Distributed Decision Tree Learning for Mining Big Data Streams

Arinto Murdopo

Master of Science Thesis
European Master in Distributed Computing

Supervisors:
Albert Bifet
Gianmarco De Francisci Morales
Ricard Gavaldà

July 2013

Acknowledgements

First or foremost, I deeply thankful to Gianmarco De Francisci Morales and Albert Bifet, my industrial supervisors, who have provided me with continuous feedback and encouragement throughout the project. I also would like to thank Ricard Gavaldá, my academic supervisor, who is always available for discussion and consultation, via email, face-to-face or Skype.

Next, big thanks to Antonio Loureiro Severien, my project partner, for the camaraderie in developing and hacking of SAMOA, and also during our philosophical lunches.

I would like also to express my sincere gratitude to all colleagues, master thesis interns, PhD interns and PhD students at Yahoo! Lab Barcelona, especially Nicolas Kourtellis and Matthieu Morel for all the advices and inspirations in developing SAMOA, Çiğdem Aslay for all the consultation sessions about piled-higher-and-deeper, Martin Saveski for all the pizzas and fussballs when we were pulling the all-nighters, and my office neighbour Jose Moreno for all the non-sense humor.

Finally, another big thanks to all my EMDC classmates especially Màrio, Ioanna, Maria, Manos, Ümit, Zafar, Anis, Aras, Ziwei, and Hui, and also our seniors for this ultra-awesome journey in EMDC. Thank you!

Barcelona, July 2013
Arinto Murdopo

To my mother, for her
continuous support throughout
this master program.

Abstract

Web companies need to effectively analyse big data in order to enhance the experiences of their users. They need to have systems that are capable of handling big data in term of three dimensions: volume as data keeps growing, variety as the type of data is diverse, and velocity as the is continuously arriving very fast into the systems. However, most of the existing systems have addressed at most only two out of the three dimensions such as Mahout, a distributed machine learning framework that addresses the volume and variety dimensions, and Massive Online Analysis(MOA), a streaming machine learning framework that handles the variety and velocity dimensions.

In this thesis, we propose and develop Scalable Advanced Massive Online Analysis (SAMOA), a distributed streaming machine learning framework to address the aforementioned challenge. SAMOA provides flexible application programming interfaces (APIs) to allow rapid development of new ML algorithms for dealing with variety. Moreover, we integrate SAMOA with Storm, a state-of-the-art stream processing engine(SPE), which allows SAMOA to inherit Storm's scalability to address velocity and volume. The main benefits of SAMOA are: it provides flexibility in developing new ML algorithms and extensibility in integrating new SPEs. We develop a distributed online classification algorithm on top of SAMOA to verify the aforementioned features of SAMOA. The evaluation results show that the distributed algorithm is suitable for high number of attributes settings.

Contents

1 Introduction **1**

1.1 Contributions 2

1.2 Structure of the Thesis 2

2 Background and Related Works **5**

2.1 Distributed Machine Learning 6

2.1.1 Distributed ML Frameworks 6

2.1.2 Distributed Decision Tree 7

2.2 Streaming Machine Learning 8

2.2.1 Streaming ML Frameworks 8

2.2.2 Streaming Decision Tree 10

2.3 Distributed Streaming Machine Learning 10

3 Decision Tree Induction **13**

3.1 Basic Algorithm 14

3.1.1 Sample Dataset 14

3.1.2 Information Gain 15

3.1.3 Gain Ratio 17

3.2 Additional Techniques in Decision Tree Induction 18

3.3 Very Fast Decision Tree (VFDT) Induction 19

3.4 Extensions of VFDT 21

4 Distributed Streaming Decision Tree Induction **23**

4.1 Parallelism Type 23

4.1.1 Horizontal Parallelism 23

4.1.2 Vertical Parallelism 24

4.1.3 Task Parallelism . . . 26
4.2 Proposed Algorithm . . . 27
4.2.1 Chosen Parallelism Type . . . 27
4.2.2 Vertical Hoeffding Tree . . . 29

5 Scalable Advanced Massive Online Analysis 33
5.1 High Level Architecture . . . 33
5.2 SAMOA Modular Components . . . 34
5.2.1 Processing Item and Processor . . . 35
5.2.2 Stream and Content Event . . . 35
5.2.3 Topology and Task . . . 36
5.2.4 ML-adapter Layer . . . 37
5.2.5 Putting All the Components Together . . . 37
5.3 SPE-adapter Layer . . . 38
5.4 Storm Integration to SAMOA . . . 39
5.4.1 Overview of Storm . . . 39
5.4.2 Proposed Design . . . 41

6 Evaluation 43
6.1 Cluster Configuration . . . 43
6.2 Test Data . . . 44
6.3 VHT Implementations . . . 45
6.4 Results . . . 46
6.4.1 Accuracy Results . . . 47
6.4.2 VHT Throughput Results . . . 51
6.5 Discussion . . . 54

7 Conclusions 57
7.1 Open Issues . . . 57
7.2 Future Work . . . 58

References 61

List of Figures

3.1 Components of Decision Tree 13
3.2 Possible Splits for the Root in Weather Dataset 16
3.3 Possible Split for *ID code* Attribute 17

4.1 Horizontal Parallelism 24
4.2 Vertical Parallelism 25
4.3 Task Parallelism 26
4.4 Sorter PI in terms of Decision Tree Model 27
4.5 Decision Tree Induction based on Task Parallelism 28
4.6 Vertical Hoeffding Tree 29

5.1 SAMOA High Level Architecture 34
5.2 Parallelism Hint in SAMOA 35
5.3 Instatiation of a Stream and Examples of Groupings in SAMOA 36
5.4 Simplified Class Diagram of SPE-adapter Layer 38
5.5 Example of Storm Topology 40
5.6 Storm Cluster 40
5.7 samoa-storm Class Diagram 42

6.1 model-aggregator PI in VHT1 45
6.2 Accuracy of VHT2 and MHT for tree-10 and tree-100 generator 48
6.3 Number of leaf nodes of VHT2 and MHT for tree-10 and tree-100 generator 49
6.4 Accuracy of VHT2 and MHT for text-10, text-100 and text-1000 generator 50
6.5 Throughput of VHT2 and MHT for Tree Generator 51
6.6 Throughput of VHT2 and MHT for Text Generator 52
6.7 Profiling of VHT2 and MHT for text-1000 and text-10000 settings 53
6.8 Throughput of VHT2 and MHT for high number of attributes settings . . 54

List of Tables

3.1 Weather Dataset 15

6.1 Sample Table Used by Text Generator 44

1 Introduction

Web companies, such as Yahoo!, need to obtain useful information from big data streams, i.e. large-scale data analysis task in real-time. A concrete example of big data stream mining is Tumblr spam detection to enhance the user experience in Tumblr. Tumblr is a microblogging platform and social networking website. In terms of technique, Tumblr could use machine learning to classify whether a comment is a spam or not spam. And in terms of data volume, Tumblr users generate approximately three terabytes of new data everyday[1]. Unfortunately, performing real-time analysis task on big data is not easy. Web companies need to use systems that are able to efficiently analyze newly incoming data with acceptable processing time and limited memory. These systems should be able to properly address the three big data dimensions: volume, velocity and variety [1].

In term of volume, the systems need to handle an ever increasing amount of data. Distributed storage systems come in the picture to solve this challenge. Prime examples are Google File System (GFS) [2] and MapReduce [3]. GFS stores data in petabyte scale and it horizontally scale to handle ever increasing amount of data. MapReduce simplifies parallel programming abstraction in distributed environment. Moving forward, Hadoop[2] was developed as the open-source version of GFS and MapReduce. Other than those, there are other distributed systems to handle high volume of data such as Apache Cassandra[3], and Amazon's Dynamo [4].

On the subject of velocity, the systems need to cope with the high data arrival rate. This characteristic prompts the inception of traditional stream processing engine (SPE) such as Aurora [5], Stream [6] and Borealis [7]. Currently, the state-of-the-art SPEs are Apache S4 [8] and Storm[4]. Both of them follow SPEs characteristic, i.e. processing the records one by one and any aggregation is left to user. Similar to MapReduce, they view the data as sequences of records that are processed based on their keys.

With regards to variety, the ubiquity of smart-phones, mobile applications and internet produces various types of unstructured data such as emails, tweets, tumbles, and Facebook statuses. The web companies need new machine learning techniques to cope with ever increasing variety of the data in order to produce insightful analysis. The systems that address this dimension are machine learning frameworks such as WEKA [9].

Other than the aforementioned systems, there are systems that address more than one

[1] http://highscalability.com/blog/2013/5/20/the-tumblr-architecture-yahoo-bought-for-a-cool-billion-doll.html

[2] http://hadoop.apache.org/

[3] http://cassandra.apache.org/

[4] http://storm-project.net/

dimension. Distributed machine learning frameworks, such as Mahout[5] and MLBase [10], address volume and variety dimensions. Mahout is a collection of machine learning algorithms and they are implemented on top of Hadoop. MLBase aims to ease users in applying machine learning on top of distributed computing platform. Streaming machine learning frameworks, such as Massive Online Analysis(MOA)[6] and Vowpal Wabbit[7], address velocity and variety dimensions. Both frameworks contain algorithm that suitable for streaming setting and they allow the development of new machine learning algorithm on top of them.

However, few solutions have addressed all the three big data dimensions to perform big data stream mining. Most of the current solutions and frameworks only address at most two out of the three big data dimensions. The existence of solutions that address all big data dimensions allows the web-companies to satisfy their needs in big data stream mining.

1.1 Contributions

In this thesis, we present our work in developing a solution to address all of the three big data dimensions, focusing on the classification task. We study existing machine learning frameworks and learn their characteristics. Moreover, we study existing algorithms for distributed classification and streaming classification. We present Scalable Advanced Massive Online Analysis (SAMOA), which is a distributed machine learning framework for big data streams. SAMOA eases the development of new distributed algorithms for big data streams. To achieve scalability, this framework executes on top of existing SPEs. In this thesis, we integrate Storm into SAMOA. Moreover, this framework provides extensibility in term of underlying SPEs i.e. SAMOA users can easily add new SPEs and choose the underlying SPE to for SAMOA execution. We also implement a distributed online decision tree induction on top of SAMOA. The preliminary results show that our implementation has comparable performance with existing non-serialized implementation. With our implementation, we show that SAMOA are able to support rapid development and evaluation of distributed machine learning algorithms for big data streams.

1.2 Structure of the Thesis

The rest of this thesis is organized as follows. Section 2 provides background and related works in machine learning frameworks and decision tree induction algorithms. Section 3 explains basic decision tree induction as the basis of the work in this thesis. We present our implementation of a distributed streaming decision tree induction algorithm in section 4. Next, section 5 presents Scalable Advanced Massive Online Analysis (SAMOA)

[5]http://mahout.apache.org/

[6]http://moa.cms.waikato.ac.nz/

[7]http://github.com/JohnLangford/vowpal_wabbit/wiki

framework that is implemented as the main part of this thesis. Section 6 presents preliminary evaluation of the implemented algorithm on top of SAMOA. Finally, we present our conclusion and future works in section 7.

2 Background and Related Works

This chapter presents the existing machine learning frameworks that handles massive amount of data. Also, it presents the related works on decision tree learning, the basis of our work in this thesis.

Machine learning (ML) is a branch of artificial intelligence which is related to the development of systems that can learn and adapt based on data. These systems generally focus on building prediction models based on the data. To speed-up the analysis using ML, researchers develop software applications as ML frameworks. One prominent example is the WEKA framework [9] from the University of Waikato, New Zealand. WEKA contains a comprehensive collection of algorithms for ML and tools for data preprocessing. Furthermore, WEKA also includes several graphical user interfaces to improve its underlying functionality. It allows researchers to perform rapid prototyping of existing and new algorithms. The important characteristics of this type of frameworks and algorithms are: they run sequentially in single machine, they require all data to be fit in the memory and they require multiple pass into the dataset during learning phase. In this thesis, we refer the early generation of ML frameworks as *conventional frameworks*.

Classification is a supervised ML task which builds a model from labeled training data. The model is used for determining the class or label of testing data. We can measure the classification performance by using several metrics, such as accuracy, precision, and recall, on the testing data.

There are many types of classification algorithms such as tree-based algorithms (C4.5 decision tree, bagging and boosting decision tree, decision stump, boosted stump, and random forest), neural-network, Support Vector Machine (SVM), rule-based algorithms(conjunctive rule, RIPPER, PART, and PRISM), naive Bayes, logistic regression and many more. These classification algorithms have their own advantages and disadvantages, depending on many factors such as the characteristics of the data and results [11,12].

Decision tree induction is an easily interpretable type of tree-based classification algorithms. This algorithm employs a divide-and-conquer approach in building a classifier model that resembles a tree. This type of algorithm is often referred as *learner*. The most common strategy in decision tree induction is top-down induction of decision trees which grows the tree from the root (top) towards the leaves (bottom) [13]. Another example of the strategy in decision tree induction is bottom-up induction as presented in [14]. Once the model is available, the classifier uses the model to predict the class value of newly arriving datum.

2.1 Distributed Machine Learning

The abundance of data and the need to efficiently process them have triggered research and development of the conventional frameworks. As a result, researchers develop distributed and parallel ML frameworks. This type of framework should capable of handling large amount of data, in the magnitude of terabytes or petabytes.

2.1.1 Distributed ML Frameworks

Mahout[1] is a distributed ML framework on top of Hadoop. It contains common algorithms for classification, clustering, recommendation, and pattern mining. The main goal of Mahout is to provide scalability when analyzing large amount of data, i.e. more processing power can be easily added to cope with additional data.

Pregel [15] is a distributed graph computation abstraction based on for developing ML algorithms. It targets ML algorithms that utilize many *asynchronous, dynamic, graph-parallel* computation such as alternating least squares for movie recommendation, video co-segmentation and named-entity-recognition. It utilizes Bulk Synchronous Parallel (BSP) computation model using message-passing in processing the graph. The open source community developed Giraph[2], an implementation of Pregel that allows wider adoption of the BSP programming paradigm.

Distributed GraphLab [16] is a distributed implementation of GraphLab [17], a programming model based on shared-memory model. It serves the same purpose as Pregel, but it uses shared-memory model instead of message-passing model. Notable features of Distributed GraphLab framework are two implementations of GraphLab engines: a chromatic engine for partial synchronous with the assumption of graph-coloring existence, and a locking engine for fully asynchronous case. This framework includes fault tolerance by introducing two snapshot algorithms: a synchronous algorithm, and a fully asynchronous algorithm based on Chandy-Lamport snapshot. Furthermore, this framework introduces distributed-data-graph format to allow efficient graph processing in distributed environment.

MLBase [10] aims to ease non-experts access to ML. It simplifies ML utilization by transforming user-friendly ML declarative plan into refined learning plan i.e. MLBase returns the current best solution for the plan, and performs optimization in the background. To support this use case, MLBase masks the underlying complexity of distributed computation by providing a set of high-level operators for implement a scalable ML algorithm. The MLBase developers implement its distributed execution engine on top of Spark [18] and Mesos [19]. MLBase is part of Berkeley Data Analytics Stack[3].

There are some works that utilize MapReduce for distributed ML implementation.In [20] the authors analyze the taxonomy of ML learning algorithms and classify them based

[1] http://mahout.apache.org/
[2] http://giraph.apache.org
[3] http://amplab.cs.berkeley.edu/bdas/

on their data processing patterns. They divide the algorithms into three categories: single-pass learning, iterative, and query-based learning. Furthermore, they identify issues related to MapReduce implementation for each category. The main issue related to MapReduce implementation for ML is the high cost of distributing data into Hadoop clusters. The authors also propose some recommendations in tailoring MapReduce for ML: to provide map tasks with a shared space on a name node, to enable MapReduce input reader tasks to concurrently process multiple parallel files, and to provide proper static type checking in Hadoop. This work shows that distributed and scalable ML is feasible, but not all ML algorithms can efficiently and effectively be implemented in the distributed settings.

2.1.2 Distributed Decision Tree

There are some efforts to improve the performance of decision tree induction when processing high amounts of data, such as by parallelizing the induction process and by performing the induction process in distributed environment. PLANET [21] is a framework for learning tree models by using large datasets. This framework utilizes MapReduce to provide scalability. The authors propose PLANET scheduler to transform steps in decision tree induction into MapReduce jobs that can be executed in parallel. In term of parallelism, PLANET scheduler implements task-parallelism where each task (node splitting) is mapped into one MapReduce jobs that run concurrently. The authors introduce forward-scheduling mechanism to reduce the high setup and tear-down cost of MapReduce jobs. In order to reduce the tear-down cost, the forward-scheduling mechanism pools MapReduce output files and processes them immediately without waiting for job completion. Furthermore, the forward-scheduling mechanism utilizes background threads that continuously set up MapReduce jobs to reduce high setup cost of MapReduce jobs, forward-scheduling utilizes background thread to. These freshly-setup MapReduce jobs will wait for new node-splitting task from Planet scheduler.

Ye et. al. [22] propose techniques to distribute and parallelize Gradient Boosted Decision Trees(GBDT). The authors first implement MapReduce-based GBDT that employs horizontal data partitioning. Converting GBDT to MapReduce model is fairly straightforward. However, due to high overhead from HDFS as communication medium when splitting node, the authors conclude that Hadoop is not suitable for this kind of algorithm. Furthermore, the authors implement the GBDT on top of MPI and Hadoop-streaming. This implementation utilizes vertical data partitioning by splitting the data based on their attributes' value. This partitioning technique minimizes inter-machine communication cost. The authors show that the second implementation is able when presented with large datasets and it executes faster than the sequential version does.

