithms are supervised ML algorithms, that *learn* a model which predicts classes(labels) for unseen data points. Examples of classification algorithms are the Decision Tree (DT) and Logistic Regression.

- **Regression:** Regression algorithms are supervised ML algorithms, that *learn* a model which predicts a continuous value for unseen data points. Linear Regression and Regression DTs are examples of regression algorithms.
2 THEORETICAL BACKGROUND

- **Clustering**: Clustering algorithms are unsupervised ML algorithms, which identify clusters in the input data points and assign each data point to a cluster. kMeans and k-Medoids are examples of clustering algorithms.

### 2.3.1 Machine Learning Definitions

**Overfitting.** *Overfitting* refers to the phenomenon where the model learned with an ML algorithm follows closely the training data set, thus it is not able to do correct predictions for new unseen observations.

**Concept Drift.** *Concept Drift* refers to gradual changes that can occur in the underlying distribution of incoming data. In that way the learned model reflects characteristics which do not hold after the concept therefore leading to an increase of the prediction error [3].

**Change Detection.** *Change detection* refers to the process of identifying whether the incoming data points come from the same distribution observed till that point in time or if there was a concept drift in the underlying distribution [3]. Change detection methods should be able to detect and react to concept drift.

**Prequential Evaluation.** *Predictive Sequential* or *Prequential Evaluation* method is an evaluation method for data stream learning models, where each data point is first used to compute the prequential error of the incremental ML algorithm and then to train the model [22].

**Prequential Error.** *Prequential Error* is the error computed during prequential evaluation in incremental ML algorithms. The prequential error is the accumulated sum of the following loss function [22]:

\[
S_i = \sum_{1}^{n} L(y_i, \hat{y}_i)
\]  

### 2.3.2 Decision Trees

Decision Tree algorithms can be used both for regression and classification problems, with the only difference the type of the value that the DT model predicts for
new observations, numerical or categorical. We will further explain how a classification DT, as the one seen in Figure 2, is learned from a DT algorithm.

A DT algorithm is an iterative algorithm that learns a model which uses a number of decision rules to predict a categorical value for a data point. The attributes of the DT can be either categorical or numerical. In each step the algorithm sorts all data points in the leaves of the tree, with respect to the learned decision rules and then decides whether to split a leaf node or not.

As depicted in Figure 2, first there is only one node in the tree, the root node. Then the algorithm chooses the best attribute to split the root node, in our example this is attribute $a_4$, according to an evaluation metric, such as Entropy, Gain Ratio or Gini Impurity, which are presented in Section 2.3.2.1. Each attribute can be chosen as a splitting attribute only once in every branch of the tree. The algorithm iteratively splits the leaves of the tree until:

- Every data point sorted in a leaf belongs to the same class
- Every attribute has been used in each branch of the decision tree
- The tree reaches a user specified tree height

![Decision Tree Induction Process](image)

**Figure 2: Decision Tree Induction Process**

Decision Tree algorithms are popular in machine learning problems for a number of reasons,[20] but mainly because trees are simple to understand and interpret, since they can be visualized. Moreover, DT algorithms can handle both numerical and categorical attributes.

One of the disadvantages of DT algorithms is that they could overfit the training data, and thus could be less accurate in their predictions in comparison with other...
classification techniques. However, the overfitting problem could be solved by the use of techniques like bagging, and random forests, which aggregate a number of DT models for boosting the predictive accuracy of the DT model.

2.3.2.1 Evaluation Metrics for Decision Trees
All DT tree algorithms use an evaluation metric in order to decide which is the best attribute to split a leaf node. The metrics used by a DT algorithm measure if the children leaf nodes will have increased class homogeneity in case of splitting on an attribute. A leaf is considered homogeneous when it contains instances of one class only. Evaluation metrics that are used from different DT algorithms are presented below.

**Entropy & Information Gain.** The Entropy is a measure of class distribution in a node of a DT. The more homogeneous a class distribution in a leaf the smaller the Entropy. For a set of classes \( c_1, c_2, \ldots, c_n \), with \( p_{c_n} \) the probability of a class \( c_n \) in a leaf \( S \), Entropy is defined as follows:

\[
Entropy(S) = - \sum_{c_i} p_{c_i} \log_2 p_{c_i}
\]  

(2)

The smallest the Entropy of a children node with respect to a splitting attribute, the highest the Information Gain obtained by splitting in that attribute. The Information Gain of a node \( S \) for splitting in attribute \( A \) is defined as follows:

\[
Gain(S, A) = Entropy(S) - \sum_{v \in Values(A)} \frac{|S_v|}{|S|} Entropy(S_v)
\]

(3)

Even though the Information Gain is widely used in most classification DT algorithms, it has the disadvantage of being biased towards attributes that split to more nodes.

**Gain Ratio.** The Gain Ratio metric is a modification of the Information Gain metric in order not to bias multi-valued values, as it considers the number of branches when deciding for the splitting attribute. The Gain Ratio of a node \( S \) for splitting in attribute \( A \) is defined as follows:

\[
GainRatio(S, A) = \frac{Gain(S, A)}{- \sum_v \frac{|S_v|}{|S|} \log \left( \frac{|S_v|}{|S|} \right)}
\]

(4)
2.3.3 Streaming Machine Learning

2.3.3.1 Data Parallelization

Nowadays, most of the applications involve large-scale data sets, resulting in the need of parallel algorithms and techniques, in order to efficiently handle them. As we already mentioned in 2.1, there are different methods of parallelizing a problem, with data parallelism being the most commonly used also from current parallel models.

There exist two types of data parallelism, Horizontal Data Parallelism and Vertical Data Parallelism:

- **Horizontal Data Parallelism.** In Horizontal Data Parallelism, the system splits the data set horizontally as depicted in Figure 3a, based on the data set’s size, so in the special case of a distributed machine learning algorithm, each parallel processor will process a subset of instances of the data.

- **Vertical Data Parallelism.** In Vertical Data Parallelism, the system splits the data set vertically as depicted in Figure 3b, based on some characteristics of the data. In a machine learning algorithm, the data set is split with respect to the attributes of the instances, thus each parallel processor will process a number of attributes of the input data points.

Depending on the problem, different parallelization technique can be used to parallelize an algorithm.

![Figure 3: Types of Data Parallelism](image)

(a) Horizontal data parallelism  
(b) Vertical data parallelism
Horizontal Data Parallelism is suitable for ML problems of high throughput rate. Nevertheless, Horizontal Parallelism would add additional network and memory overhead, as the model should be communicated and replicated in all parallel processors.

Vertical Data Parallelism is suitable for ML problems that have a lot of attributes. Moreover, network and memory overhead can be avoided as the ML model is not communicated and replicated in all parallel processors.

2.3.3.2 A Use Case: Streaming Decision Tree Induction with Hoeffding Tree Algorithm

The HT algorithm is an incremental classification decision tree algorithm which was presented in [2]. The HT algorithm is cited below, as a distributed version of it, which was first proposed and implemented in Apache SAMOA [23] will be implemented with Apache Flink Streaming API in scope of this master thesis.

Classic decision tree algorithms such as CART and ID3 [24] require multiple pass over the data, which they assume that can be fitted in the main memory of a computer. Those assumptions do not hold in the data stream environment, where as already mentioned in Section 2.2, data have a high throughput rate and the amount of data may be infinite. The HT algorithm makes decisions for the best splitting attribute in any node only after examining incrementally a small subset of the training data set. The number of examples that needed to be seen before choosing the best attribute to split, is decided with the help of the Hoeffding bound:

$$\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n}}, \text{ where } R = \log_2 \# \text{classes}$$

The Hoeffding bound is a statistical result which states that, if \( \bar{r} \) is the mean of \( n \) independent observations of a real variable \( r \), then with probability \( 1 - \delta \), the true mean of the variable \( r \) is at least \( \bar{r} - \epsilon \) [2].