2.2 Streaming Machine Learning

Streaming ML paradigm has emerged to address massive amounts of data. This paradigm is characterized by:

1. High data rate and volume, such as social media data, transactions logs in ATM or credit card operations and call logs in telecommunication company.

2. Unbounded, which means the data stream can potentially be infinite. This implies, multiple passes of analysis can not be performed due to inability of memory or disk to hold an infinite amount of data. The streaming ML algorithm needs to only analyze the data once and within small bounded time.

Another aspect of streaming ML is change detection. This aspect concerns the ability of the streaming ML algorithms to detect changes in incoming data streams. Once, the algorithms detect the changes, they need to update the induced model. Streaming ML algorithms are often referred to as *online algorithms* by ML researchers.

In term of implementations, a streaming ML algorithm should adhere to several requirements below:

1. The algorithm processes one arriving datum at a time and inspects it at most once. The algorithm may store some data for further processing but at some point it needs to discard the stored data to adhere the second requirement below.

2. The algorithm uses a limited amount of memory, since the main motivation of streaming ML is to process unbounded data that do not fit in memory.

3. The algorithm uses a limited amount of time to process one datum.

4. The algorithm is ready to predict at any point in time. This means the algorithm should manipulate the model in memory as it processes the newly arriving training data. When testing data arrive, the model should be able to perform prediction based on previously processed training data.

2.2.1 Streaming ML Frameworks

One of the existing streaming ML frameworks is Massive Online Analysis (MOA) [23]. MOA consists of well-known online algorithms for streaming classification, clustering, and change detection mechanisms. It includes also several variants of prequential evaluation for classification, hold-out procedure evaluation for classification and clustering evaluation. Furthermore, MOA provides extension points for ML developers to implement their own streaming source generators, learning algorithms and evaluation methods. It contains also visualization tools for clustering. However, MOA executes the algorithms sequentially in single machine which implies limited scalability when it faces ever increasing data stream.

Vowpal Wabbit[4] is another example of streaming ML framework based on the perceptron algorithm. The authors optimize Vowpal Wabbit for text data input. Vowpal Wabbit consists of several online algorithms such as stochastic gradient descent, truncated gradient for sparse online learning, and adaptive sub-gradient methods. Moreover, Vowpal Wabbit provides algorithms for parallel linear learning in the single-node settings.

Debellor [24] is an open source stream-oriented platform for machine learning. It utilizes component-based architecture, i.e. Debellor consists of a network of components, and each component communicates to another component via streamed data. This architecture allows ML developers to easily extend the framework with their own implementation of the components. The authors claims Debellor to be "scalable" in term of the input size i.e. Debellor is able to cope with the increase in input size which could not be handled by the corresponding batch-variant algorithm implementation. This means Debellor only supports vertical scalability. However, it does not support horizontal scalability since it is only support the single-node settings.

Another direction in streaming ML is to utilize existing stream processing engines (SPE) in the implementation of online algorithms. A SPE is a computation engine that is specialized to process high volume of incoming data stream with low latency. Stonebraker [25] defines a SPE as a system that applies SQL-style on the data stream on the fly, but this system does not store the data. There are open source SPEs such as Aurora [5]. And there are also commercial SPEs such as IBM InfoSphere Streams[5], Microsoft StreamInsight[6], and StreamBase[7]. SPEs are evolving and the state-of-the art SPEs utilize MapReduce-like programming model instead of SQL-style programming model to process incoming data stream. Prominent examples of the state-of-the-art SPEs are S4 [8] and Storm[8].

Storm.pattern project[9] is an effort to adapt existing ML model into Storm using Trident abstraction. In the current version, *Storm.pattern* does not allow online learning and it only allows online scoring using the imported Predictive Model Markup Language(PMML)[10] model. *Storm.pattern* supports PMML model for these following algorithms: random forest, linear regression, hierarchical clustering, and logistic regression. Moving forward, *Storm.pattern* developers plan to add online Rprop into storm-pattern so that it allows online learning and scoring.

Trident-ML project[11] is a real-time online ML library built on top of Storm. Similar to *Storm.pattern*, Trident-ML also utilizes Trident abstraction in its implementation. It supports online ML algorithms such as perceptron, passive-aggressive, Winnow, AROW and k-means.

[4]http://github.com/JohnLangford/vowpal_wabbit/wiki

[5]http://www.ibm.com/software/data/infosphere/streams

[6]http://www.microsoft.com/en-us/sqlserver/solutions-technologies/business-intelligence/streaming-data.aspx

[7]http://www.streambase.com

[8]http://storm-project.net/

[9]http://github.com/quintona/stormpattern

[10]http://en.wikipedia.org/wiki/Predictive_Model_Markup_Language

[11]http://github.com/pmerienne/tridentml

2.2.2 Streaming Decision Tree

One of the important works in adapting decision tree induction into streaming setting is Very Fast Decision Tree implementation (VFDT) [26]. This work focuses on alleviating the bottleneck of machine learning application in term of time and memory, i.e. the conventional algorithm are not able to process it due to limited processing time and memory. The main contribution from this work is the usage of the Hoeffding Bound to decide the minimum number of data to achieve certain level of confidence. This confidence determines how close the statistics between the attribute chosen by VFDT and the attribute chosen by the conventional algorithm. Section 3.3 discusses more about the streaming decision tree induction since it is one of the fundamental components in this thesis.

2.3 Distributed Streaming Machine Learning

Existing streaming machine learning frameworks, such as MOA, are unable to scale when handling very high incoming data rate. In order to solve this scalability issue, there are several possible solutions such as to make the streaming machine learning framework distributed and to make the online algorithms run in parallel.

In term of frameworks, we identify two frameworks that belong to category of distributed streaming machine learning: Jubatus and Stormmoa. Jubatus[12] is an example of distributed streaming machine learning framework. It includes a library for streaming machine learning such as regression, classification, recommendation, anomaly detection and graph mining. It introduces local ML model concept which means there can be multiple models running at the same time and they process different sets of data. Using this technique, Jubatus achieves horizontal scalability via horizontal parallelism in partitioning data. Jubatus adapts MapReduce operation into three fundamental concepts, which are:

1. *Update*, means process a datum by updating the local model.
2. *Analyze*, means process a datum by applying the local model to it and get the result, such as the predicted class value in classification.
3. *Mix*, merge local model into a mixed model which will be used to update all local models Jubatus.

Jubatus establishes tight coupling between the machine learning library implementation and the underlying distributed SPE. The reason is Jubatus' developers build and implement their own custom distributed SPEs. Jubatus achieves fault tolerance by using *jubakeeper* component which utilizes ZooKeeper.

[12]http://jubat.us/en/

StormMOA[13] is a project to combine MOA with Storm to satisfy the need of scalable implementation of streaming ML frameworks. It uses Storm's Trident abstraction and MOA library to implement OzaBag and OzaBoost [27]. The StormMOA developers implement two variants of the aforementioned ML algorithms, they are

1. In-memory-implementation, where StormMOA stores the model only in memory and it does not persist the model into a storage. This implementation needs to rebuild the model from the beginning when failure happens.

2. Implementation with fault tolerance, StormMOA persists model into hence it is able to recover the model upon failure.

Similar to Jubatus, StormMOA also establishes tight coupling between MOA (the machine learning library) and Storm (the underlying distributed SPE). This coupling prevents StormMOA's extension by using other SPEs to execute the machine learning library.

In term of algorithm, there are few efforts in parallelizing streaming decision tree induction. In [28], the authors propose approximate algorithm for streaming data to build the decision tree. This algorithm uses bread-first-mode of decision tree induction using horizontal parallelism. The authors implement the algorithm on top of distributed architecture which consists of several processors. Each of the processor has complete view of the built decision tree so far, i.e. it is a shared memory between processor. The implementation utilizes master-slaves technique to coordinate the parallel computations. Each slave calculates local statistics and periodically updates the master with the statistics. The master aggregates the statistics and updates the decision tree based on the aggregated statistics.

[13]http://github.com/vpa1977/stormmoa

3 Decision Tree Induction

This chapter presents the existing algorithms that are used in our work. Section 3.1 presents the basic decision tree induction. The following section, section 3.2, discusses the necessary additional techniques to make the basic algorithm ready for production environments. Section 3.3 presents the corresponding algorithm for streaming settings. In addition, section 3.4 explains some extensions for the streaming decision tree induction algorithm.

A decision tree consists of *nodes*, *branches* and *leaves* as shown in Figure 3.1. A *node* consists of a question regarding a value of an attribute, for example node n in Figure 3.1 has a question: "is attr_one greater than 5?". We refer to this kind of node as *split-node*. *Branch* is a connection between nodes that is established based on the answer of its corresponding question. The aforementioned node n has two branches: "True" branch and "False" branch. Leaf is an end-point in the tree. The decision tree uses the *leaf* to predict the class of given data by utilizing several predictor functions such as majority-class, or naive Bayes classifier. Sections 3.1 to 3.2 discuss the basic tree-induction process in more details.

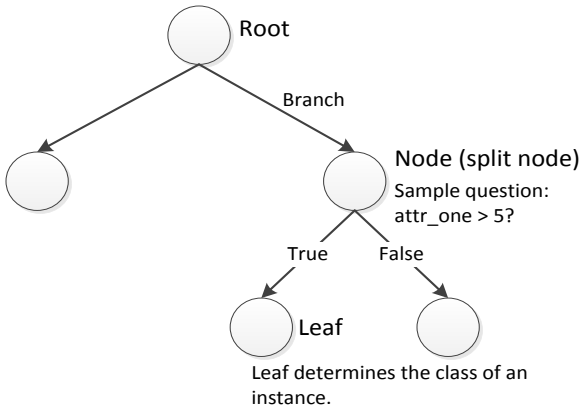


Figure 3.1: Components of Decision Tree

3.1 Basic Algorithm

Algorithm 3.1 shows the generic description of decision tree induction. The decision tree induction algorithm begins with an empty tree(line 1). Since this algorithm is a recursive algorithm, it needs to have the termination condition shown in line 2 to 4. If the algorithm does not terminate, it continues by inducing a root node that considers the entire dataset for growing the tree. For the root node, the algorithm processes each datum in the dataset D by iterating over all available attributes (line 5 to 7) and choosing the attribute with best information-theoretic criteria (a_{best} in line 8). Section 3.1.2 and 3.1.3 further explain the calculation of the criteria. After the algorithms decides on a_{best}, it creates a split-node that uses a_{best} to test each datum in the dataset (line 9). This testing process induces sub-dataset D_v (line 10). The algorithm continues by recursively processing each sub-dataset in D_v. This recursive call produces sub-tree $Tree_v$ (line 11 and 12). The algorithm then attaches $Tree_v$ into its corresponding branch in the corresponding split-node (line 13).

Algorithm 3.1 DecisionTreeInduction(D)

Input: D, which is attribute-valued dataset

```
1: Tree = {}
2: if D is "pure" OR we satisfy other stopping criteria then
3:    terminate
4: end if
5: for all attribute a ∈ D do
6:    Compare information-theoretic criteria if we split on a
7: end for
8: Obtain attribute with best information-theoretic criteria, a_best
9: Add a split-node that splits on a_best into
10: Induce sub-dataset of D based on a_best into D_v
11: for all D_v do
12:    Tree_v = DecisionTreeInduction(D_v)
13:    Attach Tree_v into corresponding branch in the split-node.
14: end for
15: return Tree
```

3.1.1 Sample Dataset

Given the weather dataset [29] in table 3.1, we want to predict whether we should play or not play outside. The attributes are *Outlook*, *Temperature*, *Humidity*, *Windy* and *ID code*. Each attribute has its own possible values, for example *Outlook* has three possible values: *sunny*, *overcast* and *rainy*. The output class is *Play* and it has two values: *yes* and *no*. In this example, the algorithm uses all the data for training and building the model.

ID code	Outlook	Temperature	Humidity	Windy	Play
a	sunny	hot	high	false	no
b	sunny	hot	high	true	no
c	overcast	hot	high	false	yes
d	rainy	mild	high	false	yes
e	rainy	cool	normal	false	yes
f	rainy	cool	normal	true	no
g	overcast	cool	normal	true	yes
h	sunny	mild	high	false	no
i	sunny	cool	normal	false	yes
j	rainy	mild	normal	false	yes
k	sunny	mild	normal	true	yes
l	overcast	mild	high	true	yes
m	overcast	hot	normal	false	yes
n	rainy	mild	high	true	no

Table 3.1: Weather Dataset

Ignoring the *ID code* attribute, we have four possible splits in growing the tree's root shown in Figure 3.2. Section 3.1.2 and 3.1.3 explain how the algorithm chooses the best attribute.

3.1.2 Information Gain

The algorithm needs to decide which split should be used to grow the tree. One option is to use the attribute with the highest *purity measure*. The algorithm measures the attribute purity in term of *information value*. To quantify this measure, the algorithm utilizes *entropy* formula. Given a random variable that takes c values with probabilities p_1, p_2, ... p_c, the algorithm calculates the information value with this following entropy formula:

$$\sum_{i=1}^{c} -p_i log_2 p_i$$

Refer to Figure 3.2, the algorithm derives the information values for *Outlook* attribute with following steps:

1. *Outlook* has three outcomes: sunny, overcast and rainy. The algorithm needs to calculate the information values for each outcome.

2. Outcome *sunny* has two occurrences of output class *yes*, and three occurrences of *no*. The information value of *sunny* is $info([2,3]) = -p_1 log_2 p_1 - p_2 log_2 p_2$ where p_1 is the probability of *yes* (with value of $\frac{2}{2+3} = \frac{2}{5}$) and p_2 is the probability of "no" in sunny outlook (with value of $\frac{3}{2+3} = \frac{3}{5}$). The algorithm uses both p values into the entropy

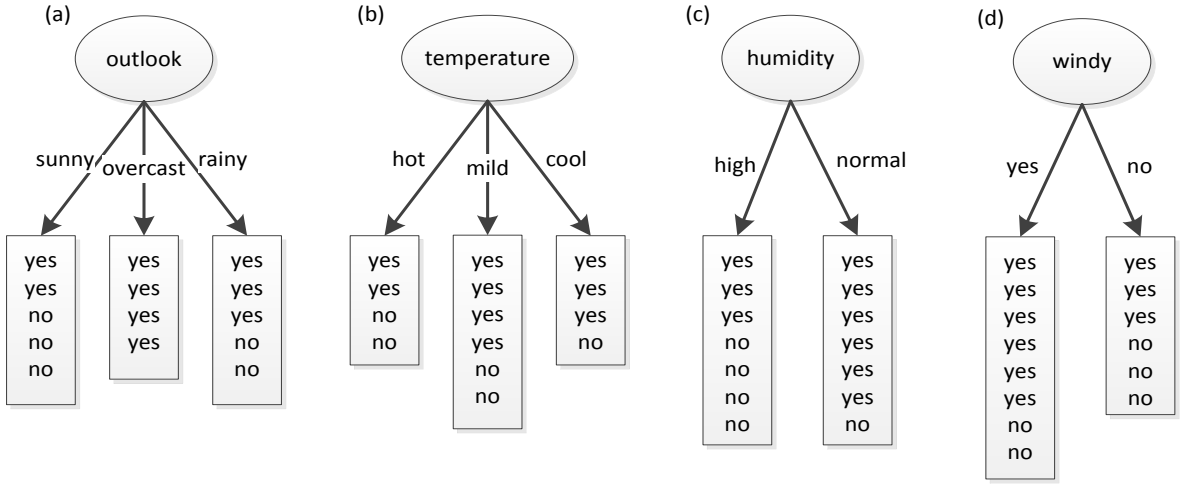


Figure 3.2: Possible Splits for the Root in Weather Dataset

formula to obtain *sunny*'s information value: $info([2,3]) = -\frac{2}{5}log_{\frac{2}{5}} - \frac{3}{5}log_2\frac{3}{5} = 0.971$ bits.

3. The algorithm repeats the calculation for other outcomes. Outcome *overcast* has four *yes* and zero *no*, hence its information value is: $info([4,0]) = 0.0$ bits. Outcome *rainy* has three *yes* and two *no*, hence its information values is: $info[(3,2)] = 0.971$ bits.

4. The next step for the algorithm is to calculate the expected amount of information when it chooses *Outlook* to split, by using this calculation: $info([2,3],[4,0],[3,2]) = \frac{5}{14}info([2,3]) + \frac{4}{14}info([4,0]) + \frac{5}{14}info([3,2]) = 0.693$ bits.

The next step is to calculate the information gain obtained by splitting on a specific attribute. The algorithm obtains the gain by subtracting the entropy of splitting on a specific attribute with the entropy of no-split case.

Continuing the *Outlook* attribute sample calculation, the algorithm calculates entropy for no-split case: $info([9,5]) = 0.940$ bits. Hence, the information gain for *Outlook* attribute is $gain(Outlook) = info([9,5]) - info([2,3],[4,0],[3,2]) = 0.247$ bits.