According to a number of research papers[2, 25, 26], HT algorithm manages to achieve results comparable to state-of-the-art batch algorithms in one pass over the data in small constant time per example, thus making it an appealing algorithm for real time analytics systems.
Algorithm 1: Hoeffding Tree[2]

**Input:**
- $S$ a sequence of examples
- $X$ a set of discrete attributes,
- $G(.)$ a split evaluation function
- $\delta$ 1 – the desired probability of choosing the correct attribute at any node.

**Result:** $HT$: a decision tree.

**Procedure** HoeffdingTree($S, X, G, \delta$)

Let $HT$ be a tree with a single leaf $l_1$ (the root).

Let $X_1 = X \cup X_0$.

Let $\overline{G}_1(X_0)$ be the $G$ obtained by predicting the most frequent class in $S$.

For each class $y_k$
- For each value $x_{ij}$ of each attribute $X_i \in X$
  - Let $n_{ijk}(l_1) = 0$.

For each example $(x, y_k)$ is $S$
- Sort $(x, y_k)$ into a leaf $l$ using $HT$.
- For each $x_{ij} \in x$ such that $X_i \in X_1$
  - Increment $n_{ijk}(l)$.

Label $l$ with the majority class among the examples seen so far at $l$.

If the examples seen so far at $l$ are not all of the same class, then:
- Compute $\overline{G}_i(X_1)$ for each attribute $X_i \in X_1 - X_0$
  - using the counts $n_{ijk}(l)$.

Let $X_a$ be the attribute with highest $\overline{G}_i$.

Let $X_b$ be the attribute with second-highest $\overline{G}_i$.

Compute $\epsilon$ using Hoeffding Bound Equation.

**if** $\overline{G}_i(X_a) - \overline{G}_i(X_b) > \epsilon$ and $X_a \neq X_0$ **then**

Replace $l$ by an internal node that splits on $X_a$.

For each branch of the split
- Add a new leaf $l_m$, and let $X_m = X - \{X_a\}$.
  - Let $\overline{G}_m(X_0)$ be the $G$ obtained by predicting the most frequent class at $l_m$.

For each class $y_k$ and and each value $x_{ij}$ of each attribute $X_i \in X_m - X_0$:
  - Let $n_{ijk}(l_m) = 0$.

**end**

Return $HT$. 
The HT algorithm was designed as a serial algorithm, but the algorithm’s data flow could be represented as a pipeline of two tasks. As depicted in Figure 4, the HT algorithm could be represented by a data source which emits data points, then possibly a mapper function that takes care of preprocessing the data points by mapping them in a format appropriate for the HT algorithm and then the HT algorithm operator, which receives the data points, updates incrementally the per node metrics and grows the HT when sufficient data points have been seen.

![Figure 4: Hoeffding Tree algorithm pipelined data flow.](image)

### 2.3.3.3 Vertical Hoeffding Tree

One of the most prominent ways to parallelize the HT algorithm is the vertical parallelization of the algorithm as it is presented and implemented in the Apache SAMOA platform [27]. In [28], the vertical parallelization of the HT algorithm is presented and explained in detail.

As depicted in Figure 5, VHT is constructed with a hybrid parallelism scheme. There is a pipeline of operators following the source; Model Aggregator (MA) and Local Statistic (LS) operator, where the Local Statistic operator is data parallel. The cyclic topology of the operators complicates the pipelining but is the main characteristic of the VHT algorithm that makes it possible to parallelize a decision tree induction algorithm.
Figure 5: Vertical Hoeffding Tree data flow in Apache SAMOA system

**Model Aggregator (MA).** The *Model Aggregator* aggregates statistics from the local statistic operators and constructs the global decision tree. Specifically, the model aggregator:

- receives all the data point instances from the source, sorts them to a leaf of the global decision tree, breaks each instance into its features and sends the features to the local-statistic operators
- when a minimum number of instances is sorted into a leaf, then broadcasts a control message to the local statistic operators in order to calculate and suggest the best attribute to split the specific leaf
- aggregates the results from the local statistics, finds the best attribute to split the leaf and decides whether to split or not the leaf (with respect to the information gain measure)

**Local Statistic (LS).** The *Local Statistic* operator receives the attributes which are assigned to it and incrementally updates the metrics of the attributes. When a control message is received from the *Model Aggregator*, the *Local Statistic* calculate the two best attributes to split the leaf from the ones assigned to them and send their suggestions to the *Model Aggregator*.

### 2.4 Apache Flink

Apache Flink is a big data processing engine which offers an open source software stack for implementing large scale data processing applications. MapReduce programming model forms the base of Apache Flink’s programming model, but it is enriched with numerous high level transformations like Join, Filter, Aggregations(Sum, Min, Max), Iterations, etc. Moreover, Flink generalizes the (Key, Value) pair data model of MapReduce by enabling the use of any Java or Scala data types [8].

Apache Flink offers both batch and streaming APIs which can be bundled with domain-specific libraries (e.g. Machine Learning Library) to create fast and scalable big-data applications. A unified batch and streaming runtime environment is responsible for the execution of both batch and streaming jobs. In Figure 6 the component stack of the Apache Flink engine as of today, as well as the general life cycle of a program is depicted [1].
When submitting a program in a Flink cluster, a client compiles and pre-process the program to produce a Directed Acyclic Graph (DAG). A DAG is a tree representation of the operations that should be applied over the data set. Each node of the DAG represents an operator (e.g. Map, Reduce, Join) and each edge of the DAG represent the data flow over the operators.

Then the optimizer inspects the DAG to decide for possible performance optimizations for the Flink job data flow (e.g. reordering of operators). The runtime layer receives from the optimizer the Flink program in the form of a JobGraph, which is a representation of the programs data flow (operators and intermediate results). The JobGraph can then be executed by the Flink Runtime. The Runtime is responsible both for scheduling tasks to different task managers, as well as to monitor the state of jobs in the the task managers. Apache Flink supports a number of different cluster managers and storage systems like HDFS [16] and S3 [29].

Following the Word Count example, an equivalent of the "Hello World" example for Big Data systems is presented in Listing 1. The example is implemented in Apache Flink Streaming Scala API and counts the occurrences of a word in the following steps; first it reads the text from the input file and then splits the text into words with the use of a flatmap function. Then the words are grouped and their occurrences are summed up, with the groupBy and sum functions correspondingly.
object WordCount {
  def main(args: Array[String]) {

    val env = StreamExecutionEnvironment.getExecutionEnvironment
    val text = env.readTextFile("/path/to/input_file")

    val counts = text.flatMap { _.toLowerCase.split("\W+") } filter { _.nonEmpty }
                 .map { (_, 1) }
                 .groupBy(0)
                 .sum(1)

    counts.print
    env.execute("Scala Stream WordCount")
  }
}

Listing 1: Scala Word Count example in Apache Flink Streaming API

2.5 Related Work

The emerging need for incremental machine learning applications has resulted to a number of systems that are early adopters of the streaming paradigm for machine learning. In this Section some of those applications will be further presented. In [30] a competitors analysis regarding the new feature for real-time predictive analytics in Apache Flink is conducted.

**Apache Spark.** *Apache Spark* [17] is a big data cluster computing system for general execution graphs. It provides a high-level API in Java, Scala, Python and R and both a batch and streaming processing API. Spark Streaming divides the stream of input data in batches called Resilient Distributed Dataset (RDD) and sends the RDDs to the main Spark engine where each operation is performed in an RDD (batch of input data). Thus, Apache Spark streaming could be characterized not to be a real streaming system as Apache Flink and Apache Storm, where each operator is applied in every input data element. Apache Spark offers MLlib, an ML library which provides a number of algorithms like classification, regression and more. A few algorithms (e.g. Streaming KMeans, Streaming Linear Regression) are offered to incrementally learn the model and apply it on the streaming data, but most of the ML algorithms can be used to learn the model from historical data (in a batch mode) and then apply it on streaming data.