Refer to the sample case in Table 3.1 and Figure 3.2, the algorithm produces these following gains for each split:

- $gain(Outlook) = 0.247$ bits
- $gain(Temperature) = 0.029$ bits
- $gain(Humidity) = 0.152$ bits

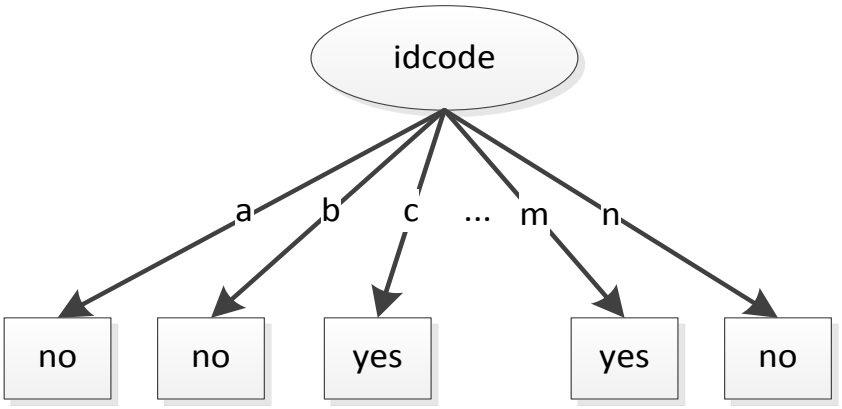


Figure 3.3: Possible Split for *ID code* Attribute

- $gain(Windy) = 0.048$ bits

Outlook has the highest information gain, hence the algorithm chooses attribute *Outlook* to grow the tree and split the node. The algorithm repeats this process until it satisfies one of the terminating conditions such as the node is pure (i.e the node only contains one type of class output value).

3.1.3 Gain Ratio

The algorithm utilizes *gain ratio* to reduce the tendency of information gain to choose attributes with higher number of branches. This tendency causes the model to overfit the training set, i.e. the model performs very well for the learning phase but it perform poorly in predicting the class of unknown instances. The following example discusses gain ratio calculation.

Refer to table 3.1, we include *ID code* attribute in our calculation. Therefore, the algorithm has an additional split possibility as shown in Figure 3.3.

This split will have entropy value of 0, hence the information gain is 0.940 bits. This information gain is higher than *Outlook*'s information gain and the algorithm will choose *ID code* to grow the tree. This choice causes the model to overfit the training dataset. To alleviate this problem, the algorithm utilizes *gain ratio*.

The algorithm calculates the gain ratio by including the number and size of the resulting daughter nodes but ignoring any information about the daughter nodes' class distribution. Refer to *ID code* attribute, the algorithm calculates the gain ratio with the following steps:

1. The algorithm calculates the information value for *ID code* attribute while ignoring the class distribution for the attribute: $info[(1,1,1,1....1)] = -\frac{1}{14}log_2\frac{1}{14} \times 14 = 3.807$ bits.

2. Next, it calculates the gain ratio by using this formula: $gain\_ratio(ID\,code) = \frac{gain(ID\,code)}{info[(1,1,1,1...1)]} = \frac{0.940}{3.807} = 0.247$.

For *Outlook* attribute, the corresponding gain ratio calculation is:

1. Refer to Figure 3.2, *Outlook* attribute has five class outputs in *sunny*, four class outputs in *overcast*, and five class outputs in *rainy*. Hence, the algorithm calculates the information value for *Outlook* attribute while ignoring the class distribution for the attribute: $info[(5,4,5)] = -\frac{5}{14}log_2\frac{5}{14} - \frac{4}{14}log_2\frac{4}{14} - \frac{5}{14}log_2\frac{5}{14} = 1.577$ bits.

2. $gain\_ratio(Outlook) = \frac{gain(Outlook)}{info[(5,4,5)]} = \frac{0.247}{1.577} = 0.157$.

The gain ratios for every attribute in this example are:

- $gain\_ratio(ID\,code) = 0.247$
- $gain\_ratio(Outlook) = 0.157$
- $gain\_ratio(Temperature) = 0.019$
- $gain\_ratio(Humidity) = 0.152$
- $gain\_ratio(Windy) = 0.049$

Based on above calculation, the algorithm still chooses *ID code* to split the node but the gain ratio reduces the *ID code*'s advantages to the other attributes. *Humidity* attribute is now very close to *Outlook* attribute because it splits the tree into less branches than *Outlook* attribute splits.

3.2 Additional Techniques in Decision Tree Induction

Unfortunately, information gain and gain ratio are not enough to build a decision tree that suits production settings. One example of well-known decision tree implementation is C4.5 [30]. This section explain further techniques in C4.5 to make the algorithm suitable for production settings.

Quinlan, the author of C4.5, proposes *tree pruning* technique to avoid overfitting by reducing the number of nodes in the tree. C4.5 commonly performs this technique in a single bottom-up pass after the tree is fully grown. Quinlan introduces *pessimistic pruning* that estimates the error rate of a node based on the estimated errors of its sub-branches. If the estimated error of a node is less than its sub-branches' error, then pessimistic pruning uses the node to replace its sub-branches.

The next technique is in term of continuous attributes handling. Continuous attributes require the algorithm to choose threshold values to determine the number of splits. To handle this requirement, C4.5 utilizes only the information gain technique in choosing the threshold. For choosing the attribute, C4.5 still uses the information gain and the gain ratio techniques altogether.

C4.5 also has several possibilities in handling missing attribute values during learning and testing the tree. There are three scenarios when C4.5 needs to handle the missing values properly, they are:

- When comparing attributes to split and some attributes have missing values.
- After C4.5 splits a node into several branches, a training datum with missing values arrives into the split node and the split node can not associate the datum with any of its branches.
- When C4.5 attempts to classify a testing datum with missing values but it can not associate the datum with any of the available branches in a split node.

Quinlan presents a coding scheme in [31] and [32] to deal with each of the aforementioned scenarios. Examples of the coding scheme are: (I) to ignore training instances with missing values and (C) to substitute the missing values with the most common value for nominal attributes or with the mean of the known values for the numeric attributes.

3.3 Very Fast Decision Tree (VFDT) Induction

We briefly described in section 2.2.2 Very Fast Decision Tree (VFDT) [26] which is the pioneer of streaming decision tree induction. VFDT fulfills the necessary requirements in handling data streams in an efficient way. The previous streaming decision tree algorithms that introduced before VFDT does not have this characteristic. VFDT is one of the fundamental concepts in our work, hence this section further discusses the algorithm.

This section refers the resulting model from VFDT as *Hoeffding tree* and the induction algorithm as *Hoeffding tree induction*. We refer to data as *instances*, hence we also refer datum as a single instance. Moreover, this section refers the whole implementation of VFDT as *VFDT*.

Algorithm 3.2 shows the generic description of Hoeffding tree induction. During the learning phase, VFDT starts Hoeffding tree with only a single node. For each training instance E that arrives into the tree, VFDT invokes Hoeffding tree induction algorithm. The algorithms starts by sorting the instance into a leaf l(line 1). This leaf is a *learning leaf*, therefore the algorithm needs to update the sufficient statistic in l(line 2). In this case, the sufficient statistic is the class distribution for each attribute value. The algorithms also increment the number of instances (n_l) seen at lea l based on E's weight (line 3). One instance is not significant enough to grow the tree, therefore the algorithm only grows

Algorithm 3.2 HoeffdingTreeInduction(E, HT)

Input: E is a training instance
Input: HT is the current state of the decision tree
1: Use HT to sort E into a leaf l
2: Update sufficient statistic in l
3: Increment the number of instances seen at l (which is n_l)
4: **if** n_l *mod* $n_{min} = 0$ **and** not all instances seen at l belong to the same class **then**
5: For each attribute, compute $\overline{G}_l(X_i)$
6: Find X_a, which is the attribute with highest $\overline{G}_l$
7: Find X_b, which is the attribute with second highest $\overline{G}_l$
8: Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
9: **if** $X_a \neq X_\emptyset$ **and** $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ **or** $\epsilon < \tau)$ **then**
10: Replace l with a split-node on X_a
11: **for all** branches of the split **do**
12: Add a new leaf with derived sufficient statistic from the split node
13: **end for**
14: **end if**
15: **end if**

the tree every certain number of instances (n_{min}). The algorithm does not grow the trees if all the data seen at l belong to the same class. Line 4 shows these two conditions to decide whether to grow or not to grow the tree.

In this algorithm, growing the tree means attempting to split the node. To perform the split, the algorithm iterates through each attribute and calculate the corresponding information-theoretic criteria($\overline{G}_l(X_i)$ in line 5). It also computes the information-theoretic criteria for no-split scenario ($X_\emptyset$). The authors of VFDT refers this inclusion of no-split scenario with term *pre-pruning*.

The algorithm then chooses the best(X_a) and the second best(X_b)) attributes based on the criteria (line 6 and 7). Using these chosen attributes, the algorithm computes the Hoeffding bound to determine whether it needs to split the node or not. Line 9 shows the complete condition to split the node. If the best attribute is the no-split scenario ($X_\emptyset$), then the algorithm does not perform the split. The algorithm uses tie-breaking τ mechanism to handle the case where the difference of information gain between X_a and X_b) is very small($\Delta\overline{G}_l < \epsilon < \tau$). If the algorithm splits the node, then it replaces the leaf l with a split node and it creates the corresponding leaves based on the best attribute (line 10 to 13).

To calculate Hoeffding bound, the algorithm uses these following parameters:

- r = real-valued random variable,with range R.
- n = number of independent observations have been made.
- $\bar{r}$ = mean value computed from n independent observations.

The Hoeffding bound determines that the true mean of the variable is at least $\bar{r} - \epsilon$ with probability $1 - \delta$. And, ϵ is calculated with this following formula:

$$\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n}}$$

What makes Hoeffding bound attractive is its ability to give the same results regardless the probability distribution generating the observations. This attractiveness comes with one drawback which is different number of observations to reach certain values of δ and ϵ depending on the probability distributions of the data.

VFDT has no termination condition since it is a streaming algorithm. The tree may grow infinitely and this contradicts one of the requirements for algorithm for streaming setting (require limited amount of memory). To satisfy the requirement of limited memory usage, the authors of VFDT introduce node-limiting technique. This technique calculates the *promise* for each active learning leaf l. A promise of an active learning leaf in VFDT is defined as an estimated upper-bound of the error reduction achieved by keeping the leaf active. Based on the promise, the algorithm may choose to deactivate leaves with low promise when the tree reaches the memory limit. Although the leaves are inactive, VFDT still monitors the promise for each inactive leaf. The reason is that VFDT may activate the inactive leaves when their promises are higher than currently active leaves' promises. Besides the node-limiting technique, the VFDT authors introduce also poor-attribute-removal technique to reduce VFDT memory usage. VFDT removes the attributes that does not look promising while splitting, hence the statistic associated with the removed attribute can be deleted from the memory.

3.4 Extensions of VFDT

Since the inception of VFDT, researchers have proposed many enhancements based on VFDT. Hulten et. al. [33] presents Concept-adaptive VFDT (CVFDT) which handles changes in the data stream input characteristics. CVFDT utilizes a sliding-window of data from data stream to monitor the changes and adjust the decision tree accordingly. To monitor the changes, CVFDT periodically determines the best splitting candidates at every previous splitting node. If one of the candidates is better than current attribute to split the node, then either the original result from the current attribute is incorrect or changes have occurred. When these cases happen, CVFDT adjusts the decision tree by deactivating some existing subtrees and growing new alternative subtrees.

Gama et. al. [34] propose VFDTc to handle continuous numeric attribute and to improve the accuracy of VFDT. VFDTc handles continuous numeric attribute by maintaining a binary tree for each observed continuous attribute. It represents split point as a node in the binary tree. A new split point is only added by VFDTc into the binary tree when the number of incoming data in each subsets is higher than a configurable constant. The binary tree allows efficient calculation of the merit in each split-point. Furthermore,

VFDTc improves the accuracy of VFDT by utilizing custom classifiers at VFDTc tree leaves. In this work, the authors use naive Bayes classifiers and majority class classifiers.

Jin and Agrawal [35] present an alternative in processing numerical attributes by using a numerical interval pruning (NIP) approach. This technique allows faster execution time. In addition, the authors also propose a method to reduce the required sample sizes to reach a given bound on the accuracy. This method utilizes the properties of gain function entropy and gini.

4 Distributed Streaming Decision Tree Induction

This chapter presents Vertical Hoeffding Tree (VHT), our effort in parallelizing streaming decision tree induction for distributed environment. Section 4.1 reviews the available types of parallelism and section 4.2 explains our proposed algorithm.

4.1 Parallelism Type

In this thesis, parallelism type refers to the way an algorithm performs parallelization in streaming decision tree induction. Understanding the available parallelism types is important since it is the basis of our proposed distributed algorithm. This section presents three types of parallelism.

For section 4.1.1 to 4.1.3, we need to define some terminologies in order to make the explanation concise and clear. A *processing item*(PI) is an unit of computation element that executes some part of the algorithm. One computation element can be a node, a thread or a process depending on the underlying distributed streaming computation platform. An *user* is a client who executes the algorithm. An user could be a human being or a software component that executes the algorithm such as a machine learning framework. We define an *instance* as a datum in the training data. Since the context is streaming decision tree induction, the training data consists of a set of instances that arrive one at a time to the algorithm.

4.1.1 Horizontal Parallelism

Figure 4.1 shows the implementation of horizontal parallelism for distributed streaming decision tree induction. Source processing item sends instances into the distributed algorithm. This component may comprise of an instance generator, or it simply forwards object from external source such as Twitter firehose. A model-aggregator PI consists of the trained model which is a global decision tree in our case. A local-statistic processing PI contains local decision tree. The distributed algorithm only trains the local decision trees with subset of all instances that arrive into the local-statistic PIs. The algorithm periodically aggregates local statistics (the statistics in local decision tree) into the statistics in global decision tree. User determines the interval period by setting a parameter in the distributed algorithm. After the aggregation finishes, the algorithm needs to update each decision tree in local-statistics PIs.

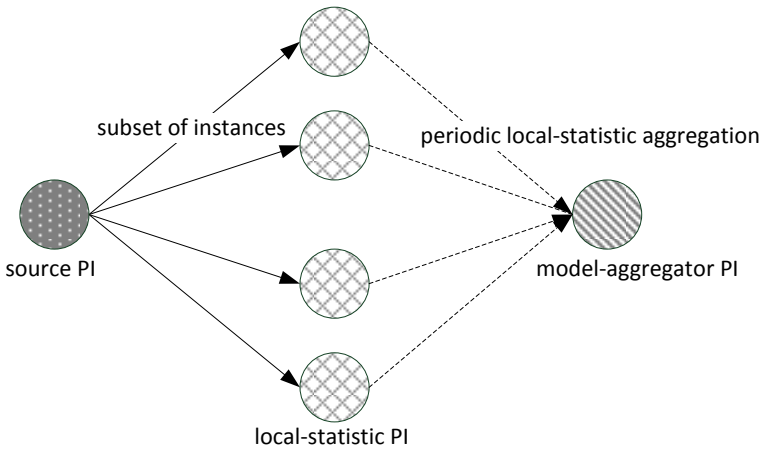


Figure 4.1: Horizontal Parallelism

In horizontal parallelism, the algorithm distributes the arriving instances to local-statistic PIs based on horizontal data partitioning, which means it partitions the arriving instances equally among the number of local-statistic PI. For example, if there are 100 arriving instances and there are 5 local-statistics PIs, then each local-statistic PI receives 20 instances. One way to achieve this is to distribute the arriving instances in round-robin manner. An user determines the parallelism level of the algorithm by setting the number of local-statistic PI to process the arriving instances. If arrival rate of the instances exceeds the total processing rate of the local-statistic PIs, then user should increase the parallelism level.

Horizontal parallelism has several advantages. It is appropriate for scenarios with very high arrival rates. The algorithm also observes the parallelism immediately. User is able to easily add more processing power by adding more PI to cope with arriving instances. However, horizontal parallelism needs high amount of available memory since the algorithm replicates the model in the local-statistic PIs. It is also not suitable for cases where the arriving instance has high number of attributes since the algorithm spends most of its time to calculate the information gain for each attribute. The algorithm introduces additional complexity in propagating the updates from global model into local model in order to keep the model consistency between each local-statistic PI and model-aggregator PI.

4.1.2 Vertical Parallelism

Figure 4.2 shows the implementation of vertical parallelism for distributed streaming decision tree induction. Source processing item serves the same purpose as the one in horizontal parallelism (section 4.1.1). However, model-aggregator PI and local-statistic PI have different roles compared to the ones in section 4.1.1.

In vertical parallelism, local-statistic PIs do not have the local model as in horizontal

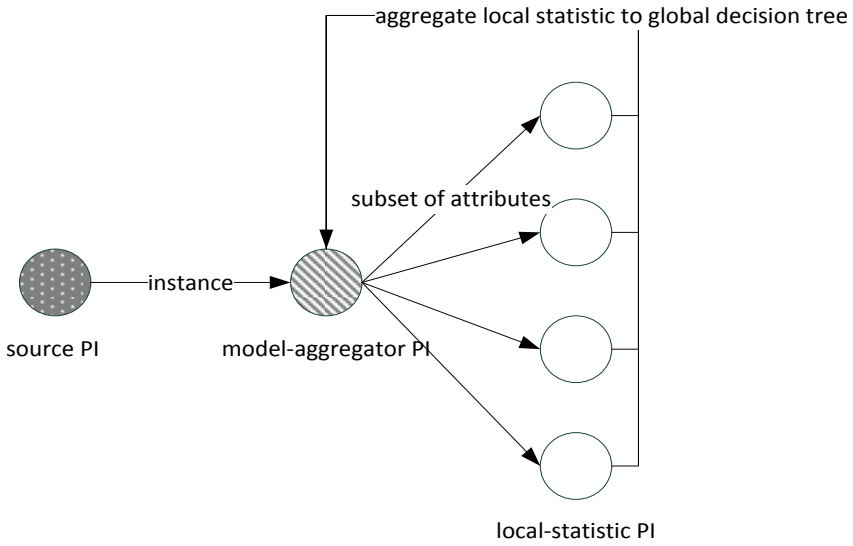


Figure 4.2: Vertical Parallelism

parallelism. Each local-statistic PI only stores the sufficient statistic of several attributes that are assigned to it and computes the information-theoretic criteria (such as the information gain and gain ratio) based on the assigned statistic. Model aggregator PI consists of a global model, but it distributes the instances by their attributes. For example if each instance has 100 attributes and there are 5 local-statistic PIs, then each local-statistic PI receives 20 attributes for each instance. An user determines the parallelism level of the algorithm by setting the number of local-statistic PI to process the arriving instances. However, increasing the parallelism level may not necessarily improve the performance since the cost of splitting and distributing the instance may exceed the benefit.

Vertical parallelism is suitable when arriving instances have high number of attributes. The reason is that vertical parallelism spends most of its time in calculating the information gain for each attribute. This type of instance is commonly found in text mining. Most text mining algorithms use a dictionary consists of 10000 to 50000 entries. The text mining algorithms then transform text into instances with the number of attributes equals to the number of the entries in the dictionary. Each word in the text correspond to a boolean attribute in the instances.

Another case where vertical parallelism suits is when the instances are in the form of *documents*. A document is almost similar to a row or a record in relational database system, but it is less rigid compared to row or record. A document is not required to comply with database schemas such as primary key and foreign key. Concrete example of a document is a tweet where each word in a tweet corresponds to one or more entries in a document. And similar to text mining, documents have a characteristic of high number attributes since practically documents are often implemented as dictionaries which have 10000 to 50000 entries. Since each entry corresponds to an attribute, the algorithm needs to process attributes in the magnitude of ten thousands.

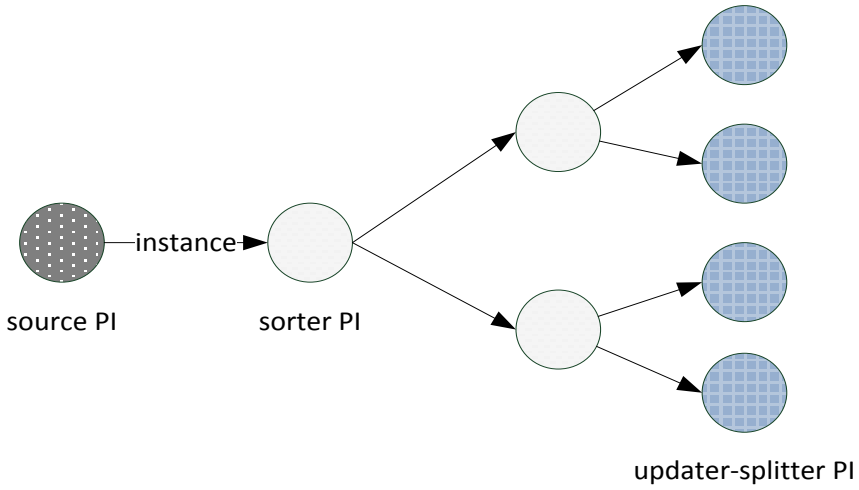


Figure 4.3: Task Parallelism

One advantage of vertical parallelism is the algorithm implementation uses lesser total memory compared to horizontal parallelism since it does not replicate the model into local-statistics PI. However, vertical parallelism is not suitable when the number of attributes in the attributes is not high enough so that the cost of splitting and distributing is higher than the benefit obtained by the parallelism.

4.1.3 Task Parallelism

Figure 4.3 shows the task parallelism implementation for distributed streaming decision tree induction. We define a *task* as a specific portion of the algorithm. In streaming decision tree induction, the task consists of: sort the arriving instance into correct leaf, update sufficient statistic, and attempt to split the node.

Task parallelism consists of sorter processing item and updater-splitter processing item. This parallelism distributes the model into the available processing items. Sorter PI consists of part of the decision tree which is a subtree and connection to another subtree as shown in figure 4.4. It sorts the arriving instance into correct leaf. If the leaf does not exist in the part that the sorter PI owns, the sorter PI forwards the instance into the correct sorter or the updater-splitter PI that contains the path to appropriate leaf. Updater-splitter PI consists of a subtree that has leaves. This PI updates sufficient statistic for the leaf and splits the leaf when the leaf satisfies splitting condition.

Figure 4.5 shows the induction based on task parallelism. This induction process starts with one updater-splitter PI in step (i). This PI grows the tree until the tree reaches memory limit. When this happens, the PI converts itself into sorter PI and it creates new updater-splitter PIs to represent subtrees generated from its leaves, as shown in step (ii) and (iii). The algorithm repeats the process until it uses all available processing items. User configures the number of PIs available for the algorithm.

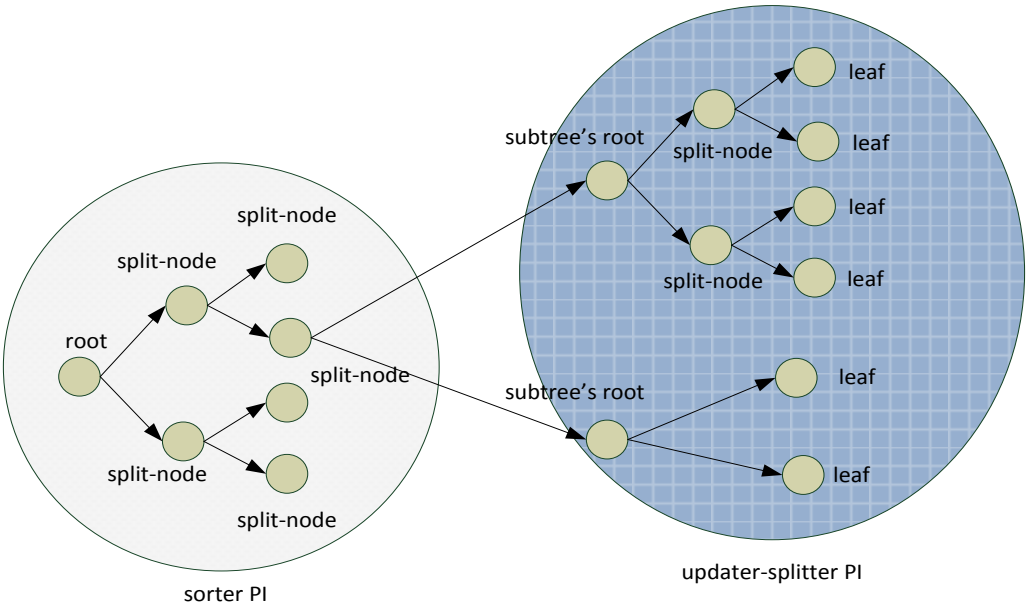


Figure 4.4: Sorter PI in terms of Decision Tree Model

Task parallelism is suitable when the model size is very high and the resulting model could not fit in the available memory. However, the algorithm does not observe the parallelism immediately. Only after the algorithm distributes the model, it can observe the parallelism.

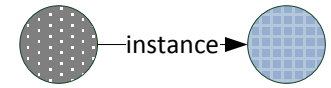
4.2 Proposed Algorithm

This section discusses our proposed algorithm for implementing distributed and parallel streaming decision tree induction. The algorithm extends VFDT algorithm presented in section 3.3 with capabilities of performing streaming decision tree induction in distributed and parallel manner.

4.2.1 Chosen Parallelism Type

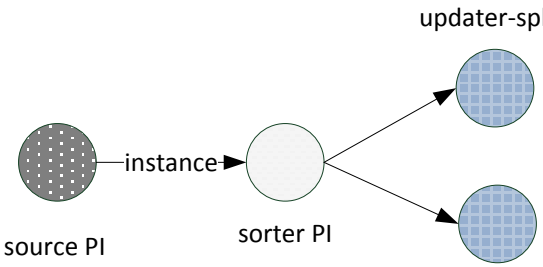
In order to choose which parallelism type, we revisit the use-case for the proposed algorithm. We derive the use-case by examining the need of Web-mining research group in Yahoo! Labs Barcelona, where we perform this thesis.

The use-case for our distributed streaming tree induction algorithm is to perform document-streaming and text-mining classification. As section 4.1.2 describes, both cases involve instances with high number of attributes. Given this use case, vertical parallelism appears to be the suitable choice for our implementation.



source PI updater-splitter PI

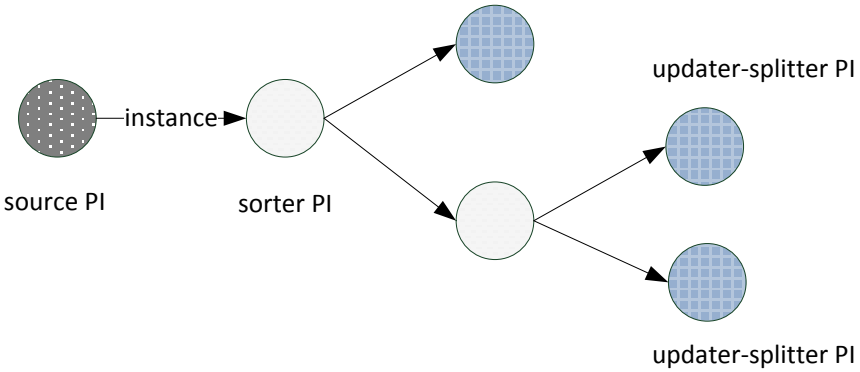
(i) induction starts with an updater-splitter PI



source PI sorter PI

(ii) the updater-splitter PI in step 1 converts to sorter PI, when the model reaches the memory limit

(iii) the algorithm creates new updater-splitter PIs that are used to extend the model in sorter PI



source PI sorter PI

(iv) the algorithm repeats step (ii) and (iii) whenever the decision tree (or subtree) in updater-splitter PI reaches memory limit.

Figure 4.5: Decision Tree Induction based on Task Parallelism

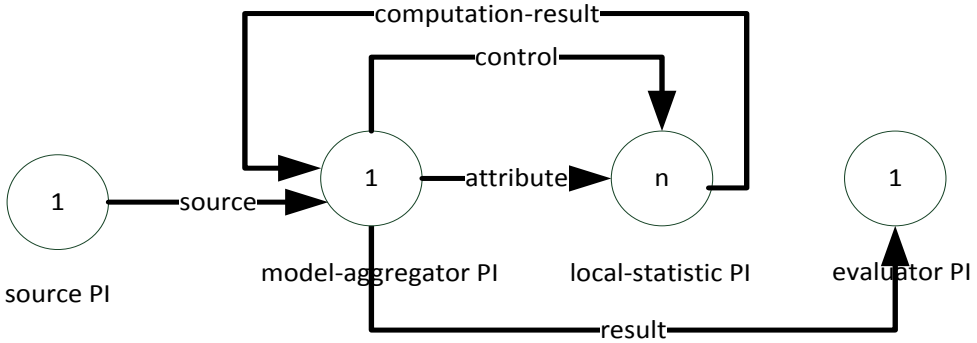


Figure 4.6: Vertical Hoeffding Tree

4.2.2 Vertical Hoeffding Tree

In this section, we refer our proposed algorithm as *Vertical Hoeffding Tree*(VHT). We reuse some definitions from section 4.1 for *processing item*(PI), *instance* and *user*. We define additional terms to make the explanation concise and clear. A *stream* is a connection between processing items that transports messages from a PI to the corresponding destination PIs. A *content event* represents a message transmitted by a PI via one or more streams.

Figure 4.6 shows the VHT diagram. Each circle represents a processing item. The number inside the circle represents the parallelism level. A model-aggregator PI consists of the decision tree model. It connects to local-statistic PI via `attribute` stream and `control` stream. As we described in section 4.1.2 about vertical parallelism, the model-aggregator PI splits instances based on attribute and each local-statistic PI contains local statistic for attributes that assigned to it. Model-aggregator PI sends the split instances via `attribute` stream and it sends control messages to ask local-statistic PI to perform computation via `control` stream. Users configure n, which is the parallelism level of the algorithm. The parallelism level is translated into the number of local-statistic PIs in the algorithm.

Model-aggregator PI sends the classification result via `result` stream to an evaluator PI for classifier or other destination PI. Evaluator PI performs evaluation of the algorithm and the evaluation could be in term of accuracy and throughput. Incoming instances to the model-aggregator PI arrive via `source` stream. The calculation results from local statistic arrive to the model-aggregator PI via `computation-result` stream.

Algorithm 4.1 shows the pseudocode of the model-aggregator PI in learning phase. The model-aggregator PI has similar steps as learning in VFDT except on the line 2 and 5. Model-aggregator PI receives `instance` content events from source PI and it extracts the instances from the content events. Then, model-aggregator PI needs to split the instances based on the attribute and send `attribute` content event via `attribute` stream to update the sufficient statistic for the corresponding leaf (line 2). `Attribute`

Algorithm 4.1 model-aggregator PI: VerticalHoeffdingTreeInduction(E, VHT_tree)

Input: E is a training instance from source PI, wrapped in `instance` content event
Input: VHT_tree is the current state of the decision tree in model-aggregator PI
1: Use VHT_tree to sort E into a leaf ls
2: Send `attribute` content events to local-statistic PIs
3: Increment the number of instances seen at l (which is n_l)
4: **if** $n_l \ mod \ n_{min} = 0$ **and** not all instances seen at l belong to the same class **then**
5: Add l into the list of splitting leaves
6: Send `compute` content event with the id of leaf l to all local-statistic PIs
7: **end if**

content event consists of leaf ID, attribute ID, attribute value, class value, and instance weight. Leaf ID and attribute ID are used by the algorithm to route the content event into correct local-statistic PIs. And attribute value, class value and instance weight are stored as local statistic in local-statistic PIs.

When a local-statistic PI receives `attribute` content event, it updates its corresponding local statistic. To perform this functionality, it keeps a data structure that store local statistic based on leaf ID and attribute ID. The local statistic here is the attribute value, weight, and the class value. Algorithm 4.2 shows this functionality.

Algorithm 4.2 local-statistic PI: UpdateLocalStatistic(*attribute*, *local_statistic*)

Input: *attribute* is an `attribute` content event
Input: *local_statistic* is the local statistic, could be implemented as $Table < leaf\_id, attribute\_id >$
1: Update *local_statistic* with data in *attribute*: attribute value, class value and instance weights

When it is the time to grow the tree(algorithm 4.1 line 4), model-aggregator PI sends `compute` content event via `control` stream to local-statistic PI. Upon receiving `compute` content event, each local-statistic PI calculates $\overline{G}_l(X_i)$ for its assigned attributes to determines the best and second best attributes. At this point,the model-aggregator PI may choose to continue processing incoming testing instance or to wait until it receives all the computation results from local-statistic PI.

Upon receiving `compute` content event, local-statistic PI calculates $\overline{G}_l(X_i)$ for each attribute to find the best and the second best attributes. Then it sends the best(X_a^{local}) and second best(X_b^{local}) attributes back to the model-aggregator PI via `computation-result` stream. These attributes are contained in `local-result` content event. Algorithm 4.3 shows this functionality.

The next part of the algorithm is to update the model once it receives all computation results from local statistic. This functionality is performed in model-aggregator PI. Algorithm 4.4 shows the pseudocode for this functionality. Whenever the algorithm receives a `local-result` content event, it retrieves the correct leaf l from the list of the splitting leaves(line 1). Then, it updates the current best attribute(X_a) and second best

Algorithm 4.3 local-statistic PI: ReceiveComputeMessage(*compute*, *local_statistic*)

Input: *compute* is an *compute* content event
Input: *local_statistic* is the local statistic, could be implemented as $Table < leaf\_id, attribute\_id >$
1: Get leaf l ID from *compute* content event
2: For each attribute belongs to leaf l in local statistic, compute $\overline{G}_l(X_i)$
3: Find X_a^{local}, which is the attribute with highest $\overline{G}_l$ based on the local statistic
4: Find X_b^{local}, which is the attribute with second highest $\overline{G}_l$ based on the local statistic
5: Send X_a^{local} and X_b^{local} using `local-result` content event to model-aggregator PI via `computation-result` stream

attribute(X_b). If all local results have arrived into model-aggregator PI, the algorithm computes Hoeffding bound and decides whether to split the leaf l or not. It proceeds to split the node if the conditions in line 5 are satisfied. These steps in line 4 to 9 are identical to basic streaming decision tree induction presented in section 3.3. To handle stragglers, model-aggregator PI has time-out mechanism in waiting for all computation results. If the time out occurs, the algorithm uses the current X_a and X_b to compute Hoeffding bound and make splitting decision.