**Apache Storm.** *Apache Storm* [31] is a distributed computation system in which each operator is applied in every input element. Apache Storm offers Trident-ML,
2 THEORETICAL BACKGROUND

an online ML library, the algorithms of which learn the model incrementally. Nevertheless, the Trident-ML algorithms do not work in a distributed way, but the update of the model will be performed only on a single machine. Even though other tasks of the algorithm, like data preprocessing could be performed on parallel, the non distributed update the model could be considered a possible bottle neck of the library which decreases the through put of the algorithm in a significant way.

MOA. MOA [32, 33] is a framework, written in Java, for implementing incremental machine learning from evolving data streams. It offers a graphical user interface which enables the user to select any of the available stream sources, define the on-line algorithm to be used as well as the evaluation method. MOA provides a number of data stream classification methods (e.g. Hoeffding Tree), data generators.

Apache SAMOA. Apache SAMOA [23, 27] is a "descendant" of MOA, builds on existing distributed processing engines such as Apache Storm[31] and Yahoo S4 [34] to offer a compositional stream-based API for defining and deploying incremental machine learning algorithms on the cluster. Consequently, Apache SAMOA provides a collection of state-of-the-art implementations of known incremental ML algorithms for further usage (e.g. classification, regression, clustering, outlier detection, and concept drift detection and recommender systems).
3 Incremental Machine Learning Pipelines on Apache Flink

In this section, the description of the incremental ML pipelines will be given. In scope of this thesis, part of the batch processing pipeline was redefined to streaming API. Moreover, the extension of the pipeline by specifying and implementing additional abstractions will be presented. Following the presentation of the Framework pipeline abstractions which were defined for the streaming-ml library. Then the framework implementation, were the class diagrams and the code listings defining the abstractions are presented. The ML pipeline framework, presented in this thesis can be used to implement different types of ML algorithms on Apache Flink Streaming API. Two variations of the HT algorithm were implemented and evaluated with the use of those abstractions. The implementation of all the algorithms is presented in Section 4. A complete package diagram, visualizing the system design can be found in the Appendix Section B.1. All UML class diagrams were designed with Visual Paradigm software [35].

3.1 Modeling Machine Learning Pipelines on Apache Flink Streaming

As we have already mentioned, machine learning is a broad field of computer science which includes numerous machine learning problems, such as classification, regression, clustering, pattern recognition, artificial neural networks and more. Irrelevant from the machine learning problem and method that can be used to solve it, we can identify a number of abstractions which can be used to model any of those problems and methods. In this section we define the abstractions that could model any machine learning task, providing in that way the user with a machine learning pipeline.

Transformer

A Transformer is an operator which applies a deterministic transformation function, to each point of the input data stream. If \( x_1, x_2, \ldots, x_n \) are the input points to the transformer with transformation function \( \Phi \), then each data point \( x_i \) is transformed to the value \( z_i = \Phi(x_i) \). The output of a Transformer is the data stream to be used for learning an ML model.
3 INCREMENTAL MACHINE LEARNING PIPELINES ON APACHE FLINK

Figure 7: Transformer abstraction for Machine Learning pipelines.

In the machine learning context a transformer is usually used for preprocessing of the input data in order to transform it to a form which will meet the requirements of the implemented machine learning algorithms. Data transformation can have a significant impact on the performance of the learned model. There are a number of transformations which can be applied to input data streams such as feature scaling and normalization, continuous feature discretization, feature extraction and more[36].

**Learner**

A Learner is an operator which takes as an input two data streams; a Training Data Stream and a Test Data Stream. The Learner uses the Training Data Stream to "learn" or else build a model which will further be used to make data-driven predictions for the Test Data Stream. The predictions for the Test Data Stream will be the output of the Learner.

Depending on the type of the machine learning problem, a Learner can build a model which either maps input data elements to target output values or recognizes patterns on the input data elements.

Figure 8: Learner abstraction for Machine Learning pipelines.

**Evaluator**

An Evaluator is an operator which takes as input a data stream and evaluates
the model learned from the Learner with respect to its predictive accuracy for data elements that have not been used for training in the learning phase. The output of an Evaluator is the prediction error of the model.

In the data stream context, an evaluator could be defined as a function $L$ that maps each element $t_1, t_2, \ldots, t_k$ of the input data stream, to an output data stream that represents the prediction error, as follows: $e_i = L(t_i)$.

![Evaluator abstraction for Machine Learning pipelines.](image)

**Figure 9:** Evaluator abstraction for Machine Learning pipelines.

**Change Detector**

A Change Detector is an operator that takes as input the a data stream and detects changes in the underlying distribution of the input data elements. In streaming machine learning, the input data stream of a change detector is the error data stream outputted from the Evaluator. The output of a Change Detector is a data stream of signal which declare whether a change was detected.

![Change Detector abstraction for Machine Learning pipelines.](image)

**Figure 10:** Change Detector abstraction for Machine Learning pipelines.

All the above defined abstractions can be used to build a machine learning pipeline for all the ML tasks. A visual representation of the ML pipeline can be found in Figure 11.
3.1.1 Document Classification Machine Learning Pipeline Example

In order for the reader to better understand the need and use of the ML pipeline presented above, we will give an example in this section which will design a solution to a text classification problem using our modeled pipeline.

Document classification is a popular ML application, were documents have to be assigned to classes (or categories). One of the main problems of a document classification algorithm is that we can not use raw documents as input to the algorithm but we have to process them. Thus, in order to classify the documents a number of stages should be followed.

As we see in Figure 11, first a *tokenizer* transformer will take as input the documents and will extract the words presented in them. Then the tokens will be fed as input in a second transformer, which will extract features by performing *feature hashing* to the input tokens. After the two transforming stages, the transformed data set can be fed into the *Learner*, which in our example is a *logistic regression* algorithm. The *Learner* will learn a model from the input transformed data set. The learned model as the output of the *Learner* is fed to the evaluator, where its accuracy can be measured with the use of test data.
3.2 Machine Learning Pipeline Implementation

This section presents the implementation of the ML pipeline abstractions modeled in 3.1. The abstractions of Transformer and Learner were redefined from batch Flink-ML API. Both are base traits which will be implemented by ML and data preprocessing algorithms respectively. The traits of Transformer and Learner are under the common package as depicted in the class diagram in Figure 13. In addition to redefining the abstractions of Transformer and Learner, in this section we present two more abstractions which were designed and implemented for the incremental ML library.
3 INCREMENTAL MACHINE LEARNING PIPELINES ON APACHE FLINK

3.2.1 Transformer

Base trait for an algorithm which transforms the input data to some output data. Data transformations can be feature extraction, scaling or centralization of the features or even completing missing feature values of the data points.

```scala
/** Ported-in from Batch flink-ml library. 
 * @tparam IN Type of incoming elements 
 * @tparam OUTS Type of outgoing elements 
 */
trait Transformer[IN,OUT] extends WithParameters {
  def transform(input: DataStream[IN], transformParameters: ParameterMap = ParameterMap.Empty):DataStream[OUT]
}
```

Listing 2: Transformer Base Trait

In the above Listing 2, is presented the formal definition of the Transformer trait. All Transformer implementations have to implement the method transform, which defines the logic of how the input data will be transformed into the output data.