Algorithm 4.4 model-aggregator PI: Receive(*local_result*, *VHT_tree*)

Input: *local_result* is an `local-result` content event
Input: *VHT_tree* is the current state of the decision tree in model-aggregator PI
1: Get correct leaf l from the list of splitting leaves
2: Update X_a and X_b in the splitting leaf l with X_a^{local} and X_b^{local} from *local_result*
3: **if** *local_results* from all local-statistic PIs received or time out reached **then**
4: Compute Hoeffding bound $\epsilon = \sqrt{\frac{R^2 \ln(1/\delta)}{2n_l}}$
5: **if** $X_a \neq X_\emptyset$ **and** $(\overline{G}_l(X_a) - \overline{G}_l(X_b) > \epsilon$ **or** $\epsilon < \tau)$ **then**
6: Replace l with a split-node on X_a
7: **for all** branches of the split **do**
8: Add a new leaf with derived sufficient statistic from the split node
9: **end for**
10: **end if**
11: **end if**

During testing phase, the model-aggregator PI predicts the class value of the incoming instances. Algorithm 4.5 shows the pseudocode for this functionality. Model aggregator PI uses the decision tree model to sort the newly incoming instance into the correct leaf and use the leaf to predict the class. Then, it sends the class prediction into the `result` stream.

Algorithm 4.5 model-aggregator PI: PredictClassValue($test\_instance$, VHT_tree)

Input: $test\_instance$ is a newly arriving instance
Input: VHT_tree is the decision tree model
1: Use VHT_tree to sort $test_instance$ into the correct leaf l
2: Use leaf l to predict the class of $test_instance$
3: Send classification result via `result` stream

5 Scalable Advanced Massive Online Analysis

Scalable Advanced Massive Online Analysis (SAMOA) is the distributed streaming machine learning (ML) framework to perform big data stream mining that we implement in this thesis. SAMOA contains a programming abstraction for distributed streaming algorithm that allows development of new ML algorithms without dealing with the complexity of underlying streaming processing engine (SPE). SAMOA also provides extension points for integration of new SPEs into the system. These features allow SAMOA users to develop distributed streaming ML algorithms once and they can execute the algorithm in multiple SPEs. Section 5.1 discusses the high level architecture and the main design goals of SAMOA. Sections 5.2 and 5.3 discuss our implementations to satisfy the main design goals. Section 5.4 presents our work in integrating Storm into SAMOA.

5.1 High Level Architecture

We start the discussion of SAMOA high level architecture by identifying the entities or users that use SAMOA. There are three types of SAMOA users:

1. Platform users, who need to use ML but they don't want to implement the algorithm.
2. ML developers, who develop new ML algorithms on top of SAMOA and use the already developed algorithm in SAMOA.
3. Platform developers, who extend SAMOA to integrate more SPEs into SAMOA.

Moreover, we identify three design goals of SAMOA which are:

1. Flexibility in term of developing new ML algorithms or reusing existing ML algorithms from existing ML frameworks.
2. Extensibility in term of extending SAMOA with new SPEs.
3. Scalability in term of handling ever increasing amount of data.

Figure 5.1 shows the high-level architecture of SAMOA which attempts to fulfill the aforementioned design goals. The *algorithm* block contains existing distributed streaming algorithms that have been implemented in SAMOA. This block enables platform users to easily use the existing algorithm without worrying about the underlying SPEs.

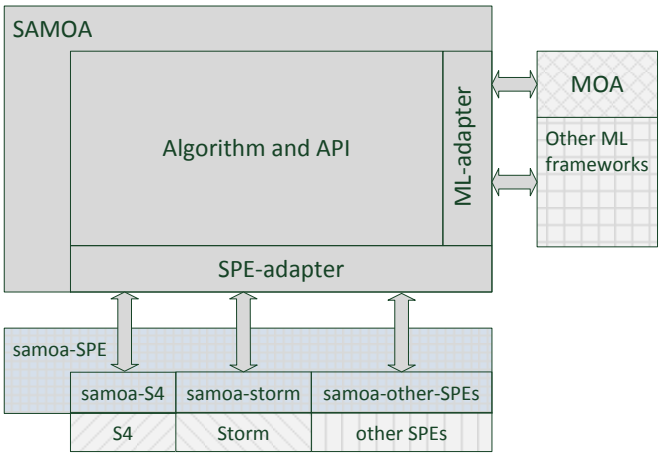


Figure 5.1: SAMOA High Level Architecture

The *application programming interface*(API) block consists of primitives and components that facilitate ML developers implementing new algorithms. The *ML-adapter* layer allows ML developers to integrate existing algorithms in MOA or other ML frameworks into SAMOA. The API block and ML-adapter layer in SAMOA fulfill the flexibility goal since they allow ML developers to rapidly develop ML algorithms using SAMOA. Section 5.2 further discusses the modular components of SAMOA and the ML-adapter layer.

Next, the *SPE-adapter* layer supports platform developers in integrating new SPEs into SAMOA. To perform the integration, platform developers should implement the *samoa-SPE* layer as shown in figure 5.1. Currently SAMOA is equipped with two layers: *samoa-S4* layer for S4 and *samoa-Storm* layer for Storm. To satisfy extensibility goal, the SPE-adapter layer decouples SPEs and ML algorithms implementation in SAMOA such that platform developers are able to easily integrate more SPEs into SAMOA. Section 5.3 presents more details about this layer in SAMOA.

The last goal, scalability, implies that SAMOA should be able to scale to cope ever increasing amount of data. To fulfill this goal, SAMOA utilizes modern SPEs to execute its ML algorithms. The reason for using modern SPEs such as Storm and S4 in SAMOA is that they are designed to provide horizontal scalability to cope with a high amount of data. Currently SAMOA is able to execute on top of Storm and S4. As this work focuses on integration of SAMOA with Storm, section 5.4 discusses about this integration.

5.2 SAMOA Modular Components

This section discusses SAMOA modular components and APIs that allow ML developers to perform rapid algorithm development. The components are: *processing item*(PI), *processor*, *stream*, *content event*, *topology* and *task*.

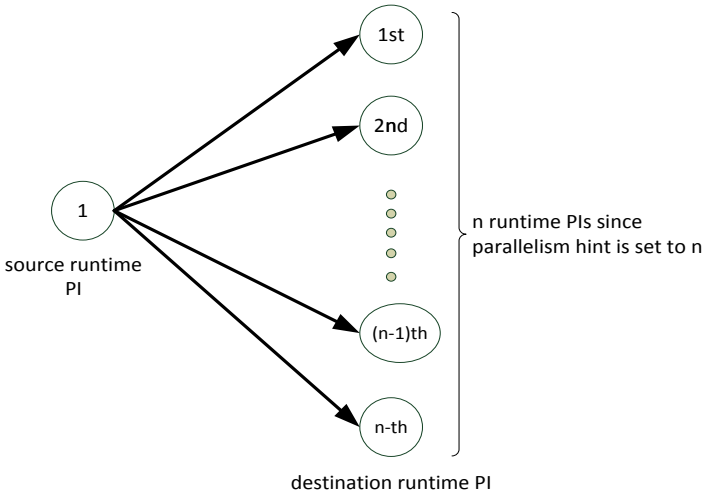


Figure 5.2: Parallelism Hint in SAMOA

5.2.1 Processing Item and Processor

A *processing item*(PI) in SAMOA is a unit of computation element that executes some part of the algorithm on *a specific SPE*. This means, each SPE in SAMOA has different concrete implementation of PIs. The SPE-adapter layer handles the instantiation of PIs. There are two types of PI, *entrance PI* and *normal PI*. An entrance PI converts data from external source into instances or independently generates instances. Then, it sends the instances to the destination PI via the corresponding stream using the correct type of content event. A normal PI consumes content events from incoming stream, processes the content events, and it may send the same content events or new content events to outgoing streams. ML developers are able to specify the *parallelism hint*, which is the number of *runtime PI* during SAMOA execution as shown in figure 5.2. A runtime PI is an actual PI that is created by the underlying SPE during execution. We implement PIs as a Java interface and SAMOA dynamically instantiates the concrete class implementation of the PI based on the underlying SPE.

A PI uses composition to contain its corresponding processor and streams. A *processor* contains the actual logic of the algorithm implemented by ML developers. Furthermore, a processor is reusable which allows ML developers to use the same implementation of processors in more than one ML algorithm implementations. The separation between PIs and processors allows ML developers to focus on developing ML algorithm without worrying about the SPE-specific implementation of PIs.

5.2.2 Stream and Content Event

A *stream* is a connection between a PI into its corresponding destination PIs. ML developers view streams as connectors between PIs and mediums to send *content event*

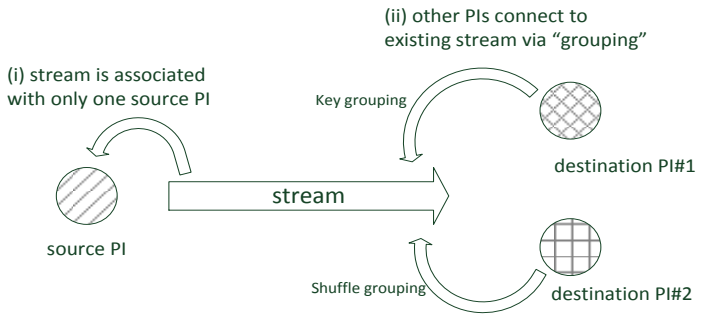


Figure 5.3: Instatiation of a Stream and Examples of Groupings in SAMOA

between PIs. A *content event* wraps the data transmitted from a PI to another via a stream. Moreover, similar to processors, content events are reusable. ML developers can reuse a content event in more than one algorithm.

Refer to figure 5.3, we define a *source PI* as a PI that sends content events through a stream. A *destination PI* is a PI that receives content event via a stream. ML developers instantiate a stream by associating it with exactly one source PI. When destination PIs want to connect into a stream, they need to specify the *grouping* mechanism which determines how the stream routes the transported content events. Currently there are three grouping mechanisms in SAMOA:

- *Shuffle* grouping, which means the stream routes the content events in a round-robin way among the corresponding runtime PIs. This means each runtime PI receives the same number of content events from the stream.
- *All* grouping, which means the stream replicates the content events and routes them to all corresponding runtime PIs.
- *Key* grouping, which means the stream routes the content event based on the *key* of the content event, i.e. the content events with the same value of key are always routed by the stream into the same runtime PI.

We design streams as a Java interface. Similar to processing items, the streams are dynamically instantiated by SAMOA based on the underlying SPEs hence ML developers do not need to worry about the streams and groupings implementation. We design content events as a Java interface and ML developers need to provide a concrete implementation of content events especially to generate the necessary *key* to perform key grouping.

5.2.3 Topology and Task

A *topology* is a collection of connected processing items and streams. It represents a network of components that process incoming data streams. A distributed streaming ML algorithm implemented on top of SAMOA corresponds to a topology.

A *task* is a machine learning related activity such as performing a specific evaluation for a classifier. Example of a task is prequential evaluation task i.e. a task that uses each instance for testing the model performance and then it uses the same instance to train the model using specific algorithms. A task corresponds also to a topology in SAMOA.

Platform users basically use SAMOA's tasks. They specify what kind of task they want to perform and SAMOA automatically constructs a topology based on the corresponding task. Next, the platform users need to identify the SPE cluster that is available for deployment and configure SAMOA to execute on that cluster. Once the configuration is correct, SAMOA deploys the topology into the configured cluster seamlessly and platform users could observe the execution results through the dedicated log files of the execution. Moving forward, SAMOA should incorporate proper user interface similar to Storm UI to improve experience of platform users in using SAMOA.

5.2.4 ML-adapter Layer

The ML-adapter layer in SAMOA consists of classes that wrap ML algorithm implementations from other ML frameworks. Currently SAMOA has a wrapper class for MOA algorithms or learners, which means SAMOA can easily utilizes MOA learners to perform some tasks. SAMOA does not change the underlying implementation of the MOA learners therefore the learners still execute in sequential manner on top of SAMOA underlying SPE.

SAMOA is implemented in Java, hence ML developers can easily integrate Java-based ML frameworks. For other frameworks not in Java, ML developers could use available Java utilities such as JNI to extend this layer.

5.2.5 Putting All the Components Together

ML developers design and implement distributed streaming ML algorithms with the abstraction of processors, content events, streams and processing items. Using these modular components, they have flexibility in implementing new algorithms by reusing existing processors and content events or writing the new ones from scratch. They have also flexibility in reusing existing algorithms and learners from existing ML frameworks using ML-adapter layer.

Other than algorithms, ML developers are also able to implement tasks also with the same abstractions. Since processors and content events are reusable, the topologies and their corresponding algorithms are also reusable. This implies, they have also flexibility in implementing new task by reusing existing algorithms and components, or by writing new algorithms and components from scratch.

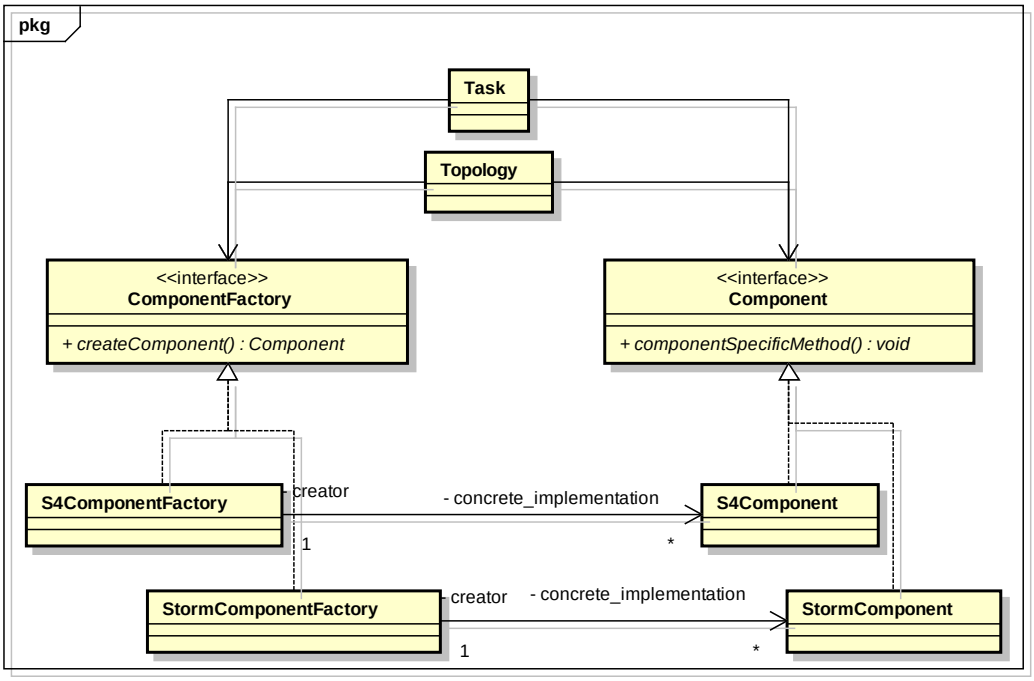


Figure 5.4: Simplified Class Diagram of SPE-adapter Layer

5.3 SPE-adapter Layer

The SPE-adapter layer decouples the implementation of ML algorithm and the underlying SPEs. This decoupling facilitates platform developers to integrate new SPEs into SAMOA. Refer to the simplified class diagram of SPE-adapter layer in figure 5.4, SPE-adapter layer uses the abstract factory pattern to instantiate the appropriate SAMOA components based on the underlying SPEs. `ComponentFactory` is the interface representing the factory and this factory instantiates objects that implement `Component` interface. The `Component` in this case refers to SAMOA's procesing items and streams. Platform developers should implement the concrete implementation of both `ComponentFactory` and `Component`. Current SAMOA implementation has factory and components implementation for S4 and Storm.

This design makes SAMOA platform agnostic which allows ML developers and platform users to use SAMOA with little knowledge of the underlying SPEs. Furthermore, platform developers are able to integrate new SPEs into SAMOA without any knowledge of the available algorithms in SAMOA.

5.4 Storm Integration to SAMOA

This section explains our work in integrating Storm into SAMOA through the aforementioned SPE-adapter layer. To start the explanation, section 5.4.1 presents some basic knowledge of Storm that related to the SPE-adapter layer. Section 5.4.2 discusses our proposed design in adapting Storm components into SAMOA.

5.4.1 Overview of Storm

Storm is a distributed streaming computation framework that utilizes MapReduce-like programming model for streaming data. Storm main use case is to perform real-time analytics for streaming data. For example, Twitter uses Storm[1] to discover emerging stories or topics, to perform online learning based on tweet features for ranking of search results, to perform realtime analytics for advertisement and to process internal logs. The main advantage of Storm over Hadoop MapReduce (MR) is its flexibility in handling stream processing. In fact, Hadoop MR has complex and error-prone configuration when it is used for handling streaming data. Storm provides at-least-once message processing. It is designed to scale horizontally. Storm does not have intermediate queues which implies less operational overhead. Moreover, Storm promises also to be a platform that "just works".

The fundamental primitives of Storm are *streams*, *spouts*, *bolts*, and *topologies*. A stream in Storm is an unbounded sequence of *tuple*. A tuple is a list of values and each value can be any type as long as the values are serializable. Tuples in Storm are dynamically typed which means the type of each value need not be declared. Example of a tuple is `{height, weight}` tuple which consists of `height` value and `weight` value. These values logically should be a `double` or `float` but since they are dynamic types, the tuple can contain any data type. Therefore, it is the responsibility of Storm users to ensure that a tuple contains correct type as intended.

A spout in Storm is a source of streams. Spouts typically read from external sources such as kestrel/kafka queues, http server logs or Twitter streaming APIs. A bolt in Storm is a consumer of one or more input streams. Bolts are able to perform several functions for the consumed input stream such as filtering of tuples, aggregation of tuples, joining multiple streams, and communication with external entities (such as caches or databases). Storm utilizes *pull* model in transferring tuple between spouts and bolts i.e. each bolt pulls tuples from the source components (which can be other bolts or spouts). This characteristic implies that loss of tuples only happens in spouts when they are unable to keep up with external event rates.

A topology in Storm is a network of spouts, bolts and streams that forms a directed-acyclic-graph. Figure 5.5 shows the example of a topology in Storm. There are two spouts(`S1` and `S2`) and five bolts(`B1` to `B5`). A spout can send tuples to more than one

[1]http://www.slideshare.net/KrishnaGade2/storm-at-twitter

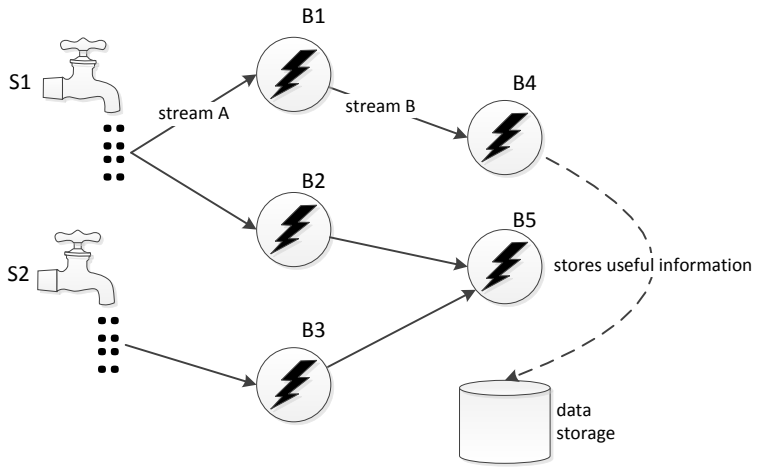


Figure 5.5: Example of Storm Topology

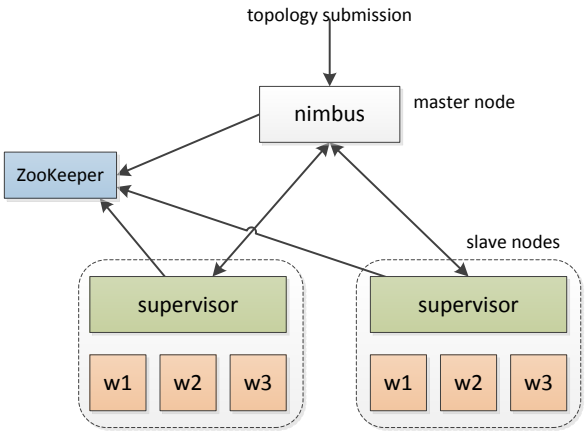


Figure 5.6: Storm Cluster

different bolts. Moreover, one or more streams can arrive to a single bolt. Refer to figure 5.5, bolt `B1` processes stream A and produces stream B. Bolt `B4` consumes stream B and communicates with external storage to store useful information such as the current state of the bolt.

Figure 5.6 shows a Storm cluster which consists of these following components: one *nimbus*, two *supervisors*, three *workers* for each supervisor and a ZooKeeper cluster. A nimbus is the master node in Storm that acts as entry point to submit topologies and code (packaged as jar) for execution on the cluster. It distributes the code around the cluster via supervisors. A supervisor runs on slave node and it coordinates with ZooKeeper to manage the workers. A worker in Storm corresponds to a JVM process that executes a subset of a topology. Figure 5.6 shows three workers (`w1`, `w2`, `w3`) in each supervisor. A worker comprises of several *executors* and *tasks*. An executor corresponds to a thread

spawned by a worker and it consists of one or more tasks from the same type of bolt or spout. A task performs the actual data processing based on spout or bolt implementations. Storm cluster uses ZooKeeper cluster to perform coordination functionalities by storing the configurations of Storm workers.

To determine the partitioning of streams among the corresponding bolt's tasks, Storm utilizes stream groupings mechanisms. Storm allows users to provide their own implementation and grouping. Furthermore, Storm also provides several default grouping mechanisms, such as:

1. `shuffle` grouping, in which Storm performs randomized round robin. Shuffle grouping results in the same number of tuples received by each task in the corresponding bolt.
2. `fields` grouping, which partitions the stream based on the values of one or more fields specified by the grouping. For example, a stream can be partitioned by an "id" value hence a tuple with the same id value will always arrive to the same task.
3. `all` grouping, which replicates the tuple to all tasks in the corresponding bolt.
4. `global` grouping, which sends all tuples in a stream in to only one task.
5. `direct` grouping, which sends each tuple directly to a task based on the provided task id.

5.4.2 Proposed Design

In order to integrate Storm components to SAMOA, we need to establish relation between Storm classes and the existing SAMOA components. Figure 5.7 shows the class diagram of our proposed implementation. In this section we refer our implementation in integrating Storm into SAMOA as *samoa-storm*. Samoa-storm consists of concrete implementation of processing items, they are `StormEntranceProcessingItem` and `StormProcessingItem`. Both of them also implement `StormTopologyNode` to provide common functionalities in creating `StormStream` and in generating unique identification. A `StormEntranceProcessingItem` corresponds with a spout with composition relation. Similarly, a `StormProcessingItem` corresponds to a bolt with composition relation. In both cases, composition is favored to inheritance because Storm differentiates between the classes that help constructing the topology and the classes that execute in the cluster. If PIs are subclasses of Storm components (spouts or bolts), we will need to create extra classes to help constructing the topology and it may increase the complexity of the layer.

In samoa-storm, `StormEntranceProcessingItem` and `StormProcessingItem` are the ones which construct topology. And their corresponding Storm components, `StormEntranceSpout` and `ProcesingItemBolt` are the ones which execute in the cluster.

A `StormStream` is an abstract class that represents implementation of SAMOA stream. It has an abstract method called `put` to send content events into destination PI. Samoa-storm uses abstract class because spouts and bolts have a different way of sending tuples

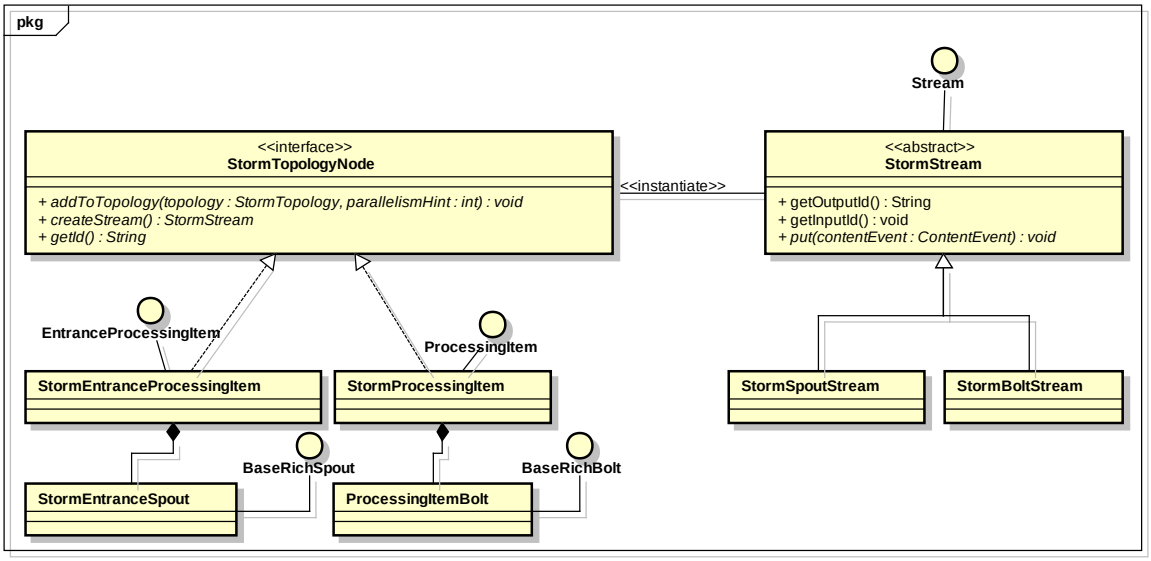


Figure 5.7: samoa-storm Class Diagram

into destination bolts. A `StormStream` basically is a container for stream identification strings and Storm specific classes to send tuples into destination PIs. Each `StormStream` has a string as its unique identification(ID). However, it needs to expose two types of identifications: input ID and output ID. Samoa-storm uses the input ID to connect a bolt into another bolt or a spout when building a topology. An input ID comprises of the unique ID of processing item in which the stream is associated to, and the stream's unique ID. The reason why an input ID consists of two IDs is that Storm needs both IDs to uniquely identify connection of a bolt into a stream. An output ID only consists of the unique ID of `StormStream` and Storm uses the output ID to declare a stream using Storm APIs.

A `StormSpoutStream` is a concrete implementation of `StormStream` that is used by a `StormEntranceProcessingItem` to send content events. The `put` method in `StormSpoutStream` puts the content events into a `StormEntranceSpout`'s queue and since Storm utilizes pull model, Storm starts sending the content event by clearing this queue. On the other hand, a `StormProcessingItem` uses a `StormBoltStream` to send content events to another `StormProcessingItem`. `StormBoltStream` utilizes bolt's `OutputCollector` to perform this functionality.

6 Evaluation

This section presents our evaluation of SAMOA and our proposed algorithm, Vertical Hoeffding Tree (VHT). Section 6.1 describes the specifications of the nodes where we performed the evaluation. Section 6.2 discusses the input data for the evaluation. Next, section 6.3 presents two versions of our algorithm that we evaluated. Section 6.4 presents the results of our evaluation. And finally section 6.5 presents our discussion on the evaluation.

6.1 Cluster Configuration

For this evaluation, we had access to a *high-memory(hm)* cluster. The hm cluster consists of three nodes. Each node had 48 GB of RAM and dual quad-core Intel Xeon CPU E5620 @ 2.40 GHz with hyper-threading that provides capacity of 16 processors. Moreover, each node had Red Hat Enterprise Linux Server Release 5.4 with Linux kernel version 2.6.18 as the operating system. This cluster was a shared cluster between scientists and engineers in our lab.

We deployed Java(TM) SE Runtime Environment version 1.7.0, Storm version 0.8.2, ZeroMQ version 2.1.7, specific jzmq version that works with Storm[1], as well as SAMOA with VHT implementation on the hm cluster. We used ZooKeeper version 3.3.3 that was deployed on top of a set of three machines in the hm cluster. Furthermore, we used YourKit Java Profiler version 11.0.10 to perform any profiling task in this evaluation section. We also used Storm UI to monitor the status of our deployed Storm cluster. We configured Storm with these following configuration:

- Each Storm cluster consisted of three supervisor nodes.
- Each Storm supervisor node had four available slots for Storm workers. Hence the total workers in our deployed Storm cluster were 12.
- Each Storm worker was configured to use maximum 6 GB of memory.

[1]http://github.com/nathanmarz/jzmq

Index	Word	Frequency
1	excite	90
2	cool	45
3	watch	5
4	game	6

Table 6.1: Sample Table Used by Text Generator

6.2 Test Data

We used two types of data source in our experiment. The first one was the MOA random-tree generator which produced a decision tree model. We refer to this generator as *tree generator*. This generator randomly chose attributes at random to split the tree leaf nodes and grow the tree model. This generator produced new instances by assigning uniformly-distributed random values to the available attributes. Then, it used the random values to determine the class label via the tree. The instances generated by this generator were *dense* instances i.e. the instances contained all values of their corresponding attributes. A dense instance with 100 attributes could be implemented as an array with size equals to 100 and each entry in the array corresponds to each attribute in the instance. This generator was based on [26].

Using the tree generator above, we generated data stream with these following number of attributes: 10, 30, 50, and 100. Each setting had the same proportion of numeric and nominal attributes i.e. 5 numeric and 5 nominal attributes for the setting with 10 instances. We refer to these configurations as "*tree-number_of_instance*" for example: tree-10 is the setting with 10 instances.

The second data source, *text generator*, was a generator that produced text data stream that resembled tweets for text-mining classification. The text generator simulated real tweets by using a table with words and their normalized frequencies. The table itself was built by the generator based on a Zipfian distribution for the frequency of the new tweets. To represent the tweet, the generator utilized *sparse* instances i.e. the instances only contained attributes which had non-zero value. Given this following generated tweet *gt*:"excite watch game" and the corresponding Table 6.1, the corresponding sparse instance for the tweet *gt* was `<1 1, 3 1, 4 1>`. Each attribute entry consisted of two integer values. The first entry of the instance, `1 1`, means the word `excite` (which has index one in the table) had frequency of one. By utilizing word-stemming[2] technique, the generated tweet *gt* may correspond to this following tweet: "More exciting than watching a game of Quidditch. #SurviveTheNight bit.ly/13IQI15 #sp"[3]. This generator was initially used in [36].

Using the text generator, we generated data stream with 15 words in each instance and we used this following number of attributes: 10, 30, 50, 100, 200, 300, and 1000. We

[2]http://en.wikipedia.org/wiki/Stemming

[3]http://twitter.com/Lord_Voldemort7/status/342701915756457984

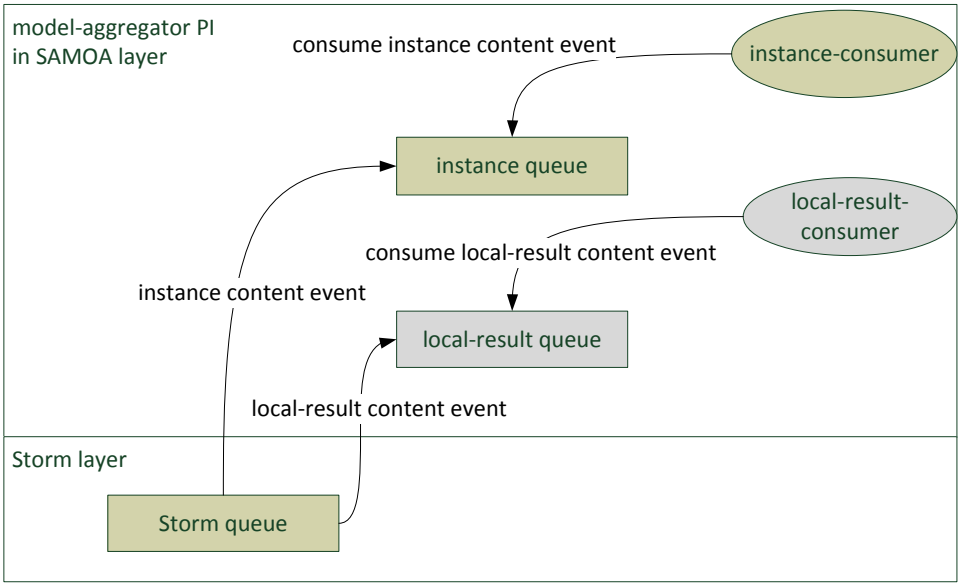


Figure 6.1: model-aggregator PI in VHT1

refer to these configurations as *text-number of instance* for example: text-10 is the setting with 10 instances.

For each generator, we produced 1100000 instances in each experiment and we measured the throughput from the 100000th instance onward. We excluded the first 100000 instances because the first 100000th instance was the warm-up period for the algorithm to setup necessary data structures. We repeated each setting for both generators five times. In the following sections, we will mention whether the presented results are the combination of the five executions, or just one of five executions.

6.3 VHT Implementations

We implemented two types of VHT in this thesis. The first implementation, VHT1, focused on achieving the same accuracy as sequential MOA Hoeffding Tree (MHT).

As shown in figure 6.1, VHT1 utilized two internal queues (`instance` queue and `local-result` queue) with their corresponding consumer threads (`instance-consumer` thread and `local-result-consumer` thread) in model-aggregator PI to consume the content events from Storm layer due to the loop configuration between model-aggregator PI and local-statistic PIs. Refer to figure 4.6 in section 4.2.2, the loop configuration caused model-aggregator PI to consume two different streams(`computation-result` stream and `source` stream) that not independent. Therefore,in order to have same accuracy as MHT, VHT1 stopped consuming content event from `source` stream and waited for all corresponding content events from `computation-result` stream when VHT1 was growing the

tree. The sequence of events that lead into this scenario were:

1. VHT1 attempted to split by sending `compute` content event.
2. New `instance` content events arrived in Storm queue.
3. Queue in Storm reached a state where there always were `instance` content event due to arrival rate was higher than processing rate.
4. VHT1 consumed `instance` content event from Storm queue by moving it into `instance` queue and waited until all `local-result` content events to arrive.
5. By moving the `instance` content event to `instance` queue, VHT1 was able to consume `local-result` content event and splits the node before continue processing the next `instance` content event.