3.2.2 Learner

In the following Listing 3, the formal definition of the Learner trait is presented.

```scala
/** Ported-in from Batch flink-ml library. 
 * @tparam IN Type of the training data 
 * @tparam OUT Type of the trained model 
 */
trait Learner[IN, OUT] extends WithParameters {
  def fit(input: DataStream[IN], fitParameters: ParameterMap = ParameterMap.Empty): DataStream[OUT]
}
```

Listing 3: Learner Base Trait

Learner is the base trait for a streaming algorithm which trains a model based on some training data. Any Learner implementation has to implement the fit method which takes the training data and learns a model from the data.

3.2.3 Evaluator

An Evaluator is the base trait for an algorithm which evaluates the current model for the input data points. The formal definition of the Evaluator trait is presented
3 INCREMENTAL MACHINE LEARNING PIPELINES ON APACHE FLINK

in Listing 4.

```scala
/**
* @tparam IN Type of incoming elements
* @tparam OUT Type of outgoing elements
*/
trait Evaluator[IN, OUT] {

/**
* @param input The points to be used for the evaluation.
* @return The Prediction error or models accuracy
*/
def evaluate(input: DataStream[IN]): DataStream[OUT]
}
```

Listing 4: Evaluator Base Trait

An Evaluator implementation has to implement the method `evaluate`, which measures how well the current model performs with the incoming data set, by comparing the predicted classification or regression value for a specific data point with the real value.

In traditional batch ML algorithms there exist a number of alternatives for evaluating the model learned from an algorithm. Nevertheless, in the data stream mining environment, two main evaluation methods are there [22, 37]:

- **Holdout test set**: Apply current model to the test set at intervals

- **Prequential (Predictive Sequential)**: A sequence of examples is used first to compute the model to error and then to train the model.

For the evaluation of the HT algorithm variations presented in Section 4.1, the Prequential evaluation method was used. As depicted in the class diagram in Figure 14, the PrequentialEvaluator class implements the Evaluator interface.
3 INCREMENTAL MACHINE LEARNING PIPELINES ON APACHE FLINK

Figure 14: Evaluator package class diagram in incremental-ml library on Apache Flink.

3.2.4 ChangeDetector

ChangeDetector trait, which is presented in Listing 5, is the base trait for an algorithm which detects concept drift on the input data. Due to the fact that data streams can be infinite, there exist a real need of identifying as soon as possible, changes that can naturally occur in the underlying process over time.

```
trait ChangeDetector[Double, Boolean] extends WithParameters {

  /**
   * @param inputPoint the new input points to change detector
   * @param parameters the parameters
   * @return Whether a change was detected (true) or not (false).
   */
  def detectChange(inputPoint: DataStream[Double], parameters: ParameterMap = ParameterMap.Empty): DataStream[Boolean]
}
```

Listing 5: ChangeDetector Base Trait

All ChangeDetector implementations have to implement the method detectChange, which identifies whether there is a change in the data distribution or not. Change detection algorithms take as input the prediction error of the ML model and try to identify if there was a change in the underlying distribution.

Page Hinkley Test algorithm was implemented and added to the incremental-ml library of Apache Flink for detecting changes in the underlying distribution. As depicted in the class diagram in Figure 15, the PageHinkeyTest class implements the ChangeDetector interface.
Figure 15: ChangeDetector package class diagram.
4 Algorithm Implementation

4.1 Hoeffding Tree on Apache Flink

In this section we present the native implementation of the VHT algorithm in Apache Flink in Section 4.1.1, as well as another variation of the HT algorithm, the RmVHT in Section 4.1.2.

In order to represent the DT model, a DecisionTreeModel class was defined. The DecisionTreeModel implements a number of methods which enable the initialization of the DT model, the growth of the model by splitting the nodes of the tree when required, as well as labeling the leaves of a node with a class label. A DecisionTreeModel object is mainly a collection of decision tree nodes, represented by the DTNode class.

The DTNode class is an inner class of the DecisionTreeModel which offers the access to the splitting nodes, creating child nodes and append them to the existing DT. The class diagram of the package containing the implementation of VHT algorithm is presented in Appendix Section B.2.1.

4.1.1 Centralized Model of VHT Algorithm

As depicted in the class diagram of the VHT algorithm in Figure 34, VHT is a Learner which incrementally learns a classification DT model. The VHT algorithm which was ported from the Apache SAMOA platform [27], is a centralized algorithm, with vertical data parallelization. The data flow of the centralized VHT algorithm is depicted in Figure 16 and as seen it can be decomposed in two components:

- GlobalModelMapper: An operator that holds and decides for growing the global unique model learned from the algorithm
- PartialVHTMetricsMapper: An operator that performs the processing of the different attributes for the algorithm in parallel.
GlobalModelMapper. GlobalModelMapper is an operator that receives as input both data points from the source as well as the partial results for the proposed best attributes to split a specific node from the PartialVHTMetricsMapper. Depending on the input object, the GlobalModelMapper performs different operation:

- **DataPoint:** Upon receiving a DataPoint from the source the GlobalModelMapper performs the following:
  
  - Sorts the DataPoint into a tree leaf with respect to the node splitting conditions already learned form the DT model
  
  - Splits the DataPoint into its attributes, wraps the attribute id and value, the id of the leaf that it was sorted into and the class label of the input data point on an VHTattributes object and sends the attributes on the

Figure 16: Data flow of native VHT implementation on Apache Flink
PartialVHTMetricsMapper operators.

- In case that a sufficient number of input data points has been sorted in the specific leaf, broadcasts a CalculateMetricsSignal signal to the PartialVHTMetricsMapper operators to calculate their local two best attributes to split the specific leaf.

- **Local Best Splitting Attribute:** Upon receiving an EvaluationMetric object from a PartialVHTMetricsMapper operator with its local two best attributes to split:
  
  - Appends the PartialVHTMetricsMapper on a list of EvaluationMetric objects received for the specific leaf from the PartialVHTMetricsMapper operators.
  
  - If an EvaluationMetric has been received from all PartialVHTMetricsMappers for that leaf, then it finds the two best attributes to split from the proposed ones, and checks if the splitting condition is fulfilled. If the condition is fulfilled, splits the leaf node otherwise it drops the proposed splitting values.

**PartialVHTMetricsMapper.** PartialVHTMetricsMapper is an operator that receives input from the GlobalModelMapper operator both a VHTattributes and a CalculateMetricsSignal signal. Depending on the input object, the GlobalModelMapper performs different operation:

- **VHTattributes:** Upon receiving an attribute the PartialVHTMetricsMapper incrementally updates the metrics of the attribute for the leaf in which the attribute was sorted.

- **CalculateMetricsSignal:** Upon receiving a signal, the PartialVHTMetricsMapper calculates the two best attributes to split the specified in the signal leaf node and then sends them to the GlobalModelMapper operator, which will decide whether to split the leaf or not.
4.1.2 Replicated Model of VHT Algorithm (RmVHT)

The *Replicated Model* variation of the HT presented in this section has two main differences from the VHT:

1. Addition of *DecisionMaker* operator between the *PartialVHTMetricsMapper* and the *GlobalModelMapper*.
2. Replication of the Global Model Parallelism, by increasing the parallelization of the *GlobalModelMapper* operator.

In data streaming machine learning context the speed of the input elements from the source to the *Learner* is often too high. A parallelization of the *GlobalModelMapper* operator can achieve a higher throughput of the algorithm which will result in processing much more input *DataPoint* instances per second from the source.

In order to keep the cohesion between the parallel models learned in each of the *GlobalModelMapper* operators, the decision about the best attribute to split each node is done on a new operator called *DecisionMaker*. Even though we still have
a centralized parallel algorithm, the rate of EvaluationMetric objects received in the DecisionMaker is much lower.

**DecisionMaker.** The DecisionMaker operator receives all the local two best attributes to split from the PartialVHTMetricsMapper, finds the two global best attributes to split and broadcasts them to the GlobalModelMapper operators. In that way, all GlobalModelMapper operators will learn the same model, maybe with a minor delay between each other.

### 4.1.3 Prequential Evaluator

Prequential Evaluator was implemented to measure the error of the model learned by the VHT classifier.

As we already referred in 2.2.1, the prequential error is incrementally computed, as the sum of a loss function, with respect to the observed and predicted values. For classification problems, the prequential error and loss function is defined as follows:

\[
S_t = \sum_{i=1}^{n} L(y_i, \hat{y}_i), \text{ where } L = \begin{cases} 
1, & \text{if } y_i \neq \hat{y}_i \\
0, & \text{if } y_i = \hat{y}_i 
\end{cases}
\]  

(7)

**Algorithm 2:** Prequential Evaluator Algorithm pseudo code

- **Input:** (prediction, class) Predicted and real class value data stream
- **Result:** $e$: Prequential Error
- **Procedure Prequential Evaluator:**
  - var instancesSeen = 0
  - var missclassifiedInstances = 0
  - for each input tuple of the data stream do
    - instancesSeen += 1.0
    - if (prediction $\neq$ class) then
      - missclassifiedInstances += 1
    - end
  - return instancesSeen / missclassifiedInstances

### 4.2 Page Hinkley Test Change Detector

Page Hinkley Test (PHT) is a broadly used algorithm for detecting change on data streams [3, 37]. In [3] a comparison between different change detection
algorithms was presented and the PHT was found to have a high performance, with respect to the number of correct change detections. The high performance of the algorithm as well as its simplicity, formed the rational for deciding to implement first this change detection algorithm for the incremental-ml library.

Algorithm 3: Page Hinkley Test algorithm [3]

Input: $x_1, x_2, \ldots, x_{t=T}$ data stream
\hspace{1cm} $\delta$ magnitude threshold
\hspace{1cm} $\lambda$ detection threshold

Result: CDFlag: True when a change was detected.

Procedure Page Hinkley Test($X, \delta, \lambda$)
for each point of the data set do
\hspace{1cm} $\bar{x}_T = \frac{1}{T} \sum_{t=1}^{T} x_t$
\hspace{1cm} $U_T = \sum_{t=1}^{T} (x_t - \bar{x}_T - \delta)$
\hspace{1cm} $m_T = \min(U_t, t = 1 \ldots T)$
\hspace{1cm} if $(PH_T = U_T - m_T > \lambda)$ then
\hspace{1cm} \hspace{1cm} return true
\hspace{1cm} end
end

In the PHT algorithm which is presented in the pseudo-code in algorithm 3, the variable $U_T$ is defined as the cumulative difference between the data observation and the mean value of the observations. The $\delta$ variable defines the magnitude of the differences that can be observed in the data sequence, without being considered a change. A change on the underlying sequence is reported when the difference of the cumulative variable $U_T$ and the minimum value of all observed $U_T$ is greater than the variable $\lambda$. Variable $\lambda$ influences the number of detected changes, since by increasing $\lambda$ changes of bigger magnitude are allowed.

4.2.1 AttributeObserver

AttributeObserver is the base trait for a class that will keep the metrics for either a Nominal or a Numerical attribute. The metrics are important statistics which are needed from a decision tree Learner for learning the model. The formal definition of the AttributeObserver trait is presented in Listing 6. In the implementation of the VHT and RmVHT the Information Gain evaluation metric was used for deciding whether to split on an attribute or not in each of the nodes.
trait AttributeObserver[IN] extends Serializable {

/** @return (InformationGainMetric,listOfSplittingValues). */
def getSplitEvaluationMetric: (Double, List[Double])

/** It is called for incrementally update of the metrics */
/** for a specific attribute. */
def updateMetricsWithAttribute(attribute: IN): Unit
}

Listing 6: AttributeObserver Base Trait

Figure 18: AttributeObserver package class diagram.

The AttributeObserver trait as well as the two classes implementing it NumericalAttributeObserver and NominalAttributeObserver are in the attributeObserver package, as seen in the class diagram in Figure 18.

An AttributeObserver implementation has to implement the getSplitEvaluationMetric and updateMetricsWithAttribute methods. Depending on the type of the attribute, nominal or numerical, both functions have different functionality which is further explained in Sections 4.2.1.1 and 4.2.1.2.

4.2.1.1 NominalAttributeObserver
A NominalAttributeObserver is created to incrementally update the statistics which are needed for a nominal attribute, in order to decide which will be the entropy for splitting a node in that attribute. The only variable of the NominalAttributeObserver constructor is the numberOfClasses which defines the total number of classes of the specific classification problem.

Statistical Metrics
For the nominal attributes, the class distribution for each distinct value of the attribute is incrementally updated with every new data observation. Specifically, a variable of type "mutable.HashMap[Double, List[Int]]", called attributeValues is updated by the function updateMetricsWithAttribute every time a new value of that attribute is observed.

The (Key,Value) instances of the variable attributeValues: mutable.HashMap[Double, List[Int]] represent:

- Double (Key): The distinct values of an attribute
- List[Int] (value): The distribution of classes per attribute value.

Calculation of the Information Metric

The entropy for each attribute is used for deciding its appropriateness as a splitting attribute of a node. The function getSplitEvaluationMetric of a NominalAttributeObserver is called for calculating the entropy obtained in case of splitting a node in the specific nominal attribute. The entropy is calculated with the use of the class distribution statistics which are updated from the updateMetricsWithAttribute function and returns a tuple of type (Double, List[Double]), where the first item is the entropy for that attribute and the second item of the tuple is a List with the splitting values of the attribute.

4.2.1.2 NumericalAttributeObserver

A NumericalAttributeObserver is created to incrementally update the statistics which are needed for a numerical attribute, in order to decide which will be the entropy for splitting a node in that attribute. A NumericalAttributeObserver constructor takes two variables, the number of classes of the classification problem and the attribute Id.
Statistical Metrics
For the numerical attributes the following statistics are incrementally updated, for every observation of the attribute, from the `updateMetricsWithAttribute` function:

- **minMaxValuePerClass**: The minimum and maximum values observed per class for that attribute
- **meanStdPerClass**: The mean and standard deviation of the attribute values’ observed per class for the specific attribute

Calculation of the Information Metric

The `getSplitEvaluationMetric` function of an `NumericalAttributeObserver` is called for calculating the entropy obtained in case of splitting a node in the specific numerical attribute. The entropy is calculated with the statistics which are updated from the `updateMetricsWithAttribute` function and returns a tuple of type `(Double, List[Double])`, where the first item is the entropy for that attribute and the second item of the tuple is a List of size 1, containing the splitting value of the numerical attribute.

Deciding on best value for binary split of numerical attributes

There exist a number of methods regarding the identification of the best value to binary split a numerical attribute [26, 38]. In the implementation of VHT on Apache Flink, the **Gaussian Approximation** method, which was presented on [26] was used, in order to decide the information gain of splitting on a value.

Supposedly that we have a three class classification problem and at least a numerical attribute in our data points. After a number of data items has been observed and used to update the `meanStdPerClass` metrics of the attribute, the classes distributions for the numerical attribute could look as in Figure 19.

Deciding for the best point to split a numerical attribute has two stages:

- **Identification**: a uniform selection of a constant number of potential split points between the minimum and maximum observed values for this attribute. The selection of the potential splitting points is performed with the method presented in pseudo-algorithm 4.
- **Evaluation**: Evaluation of each identified splitting point using the Gaussian distributions of the classes, as presented in pseudo-algorithm 5. The split-
ting point that has the smallest entropy, thus will give the highest information gain, will be chosen as the best attribute for splitting the node.