Once VHT1 achieved the same accuracy value with MHT for the same input data, we extended VHT1 to improve the speed. We refer to our second implementation as VHT2 which was implemented with these following enhancements:

1. VHT2 used Kryo[4] serializer for `attribute` content event and `compute` content event instead of using Java default serialization.
2. VHT2 used `long` as leaf identifier instead of `string`. Note that the leaf identifier were always sent to local-statistic PI as part of `attribute` content event.
3. VHT2 replaced the internal queues in VHT1 with only one single queue, but VHT2 discarded `instance` content events that arrive into a splitting leaf i.e. an instance arrived into a leaf while the leaf was attempting to split and waiting for all its corresponding `local-result` content events.

To speed up the development of both algorithms, we reused several MOA and WEKA APIs such as the instance and the attribute data structure. We present the evaluation for both implementations in the section 6.4 below.

6.4 Results

In this evaluation, we used a machine learning task called *prequential* evaluation to measure the accuracy over time. This evaluation task used each instance for testing the model (i.e. the task used the model to predict the class value of the tested instance), and then it further used the same instance for training the model which was equal to growing the tree in our case. We utilized two metrics to perform the evaluation. The first metric is accuracy and it refers to the number of correctly classified instances divided by the

[4]http://code.google.com/p/kryo/

total number of instances that are used in the evaluation. The second one is throughput and it refers to the number of instances per second that prequential evaluation is able to process using a specific classification algorithm. In this chapter, we refer to throughput for prequential evaluation using VHT as *throughput for VHT* and for MHT as *throughput for MHT*.

6.4.1 Accuracy Results

For VHT1, we used tree-10 and tree-30 generator and we observed that VHT1 had the same accuracy as MHT. The reason was that VHT1 waited for all `local-result` content events for the splitting leaf before further processing the `instance` content event. This characteristic made VHT1 to have the same logic as the MHT implementation. However, we were not able to test VHT1 with high number of attributes due to out of memory error. The reason for this error was that the `instance` queue in VHT1 grew infinitely and consumed all the allocated memory. We performed application profiling for VHT1 and we found that serialization and deserialization in model-aggregator PI consumed close to 50% of total execution time. Furthermore, Storm UI showed that model-aggregator PI was the bottleneck. Hence, we did not perform more evaluation of VHT1 and we directly proceeded to implement VHT2 which contained some enhancements for VHT1 presented in section 6.3.

We configured VHT2 with parallelism hints of 1, 3, 6 and 9. This parallelism hint corresponded to the number of Storm worker processes for the local-statistic PIs. Figure 6.2 shows the accuracy of VHT2 and MHT using tree-10 and tree-100 in one of the five executions that we performed. We observed that the other executions had the same trend as the result shown. In figure 6.2, *VHT2-par-x* refers to VHT2 implementation with parallelism hint of x, the y-axis shows the accuracy and the x-axis shows the number of instances that arrive into the algorithm.

Refer to figure 6.2, MHT has higher accuracy compared to the all of the VHT2 configurations. Since VHT2 discarded instances when splitting a leaf, VHT2 produced model that less accurate as MHT. In term of decision tree, the model generated by VHT2 contained less number leaf and split nodes as shown in figure 6.3. All of the VHT2 configurations increased the number of leaf nodes when the number of instances increased. This behavior implies that VHT2 was able to learn from instances generated by the tree generator and performed the decision tree induction correctly.

Figure 6.4 shows the accuracy of VHT2 versus MHT using text-10, text-100 and text-1000 configuration. Compared to MHT, VHT2 achieved very close accuracy for text-10 and text-100 configuration. However it performed rather poor for text-1000 setting, and it might be due to too many discarded instances in text-1000 configuration. Both MHT and VHT2 achieved close to 95% for text-100 and their accuracy dropped in text-1000 configuration. And in all configurations, we observed that the accuracy was stagnant, i.e. the accuracy did not increase significantly when the number of instance increased. This stagnancy implied that both VHT2 and MHT did not grow the tree effectively to increase their accuracy.

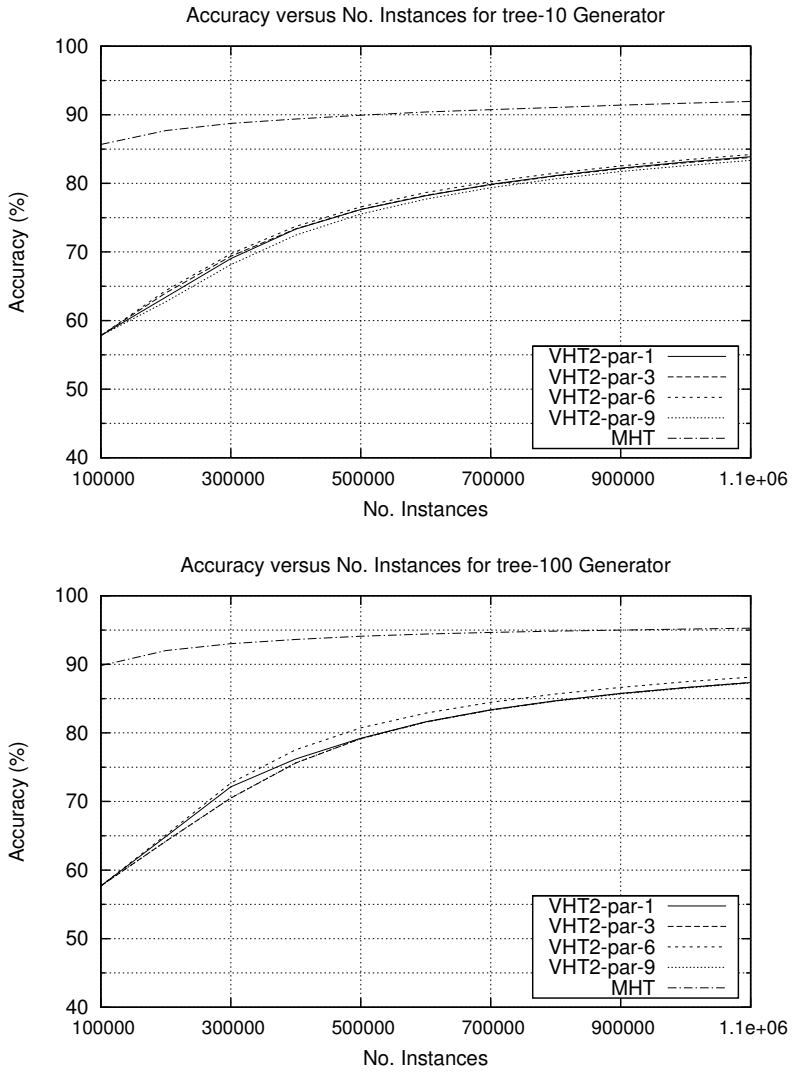


Figure 6.2: Accuracy of VHT2 and MHT for tree-10 and tree-100 generator

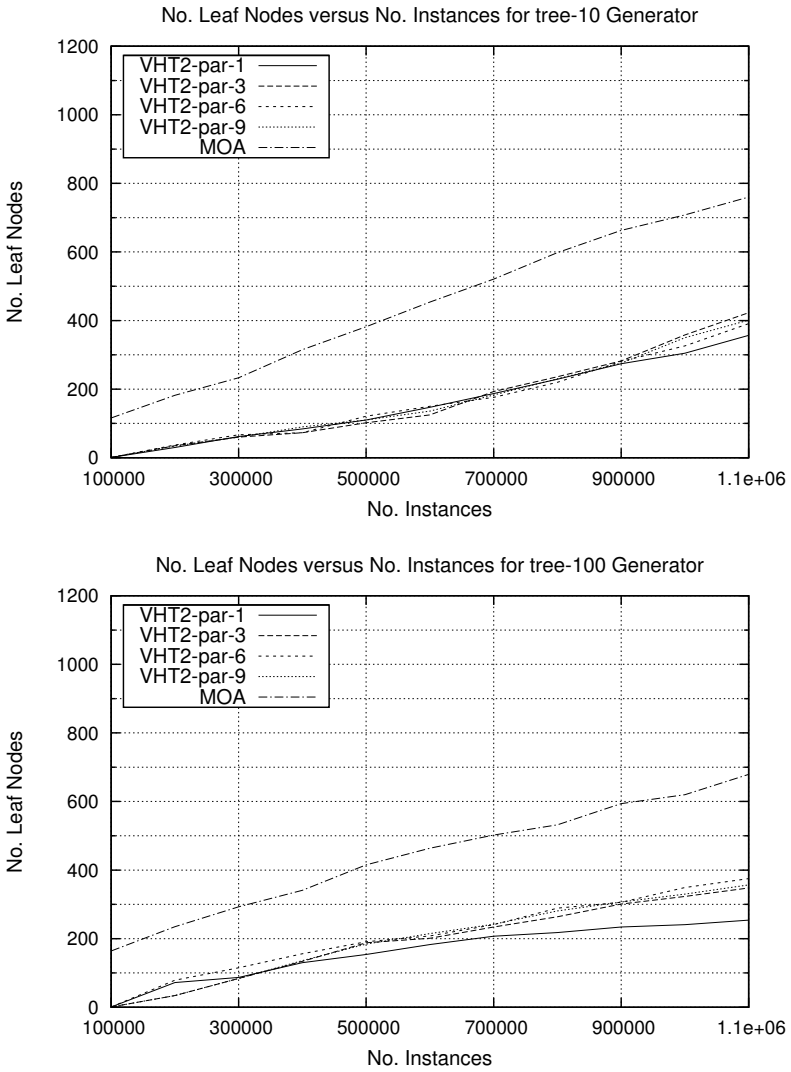


Figure 6.3: Number of leaf nodes of VHT2 and MHT for tree-10 and tree-100 generator

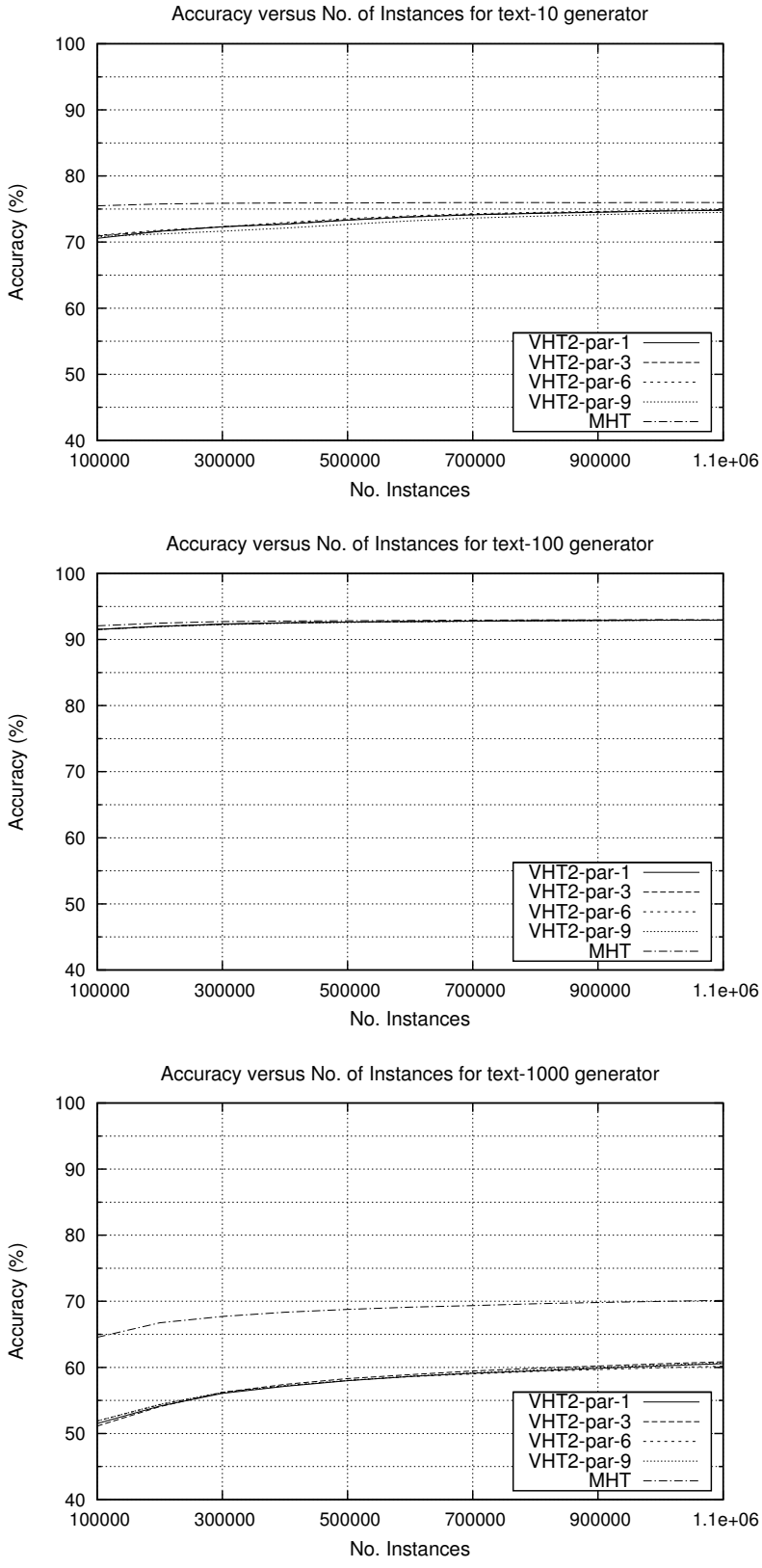


Figure 6.4: Accuracy of VHT2 and MHT for text-10, text-100 and text-1000 generator

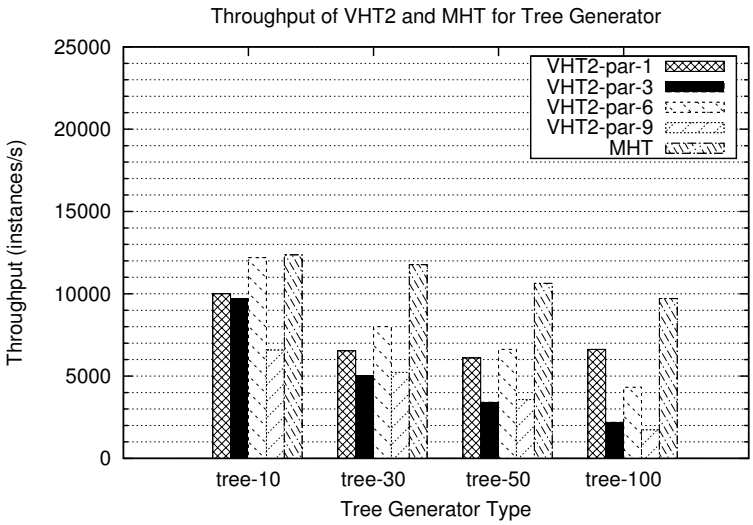


Figure 6.5: Throughput of VHT2 and MHT for Tree Generator

6.4.2 VHT Throughput Results

Figure 6.5 shows the throughput of VHT2 and MHT with tree generator. Although we increased the parallelism hint, VHT2 had lower throughput compared to MHT for all types of tree generator.

To find out why VHT2 had lower throughput than MHT, we performed profiling on VHT2's model-aggregator PI and the profiling showed that the bottleneck was in Storm internal queue's lock and ZeroMQ call to send data. This bottleneck was caused by the way model-aggregator PI split an `instance` content event to its corresponding `attribute` content events. Model-aggregator PI iterated over all attributes in an `instance` content event and it used SAMOA's shuffle-grouping mechanism to route the `attribute content event`. If there was 100 attributes and 1000000 instances, model-aggregator PI needs to send 100000000 `attribute` content events into its corresponding local-statistic PIs.

We continued our evaluation with using text generator. Figure 6.6 shows the throughput of VHT2 and MHT with text generator. For brevity and clarity of the figure, we exclude text-30 and text-200 because text-30 results were similar to text-10 and text 200 results were similar to text-300. Figure 6.6 shows that VHT2 achieved the same or even higher throughput than MHT, such as in VHT2 with parallelism hint of 3 (VHT2-par-3) in the text-300 setting. However, increasing the parallelism hint did not increase the throughput significantly. This behavior was unexpected since sparse instances only had very few non-zero attributes so that model-aggregator PI should split the sparse instance content event into few number of its corresponding attribute content events.

We then inspected our implementation and discovered that the MOA API to iterate through the sparse instances attribute was the root cause. VHT2 used MOA API to iterate the attributes and the API performed the iteration based on the maximum possible

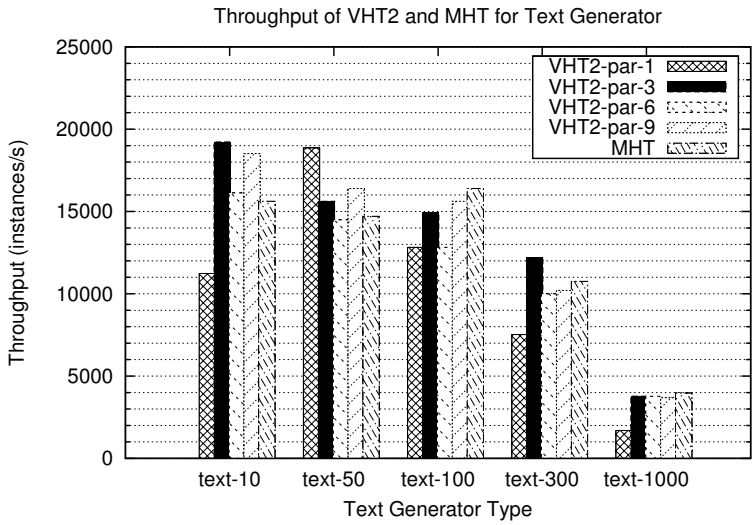


Figure 6.6: Throughput of VHT2 and MHT for Text Generator

number of the attributes that sparse instances may have. For example if a sparse instance contains only five attributes and it could have up to 1000 attributes, the API will iterate through all 1000 attributes instead of the five attributes contained by the sparse instance.