![Gaussian approximation for deciding on splitting a numerical attribute on sp5 splitting value.](image)

**Figure 19**: Gaussian approximation for deciding on splitting a numerical attribute on \( sp5 \) splitting value.

The identification part of deciding for the splitting points is pretty obvious, so the evaluation stage of the potential splitting points and the selection of one of those will be further explained here with an example. Let \( sp5 \), depicted in Figure 19, be the splitting point to evaluate for a 3-class classification problem. The algorithm presented in 5, for all three classes, calculates the number of the attribute instances per class that have:

- **value < sp5**, are categorized as instances that would go to the *right* hand child in case of split in \( sp5 \)

- **value >= sp5**, are categorized as instances that would go to the *left* hand child in case of split in \( sp5 \)

In that way the number of occurrences of each of the classes in the children nodes can be calculated enabling the calculation of the entropy for splitting in \( sp5 \).
Algorithm 4: Identifying potential split points of a numerical attribute.

| Input: k     | number of points |
| min         | minimum observed attribute value |
| max         | maximum observed attribute value |
| Result: SP  | List of potential splitting points. |

\[ c_i = \min + \frac{i \cdot \max - \min}{k + 1} \]

Algorithm 5: Gaussian approximation for evaluating potential splitting points with respect to entropy.

| Input: SP   | List of potential splitting points |
| Result:     | (entropy, bestValueToSplit) |
| Procedure   | decideBestValueToSplit{ |
| var leftHandSide |
| var rightHandSide |

for each splitting point sp in SP do
    for each class c do
        if (sp < c.minValue) then
            rightHandSide += class.numberOfInstances
        end
        else if (sp >= c.maxValue) then
            leftHandSide += class.numberOfInstances
        end
        else
            //use cumulative distribution function (cdf)
            leftHandside += class.cdf * class.numberOfInstances
            rightHandside += (1 - class.cdf) * class.numberOfInstances
        end
    end
5 Creating a Flink Adapter on Apache SAMOA

Apache Scalable Advanced Massive Online Analysis (SAMOA) is a platform for mining data streams with the use of distributed streaming Machine Learning algorithms, which can run on top of different Data Stream Processing Engines (DSPE)s.

As depicted in Figure 20, Apache SAMOA offers the abstractions and APIs for developing new distributed ML algorithms to enrich the existing library of state-of-the-art algorithms [27, 28]. Moreover, SAMOA provides the possibility of integrating new DSPEs, allowing in that way the ML programmers to implement an algorithm once and run it in different DSPEs [28].

An adapter for integrating Apache Flink into Apache SAMOA was implemented in scope of this master thesis, with the main parts of its implementation being addressed in this section. With the use of our adapter, ML algorithms can be executed on top of Apache Flink. The implemented adapter will be used for the evaluation of the ML pipelines and HT algorithm variations.

![Figure 20: Apache SAMOA's high level architecture.](image)

5.1 Apache SAMOA Abstractions

Apache SAMOA offers a number of abstractions which allow users to implement any distributed streaming ML algorithms in a platform independent way. The most important abstractions of Apache SAMOA are presented below [27, 28].
**Processor.**  
*Processor* is the basic logical processing unit, where all logic of the algorithm is written. The parallelism of processors can be specified by the user. In Apache SAMOA, there are two types of *Processors*:

1. *Entrance Processors*: Processors which generate data streams and are used as the source of the topology.
2. *Simple Processors*: Processors which operate over an incoming stream of data.

**Processing Item (PI).**  
*Processing Item* is a wrapper of the processor. There are two type of *Processing Items*: *Simple Processing Items* and *Entrance Processing Items*, depending on the type of *Processor* that they wrap.

**Stream.**  
A *Stream* is a connector between PIs and it is associated only with one source PI. In Apache SAMOA there are three routing mechanism for the events emitted by the PIs:

1. *Shuffle Grouping*: Events are routed in a Round-Robin way
2. *All Grouping*: Events are broadcasted to all PIs
3. *Key Grouping*: Events with the same key are always routed to the same PIs

**Content Event.**  
*Content Event* is a reusable object that wraps the data transmitted from one *Processing Item* to another via a *Stream*.

**Topology.**  
A *Topology* is a collection of connected processing items and streams, which is defined inside a *Task*.

**Task.**  
A *Task* is an interface with represents an execution entity similar to a job in Apache Hadoop. In SAMOA, only classes which implement the *Task* interface can be executed.

### 5.2 Flink Integration to SAMOA

As we have already mentioned a Flink program is represented by a DAG of transformation operators over the input data streams. Therefore, a SAMOA topology is transformed to a *DAG* of Flink’s custom operators which encapsulate SAMOA’s Processing Items and Flink’s stream partitioning is used in order to simulate
SAMOA’s stream routing mechanisms between operators. The most important issues of the transformation of a SAMOA topology into a Flink DAG of operators are presented below. The implemented adapter can be found in the incubator-samoa github repository [39].

**Circle Detection in SAMOA topologies**

Most of the streaming ML algorithms implemented in Apache SAMOA construct topologies which contain circles. In Apache Flink Streaming only explicit circles are allowed in the DAG topology, thus any circles contained in a SAMOA topology have to be detected in advance.

Before the initialization of the SAMOA topology, a variation of the Depth First Search (DFS) algorithm is used in order to identify the Processing Items which construct a circle. After detecting any potential circles in the topology, Flink’s operators and streams are initialized in an incremental way, from the sources to the sinks and also circles are explicitly defined in the DAG. Detected circles are implemented in Flink iterations operators.

**SAMOA streams to Flink Data Streams**

In Apache SAMOA topologies, a *Processing Item* can have more than one output streams, but Flink operators, which as already mentioned encapsulate the *Processing Items*, can have only one output stream. In order to facilitate more than one output streams in one operator the *split* and *select* Flink operators were used. *Split* operator is used to tag each data element of a stream with respect to a specific condition (e.g. the type of emitted events). Then the *select* operator can be used to create data streams which will receive only the elements with a specific "name tag" (thus fulfilling a specific condition). An example of the use of the operators is depicted in Listing 7.

```scala
val split = outputDataStream.split(outputSelector).
val evaluatorDs = split.select("evaluatorEvent");
val metricsMapperDs = split.select("PartialMetricMappersEvent");
```

Listing 7: Scala code example of split and select operators in Flink

**Composing named streams on Flink**

When a new SAMOA stream is created, in Flink a named stream is composed by tagging a Flink DataStream with a name. Then, during the initialization of the topology, data streams are filtered in each operator’s output, with respect to their "tag", so that they can be routed to the right operator respectively. In that way, whenever we define a processor to take as an input a Flink DataStream, we implicitly select a filter.
6 Experimental Evaluation

6.1 Evaluation Pipeline of Hoeffding Tree Variations in Apache Flink

In order to evaluate the implementation of the two variations of HT algorithm in Apache Flink, the ML pipeline presented in Figure 21 was used.

The source provides instances of *LabeledVectors* to a mapper which then wraps to *DataPoints* object in order to send them to the VHT algorithm. For each of the *DataPoints* instances the VHT sends an *InstanceClassification* to the *Evaluator*, so as to incrementally update the *Prequential Error*. The *Prequential Error* is the main evaluation metric of the algorithm. Example Scala code constructing the pipeline is presented in Listing 8.

```scala
// Evaluation pipeline for VHT algorithm in Apache Flink Streaming
*

def main(args: Array[String]) {

  val inputDataStream = readtextFile("FileName").map(dp => DataPoints(dp))
  val vhtLearner = VerticalHoeffdingTree(env)
  val evaluator = PrequentialEvaluator()

  val learnerDatastream = vhtLearner.fit(inputDataStream,parameters)
  val errorDataStream = evaluator.evaluate(learnerDatastream)
}
```

Listing 8: Scala code of the pipeline used for evaluating VHT.
6.2 Experimental Setup

All algorithms have been implemented in Scala. The experiments were run on a machine with quad-core Intel Core i7 @ 2.20 GHz and 16 GB of RAM. For each data set and selection of parameters, the experiments were run five times to avoid possible oscillation due to the order of data points received from the sources. The implementation of VHT algorithm on Apache SAMOA Framework run over Apache Flink Streaming Processing Engine.

The algorithm specific parameters of VHT both in Apache Flink and Apache SAMOA were the following:

- \( \text{MinNumberOfInstances: 200} \), the minimum number of instances that each leaf have to “see” before checking whether to split or no.
- \( \tau: 0.05 \), tie breaking parameter which is used when the two best attributes to split have a difference less than the value of Hoeffding Bound.
- \( \delta: 0.000001 \), parameter used for the calculation of the Hoeffding Bound.

The same value for the parameters was used for the RmVHT algorithm variation in Flink.

6.2.1 Data Collection

Most of the data sets was selected from those used in previous investigations, such as the Forest Covertype, Waveform21 and Led. In addition, HIGGS data set was selected as an additional artificial data set. A description of the main characteristics of the datasets as well as the main source to the data sets can be found in the Appendix A.

6.2.2 Data Analysis

We compare the performance of the two variations of HT algorithm; the VHT and RmVHT, on Apache Flink with respect to the following evaluation variables:

- Prequential classification error, \( e \)
- Average number of instances “seen” between node splits
- Average time delay between node splits
Moreover, the same evaluation variables are used to compare the native implementation of VHT algorithm on Apache Flink with the VHT implementation on Apache SAMOA.

For the evaluation of the results, the correlations between the different evaluation metrics and input parameters were analyzed. Moreover, the parallelism of Model Aggregator mapper affect the throughput and execution time of the classification algorithm.

6.3 Experimental Results

In this section the results of the experimental evaluation of the Flink's native VHT algorithm implementation against VHT implementation on Apache Samoa, but also against the RmVHT variation are presented and discussed. We first start comparing the performance of the algorithms with respect to their predictive error. Then, we also compare the average number of instances that each algorithm need in order to make a split decision.

6.3.1 Forest Covertype data set

Forest Covertype data set is the only real-world data set of this size, the characteristics of which are presented in Appendix Section A.1.1. In Figure 22, we see the evolution of the Prequential Classification Error of the native implementation of VHT algorithm against SAMOA's VHT implementation, as well as the RmVHT algorithm, as more instances are used to train and test the model. For the native implementation of VHT algorithm a data source with parallelism equal to one was used.
6 EXPERIMENTAL EVALUATION

Figure 22: Prequential classification error of Flink’s native VHT SAMOA’s VHT and RmVHT algorithm for UCI-Forest Covertype data set. Flink’s native VHT has data source with parallelism equal to 1.

As we see, Flink’s native VHT outperforms SAMOA’s VHT implementation with the error of Flink’s native VHT implementation to be around 9% lower at the end of the training.

Moreover, in Figure 23 we see that the average delay between node splits is by 3 seconds lower in Flink’s native VHT than SAMOA’s VHT. Furthermore, in Figure 24 we observe that the average number of instances seen in each leaf before a split in native VHT of Flink, is approximately 3 times the instances seen in SAMOA’s VHT.

Figure 23: Average delay in seconds between node splits of VHT classifier, for UCI-Forest Covertype data set on Apache Flink and Apache SAMOA.
Combining the above observations, we can conclude that the smaller delay between node splits in Flink’s native VHT implementation, results in a DT model which is more up-to-date than SAMOA’s VHT model. As a result, new incoming instances are classified with a more up-to-date or else more informed model, thus resulting in lower classification error.

In order to further understand and test the effect of higher throughput as well as the effect of the incoming order of the instances in the VHT learner, we run more experiments, where Flink’s native VHT implementation is fed by a data source of parallelism equal to 8 (instead of parallelism equal to 1 as in the previous experiments). In this case more input points are seen per second by the model, but not in the same order as before. As we see in Figure 25, in that case the error of Flink’s VHT does not evolve in the same way as when source and model parallelism are equal to 1. Thus the final error is almost the same as for the other models, but yet higher at the beginning of the algorithm.

Figure 24: Average number of instances seen in each leaf before it splits on Apache Flink and Apache SAMOA.
Regarding RmVHT, in Figure 22 we observe that the final prequential error of the RmVHT implementation is almost equal to SAMOA's VHT prequential error and worser than Flink's native VHT algorithm. Moreover, we see that in total RmVHT has a higher prequential classification error than the error of the centralized VHT algorithm. This is caused mainly due to the increased throughput of the RmVHT variation, where more instances are classified with a DT model which is not updated, thus more instances are misclassified. Another reason for the increased error of the RmVHT model is the incoming order of the data points, as it is known that in classification algorithms, the incoming order plays a significant role.

In order to understand why the RmVHT has higher error than the VHT algorithm we compared the average number of instances seen in each Global Model operator in RmVHT variation, presented in Figure 26. Combining those results with the prequential classification error of Figure 22, we understand that when the parallelism of the model increases, the throughput of the algorithm increases, which results in more instances "spent" in each of the leaves before they split, thus more instances are classified with an "expired" DT model.
Figure 26: Average number of instances seen in each leaf before it splits on Apache Flink VHT for different model parallelism.

6.3.2 HIGGS data set

The Higgs data set is a synthetic data set, a detailed description of which is presented in Appendix Section A.2.1. In general we observe that Higgs is not such a good data set to be used for classification with a DT classifier. As we see in Figure 27, SAMOA’s VHT learns slower than Flink’s native VHT but achieves lower prequential classification error at the end. On the other hand Flink’s VHT seems to learn faster at the beginning, but then its prequential classification error remains stable and slightly greater than SAMOA’s.

Figure 27: Prequential classification error of Flink’s native VHT, SAMOA’s VHT and RmVHT algorithm for UCI-HIGGS data set.
Also, in Figure 28 we observe that, for Higgs data set, the time delay between leaf node splits is slightly smaller for Flink’s native implementation than for SAMOA’s implementation, thus we do not observe such a big difference in the classification error of the two implementations.

![Figure 28: Average delay in seconds between node splits of VHT classifier, for UCI-HIGGS data set on Apache Flink and Apache SAMOA.](image)

Moreover, in Figure 29 we observe that in average the same number of instances is "seen" between node splits both in Flink’s native VHT as well as SAMOA’s VHT implementation. Lastly, as we see in Figure 30, for HIGGS data set, the average number of instances seen in each leaf node, before splitting, is almost the same regardless of the parallelism of the classification model, which also explains the only slight variations in the evolution of the prequential classification error in the RmVHT.
6 EXPERIMENTAL EVALUATION

Figure 29: Average number of instances seen in each leaf before it splits on Apache Flink and Apache SAMOA.

Figure 30: Average number of instances seen in each leaf before it splits on Apache Flink VHT for different model parallelism.

6.3.3 Waveform21 and Led data set

In this section we present the prequential classification error of the HT variations for the Waveform21 and Led synthetic data sets, a description of which can be found in Appendix Sections A.2.2 and A.2.3 correspondingly. No performance results were collected for those two data sets. In the experiments presented below, the source parallelism of Flink’s native VHT implementation is in all cases equal to 8, thus the order on input instances is not the same in Flink and SAMOA.
As we observe in Figure 31, for the Waveform21 data set SAMOA’s VHT outperforms Flink’s native VHT implementation. Moreover, we see that SAMOA’s VHT is learning slower, but achieves lower classification error at the end, whereas Flink’s native VHT learns faster, as it decreases very fast the classification error, but then its error remains stable.