The throughput could be improved by reducing communication time between model-aggregator PI and local-statistic PIs. This communication time corresponds to the number of content events sent between model-aggregator PI and local-statistic PIs.

We continued our evaluation by performing Java profiling for text-1000 and text-10000 settings to find out part of the code that consume significant amount of time. Figure 6.7 shows the profiling results. *t_serial* refers to the execution time of parts of code that were not parallelized. *t_comm* refers to communication time to distribute the attributes into local-statistic PIs. And *t_calc* refers to the calculation time for determining the attributes with the best information-theoretic criteria.

Refer to figure 6.7, for text-1000 setting, VHT2 with parallelism hint 3 (VHT2-par-3) had 111.69 seconds of communication time which corresponds to 41.16% of total execution time. If we could reduce this communication time, VHT2-par-3 could achieve lower execution time and higher throughput compared to MHT. The more interesting results were in the text-10000 settings where execution time of VHT2-par-3 was significantly lower execution time compared to MHT. VHT2-par-3 had higher throughput and this implies VHT was better than MHT for sparse instances with high number of attributes.

Figure 6.8 shows the throughput for three high number of attributes settings. VHT2 was able to achieve around 5 times faster than MHT for text-10000 setting and almost 3 times faster than MHT for text-100000 setting. This means, VHT2 was suitable for high number of attributes. The decrease in the overall throughput when the number of attribute was increased could be caused by the cost of instance serialization since in our current implementation, the Kryo serializers were only used for serializing the content events inside VHT.

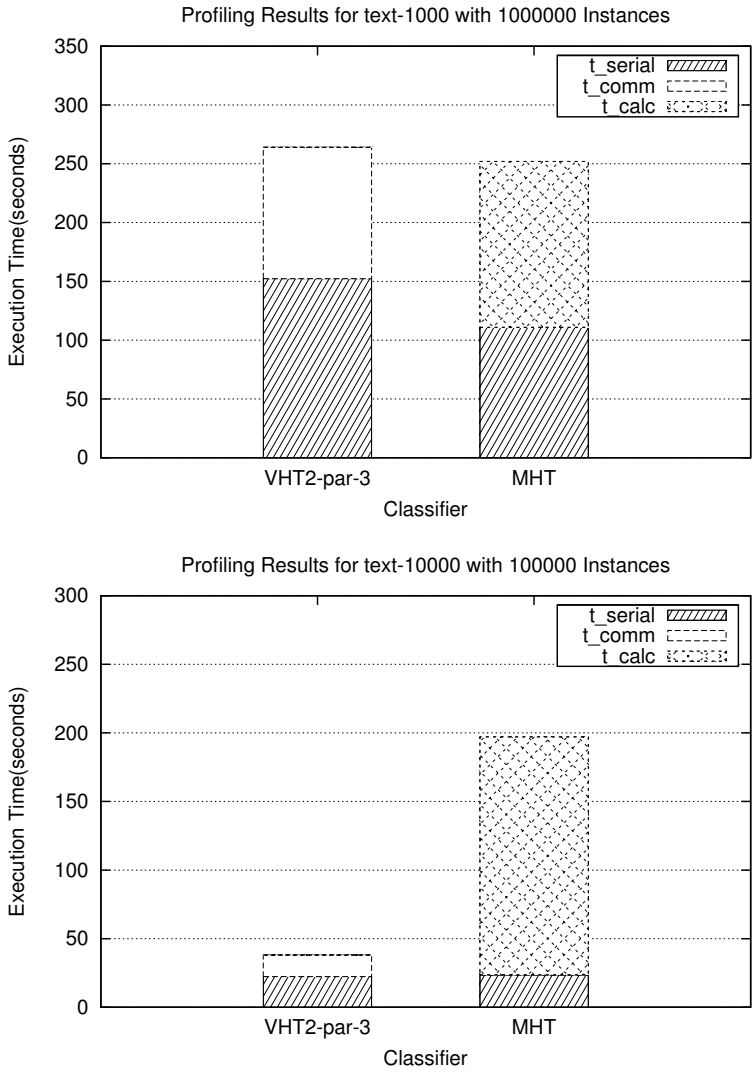


Figure 6.7: Profiling of VHT2 and MHT for text-1000 and text-10000 settings

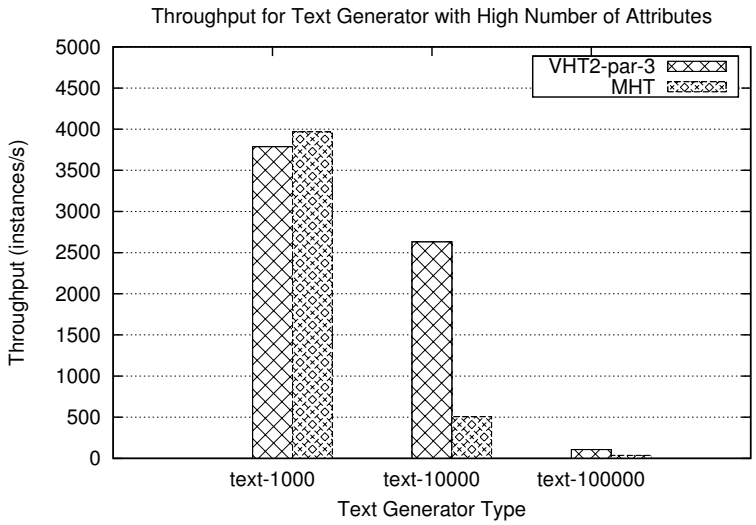


Figure 6.8: Throughput of VHT2 and MHT for high number of attributes settings

6.5 Discussion

Ye et. al. [22] show that minimizing the communication time between distributed components improve the performance of their distributed classification algorithm. In VHT2, minimizing the communication time between model-aggregator PI and local-statistic PI could reduce the discarded instances and increase the accuracy of VHT2. It could also improve the throughput of VHT2. One possible way to implement this solution is to send the content events in batch using Storm's direct grouping and not using shuffle grouping. In this implementation, the responsibility of deciding which destination task shifts from Storm's shuffle grouping to SAMOA's processors. This implies ML developers need to ensure that the correct tasks receive the correct message. Another possible implementation is to rewrite the MOA API to perform the iteration so that it only iterates through the attributes that a sparse instance contains. There is also another possible workaround which do not require us to modify the algorithm, which is to deploy VHT2 only in one machine, hence the parallelism hint corresponds to parallelism in term of CPU processors.

We developed VHT1 and VHT2 presented in section 6.3 only in relatively short time, which was ten working days. This rapid development of VHT1 and VHT2 shows that SAMOA framework fulfills its design goal of flexibility. We also utilized MOA Hoeffding Tree classifier inside SAMOA for evaluation and we were able to perform the integration with ease.

In general, this evaluation shows that we need to design distributed and parallel streaming ML algorithm with care especially in term of communication between the components of the algorithm. Minimizing the communication time between distributed components is very important to achieve a high performance distributed and parallel streaming ML algorithm. This evaluation also shows that vertical parallelism is suitable for high number of attributes. Moreover the existence of a flexible distributed streaming framework, such as SAMOA, increases the productivity of ML developers and speeds up the

development of new distributed streaming ML algorithms.

7 Conclusions

Mining big data streams is one of the biggest challenge for web companies. The need of systems that are able to perform this task is becoming very important. These systems need to satisfy the three dimensions of big data: volume, velocity and variety. However, most of the existing systems only satisfy up to two out of the three dimensions. Machine learning frameworks for processing data streams in distributed and parallel manner are needed to solve this challenge. These frameworks should satisfy all the three dimensions of big data.

In this thesis, we developed Scalable Advanced Massive Online Analysis (SAMOA), a distributed streaming machine learning framework to perform big data streams mining. SAMOA ML-adapter layer decouples SAMOA with underlying streaming processing engine. As a result, we successfully integrated Storm into SAMOA by extending the ML-adapter layer with samoa-storm module. Furthermore, we also studied possible parallelisms for implementing a distributed algorithm. Then, we implemented one of the parallelisms as a distributed online decision tree induction algorithm on top of SAMOA.

We can conclude that SAMOA provides the flexibility in developing new machine learning algorithms or reusing existing algorithms. SAMOA also provides extensibility in integrating new SPEs using the ML-adapter module. In term of the proposed distributed decision tree algorithm, we can conclude that the algorithm needs to reduce the communication time between its components to achieve higher throughput performance compared to the existing sequential online decision tree induction algorithms. Moreover, we also can conclude that the proposed algorithm is suitable for high number of attributes setting.

7.1 Open Issues

Due to the time restrictions of this project, there are several issues that remain unexplored. The first one is the usage of MOA data structures in our SAMOA implementation. SAMOA should have its own basic data structure to represent fundamental components of distributed streaming machine learning algorithms. Examples of the components are `instance` and `attribute`. By having the fundamental components that are not part of other framework, we could optimize the components easily for implementing distributed algorithms. One example of optimization is to provide Kryo serializer for each component so that SPEs that support Kryo can minimize the communication cost in sending this component. Another issue is the implementation VHT3, which is the third iteration of our proposed distributed online decision tree induction algorithm. VHT3 should con-

tain the enhancements that we propose in evaluation section so that it can be used in document-streaming and text-mining classifications.

7.2 Future Work

We plan to release SAMOA as an open-source project so that more SPEs and distributed online algorithms could be integrated into the platform. This effort aims to nurture SAMOA into an established machine learning framework. One example of SAMOA enhancement is the introduction of evaluation layer in SAMOA. This layer provides extension points to implement customized evaluation techniques for evaluating the distributed algorithms on top of SAMOA. Therefore, SAMOA will have loose coupling between the algorithm logic and evaluation logic. This will benefit SAMOA users since they can customize the evaluation based on their needs and requirements.

Furthermore, we plan to implement online classification algorithms that are based on horizontal-parallelism on top SAMOA. There are several online classification algorithms that fit into horizontal-parallelism such as online bagging and boosting ensemble classifiers. As these classifiers are naturally parallel, it should take little effort to implement in SAMOA. The implementation of these ensemble classifiers will provide us with a better understanding of horizontal-parallelism-based algorithm for online classification in distributed environments.

References

[1] Gartner, "Gartner says solving 'Big data' challenge involves more than just managing volumes of data," 2011.

[2] S. Ghemawat, H. Gobioff, and S.-T. Leung, "The Google File System," in *Proceedings of the nineteenth ACM symposium on Operating systems principles*, SOSP '03, (New York, NY, USA), pp. 29–43, ACM, 2003.

[3] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing On Large Clusters," *Commun. ACM*, vol. 51, pp. 107–113, Jan. 2008.

[4] G. DeCandia, D. Hastorun, M. Jampani, G. Kakulapati, A. Lakshman, A. Pilchin, S. Sivasubramanian, P. Vosshall, and W. Vogels, "Dynamo: Amazon's Highly Available Key-value Store," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 205–220, Oct. 2007.

[5] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol. 12, pp. 120–139, Aug. 2003.

[6] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R. Motwani, U. Srivastava, and J. Widom, "Stream: The Stanford Data Stream Management System," *Book chapter*, 2004.

[7] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. S. Maskey, A. Rasin, and E. Ryvkina, "The Design of the Borealis Stream Processing Engine," 2005.

[8] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari, "S4: Distributed Stream Computing Platform," in *2010 IEEE International Conference on Data Mining Workshops (ICDMW)*, pp. 170–177, 2010.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten, "The WEKA Data Mining Software: An Update," *ACM SIGKDD Explorations Newsletter*, vol. 11, no. 1, pp. 10–18, 2009.

[10] T. Kraska, A. Talwalkar, J. Duchi, R. Griffith, M. J. Franklin, and M. Jordan, "MLBase: A Distributed Machine-Learning System," in *In Conference on Innovative Data Systems Research*, 2013.

[11] R. Caruana and A. Niculescu-Mizil, "An Empirical Comparison of Supervised Learning Algorithms," in *Proceedings of the 23rd international conference on Machine learning*, ICML '06, (New York, NY, USA), pp. 161–168, ACM, 2006.

[12] R. Caruana, N. Karampatziakis, and A. Yessenalina, "An Empirical Evaluation of Supervised Learning in High Dimensions," in *Proceedings of the 25th international*

conference on Machine learning, ICML '08, (New York, NY, USA), pp. 96–103, ACM, 2008.

[13] J. R. Quinlan, "Induction of Decision Trees," *Machine learning*, vol. 1, no. 1, pp. 81–106, 1986.

[14] R. C. Barros, R. Cerri, P. A. Jaskowiak, and A. de Carvalho, "A Bottom-up Oblique Decision Tree Induction Algorithm," in *Intelligent Systems Design and Applications (ISDA), 2011 11th International Conference on*, pp. 450–456, 2011.

[15] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, "Pregel: A System for Large-Scale Graph Processing," in *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, SIGMOD '10, (New York, NY, USA), pp. 135–146, ACM, 2010.

[16] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein, "Distributed GraphLab: A Framework for Machine Learning and Data Mining in The Cloud," *Proc. VLDB Endow.*, vol. 5, pp. 716–727, Apr. 2012.

[17] Y. Low, J. Gonzalez, A. Kyrola, D. Bickson, C. Guestrin, and J. M. Hellerstein, "GraphLab: A New Framework for Parallel Machine Learning," *arXiv:1006.4990*, June 2010. The 26th Conference on Uncertainty in Artificial Intelligence (UAI 2010), Catalina Island, California, July 8-11, 2010.

[18] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. Franklin, S. Shenker, and I. Stoica, "Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing," in *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pp. 2–2, 2012.

[19] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. Katz, S. Shenker, and I. Stoica, "Mesos: A Platform for Fine-Grained Resource Sharing in the Data Center," in *Proceedings of the 8th USENIX conference on Networked systems design and implementation*, NSDI'11, (Berkeley, CA, USA), pp. 22–22, USENIX Association, 2011.

[20] D. Gillick, A. Faria, and J. DeNero, "MapReduce: Distributed Computing for Machine Learning," *Technical Report, Berkeley*, 2006.

[21] B. Panda, J. S. Herbach, S. Basu, and R. J. Bayardo, "PLANET: Massively Parallel Learning of Tree Ensembles with MapReduce," *Proc. VLDB Endow.*, vol. 2, pp. 1426–1437, Aug. 2009.

[22] J. Ye, J.-H. Chow, J. Chen, and Z. Zheng, "Stochastic Gradient Boosted Distributed Decision Trees," in *Proceedings of the 18th ACM conference on Information and knowledge management*, CIKM '09, (New York, NY, USA), pp. 2061–2064, ACM, 2009.

[23] A. Bifet, G. Holmes, R. Kirkby, and B. Pfahringer, "MOA: Massive Online Analysis," *J. Mach. Learn. Res.*, vol. 11, pp. 1601–1604, Aug. 2010.

[24] M. Wojnarski, "Debellor: A Data Mining Platform with Stream Architecture," in *Transactions on Rough Sets IX*, pp. 405–427, Springer, 2008.

[25] M. Stonebraker, U. Çetintemel, and S. Zdonik, "The 8 Requirements of Real-Time Stream Processing," *SIGMOD Rec.*, vol. 34, pp. 42–47, Dec. 2005.

[26] P. Domingos and G. Hulten, "Mining High-Speed Data Streams," in *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, (New York, NY, USA), pp. 71–80, ACM, 2000.

[27] N. C. Oza and S. Russell, "Online Bagging and Boosting," in *In Artificial Intelligence and Statistics 2001*, pp. 105–112, Morgan Kaufmann, 2001.

[28] Y. Ben-haim and E. Yom-tov, "A Streaming Parallel Decision Tree Algorithm," in *In Large Scale Learning Challenge Workshop at the International Conference on Machine Learning (ICML*, 2008.

[29] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[30] J. R. Quinlan, *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[31] J. R. Quinlan, "Decision Trees as Probabilistic Classifiers," in *Proceedings of the Fourth International Workshop on Machine Learning*, pp. 31–37, 1987.

[32] J. R. Quinlan, "Unknown Attribute Values in Induction," in *Proceedings of the sixth international workshop on Machine learning*, pp. 164–168, 1989.

[33] G. Hulten, L. Spencer, and P. Domingos, "Mining Time-Changing Data Streams," in *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '01, (New York, NY, USA), pp. 97–106, ACM, 2001.

[34] J. Gama, R. Rocha, and P. Medas, "Accurate Decision Trees for Mining High-Speed Data Streams," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, (New York, NY, USA), pp. 523–528, ACM, 2003.

[35] R. Jin and G. Agrawal, "Efficient Decision Tree Construction on Streaming Data," in *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '03, (New York, NY, USA), pp. 571–576, ACM, 2003.

[36] A. Bifet, G. Holmes, and B. Pfahringer, "MOA-Tweetreader: Real-Time Analysis in Twitter Streaming Data," in *Discovery Science*, pp. 46–60, 2011.