Figure 31: Classification error of VHT and RmVHT classifier, for Waveform 21-attribute data set on Apache Flink and Apache SAMOA.

In Figure 32, we observe that for the Led data set Flink’s native VHT outperforms SAMOA’s VHT implementation but also that SAMOA’s implementation has a higher classification error than Flink’s RmVHT algorithm.

Figure 32: Classification error of VHT and RmVHT classifier, for Led data set on Apache Flink and Apache SAMOA.
6.4 Discussion

In general form our experimental results, we conclude that Flink’s native VHT implementation outperforms SAMOA’s VHT implementation for real-world data sets. Moreover, we experimentally show that the throughput as well as the order of the incoming data points does affect the performance of the classifier, as presented with the Forest Covertype data set.

For synthetic data sets, Flink’s native VHT performance varies, where in some cases it outperforms SAMOA’s VHT implementation with respect to lower prequential classification error. Specifically, we observe that Flink’s native VHT has a lower classification error for Led data set, whereas for Waveform21 and Higgs data set the error is slightly higher than SAMOA’s VHT.

Both Waveform21 and Higgs data sets share a common characteristic which could affect the performance of a machine learning algorithm, they have correlated features. Correlated features can affect seriously the performance of a classification algorithm, as they do not offer any new information to the Learner, thus Learner’s performance does not improve. Figures 31 and 32 are a nice example of the influence of data set characteristics to the model learned from a classification algorithm.
7 Conclusions and Future Work

In this master thesis we provide a machine learning pipeline model and build a prototype of our model on Apache Flink. Furthermore we used our prototype to implement two variations of Hoeffding Tree; Vertical Hoeffding Tree and Replicated model Vertical Hoeffding Tree, which we then compared with the original implementation of VHT on Apache SAMOA.

The simplicity and efficiency of the evaluation pipeline which we implemented to inspect the performance for the HT algorithm variations, proves that it can be a good baseline for implementing various parallel streaming ML algorithms.

From our experimental results we conclude that Flink’s native VHT implementation outperforms SAMOA’s VHT implementation for real-world data sets. However, our analysis showed also that depending on the data set and the correlation between the different features of the data set, the performance of each implementation, with respect to classification error, can have big variations. This suggests that in order to understand the performance or our algorithm we have to know the impact that the data set’s features will have in the learning of the classification model. Moreover, we experimentally showed that the throughput and the order of the input instances to a Learner plays an important role for the performance of a classification algorithm.

In future work it is important to identify how we could parallelize the evaluation step of the ML pipeline. The Evaluator is usually a bottleneck of a streaming ML pipeline, as an online distributed evaluation could not be accurate, since each evaluator would not able to have a spherical view of the misclassified instances, thus the evaluation is performed in a centralized mode.

Furthermore, we could design additional abstraction for our ML pipeline which would target in the automatic chaining of the Transformers, Learners and Evaluators provided by the user.

Moreover, a stress testing of Flink’s native VHT implementation should be performed with data sets that have nominal attributes, as in our experimental evaluation only data sets with numerical attributes were used. Lastly, the extension of Flink’s native VHT algorithm implementation in order to support, more split evaluation metrics, such Information Gain and Gini Impurity. Moreover, the implementation of a Bayesian predictor for assigning class labels to the leaf, except from the majority vote used in the existing algorithm implementation. Those extensions could be offered to the user as possible algorithm parameterization in order to fit the algorithm to user’s needs.
References


[27] “Apache SAMOA - Scalable Advanced Massive Online Analysis.”


[31] “Storm, distributed and fault-tolerant realtime computation.”


[33] “MOA Massive Online Analysis | Real Time Analytics for Data Streams.”


[39] “incubator-samoa Repository on github.”

Appendix
A Data Sets and Data Generators

The selection of the data sets used for the experiments is one a demanding process, as the researcher has to find data sets that would bring out both the good performance of an algorithm, but also potential vulnerabilities of the implementation, thus to give the opportunity to the developer to improve his/her implementation.

In scope of this master thesis, 5 different data sets were use, one real-world data set and 4 artificially generated data sets. The characteristics and the problem that each dataset describes are given in detail in the following sections.

The data sets and data generators used for the experiments were obtained by three sources:

1. UCI Machine Learning Repository [40]
2. MOA Generators [33]

A.1 Real Data Sets

A.1.1 Forest Covertype Data Set

The Forest Covertype data set was obtained from the UCI repository. Forest Covertype is a two class problem which has 581,012 distinct observations and it is formed from 54 attributes and it is the only real-world data set of that many observations.

A.2 Synthetically Generated Data Sets

A.2.1 HIGGS Data Set

HIGGS data set was obtained from UCI repository. HIGGS data set is a 2-class classification problem to distinguish between a signal process which produces Higgs bosons and a background process which does not. HIGGS has 11 million instances each of which has 28 numerical attributes. A bayesian error rate is not known for this data set.
A.2.2 Waveform-21 Data Set

Waveform-21 is a 3-class classification problem formed by 21 attributes, all of which are noisy. The MOA Waveform generator [33] was used for generating a dataset of 10 million observations. Each class of waveform is generated as a combination of 2 out of 3 classes of waveform. The Bayes error rate is known to be 14%.

A.2.3 LED Display Domain Data Set

The Led Display data generator was obtained from the UCI repository. Led data set has been used in numerous papers, few of which are experimenting with sequential implementations of HT algorithm. LED data set is a 10-class classification problem, formed by 7 Boolean attributes with 10% probability of noise for each attribute. The data generator was used in order to generate sample data three sets of size 10, 100 and 200 million data points. The data set of 10 million data points was used for the comparison the classification error for the two variations of the VHT implementation, whereas the larger data sets of 100 and 200 million data points were used for testing the throughput of the two variations, by comparing the difference in the execution times. The Bayes error for this data set is 26%.
B  UML Class Diagrams

Unified Modelling Language (UML) is a modeling language that is broadly used to visualize the design of a system. UML diagrams are used in this research project to help us present a blueprint of the system designed in scope of this master thesis.

B.1 Package Diagram

As depicted in the following package diagram 33, the `incrementalML` is the main package of the designed system, which contains all the other packages.

![Package Diagram for incremental ML scala library.](image)
The rationale behind the represented packages’ decomposition was the functionality that each package is enclosing:

- **changeDetector**: Contains the base trait and algorithms for change detection.

- **classification**: Contains the main class of VHT, as well as the two packages:
  - **attributeObserver**: Contains the base trait and classes for keeping important statistics for the observed attributes.
  - **metrics**: Contains the base trait and classes for representing the information exchanged between different nodes of the system.

- **common**: Contains the base traits which will be used for all ML algorithms and data preprocessing algorithms implementation.

- **preprocessing**: Will contain all the data preprocessing algorithms implementation. Currently contains only an *Imputer* data preprocessing algorithm.
Figure 34: Class diagram of classification package, containing VHT and DecisionTreeModel classes among others.
B.2.2 Metrics Class Diagram

In order to be able to send different type of messages between operators in Flink’s native VHT implementation, all messages should have the same supertyp. Thus, base trait Metrics was extended form all classes that represent a message object between operators, as depicted in the class diagram in Figure 35.

The messages send between operators in Flink’s native VHT implementation:

- **CalculateMetricsSignal**: The signal send from the Global Model operator to the PartialMetricsMapper operators, in order to calculate the two local best attributes to split a leaf node
- **DataPoints**: The type of the input instances to the VHT Learner
- **EvaluationMetric**: The type of the message containing the two best local attributes to split a leaf node
- **InstanceClassification**: The message send from the VHT Learner to the evaluator
- **VHTAttributes**: The object that contains any of the attributes of an input instance

![Figure 35: Metrics package class diagram.](image)