ARINTO MURDOPO, ANTONIO SEVERIEN,

GIANMARCO DE FRANCISCI MORALES, AND

ALBERT BIFET

SAMOA

DEVELOPER'S GUIDE

DRAFT VERSION 0.0.1
YAHOO LABS BARCELONA

Contents

1 *Introduction* 1

2 *Scalable Advanced Massive Online Analysis* 5

3 *Decision Tree Induction* 19

4 *Distributed Streaming Decision Tree Induction* 29

5 *Distributed Clustering Design* 41

6 *Bibliography* 47

1 Introduction

Scalable Advanced Massive Online Analysis (SAMOA) is a framework that includes distributed machine learning for data streams with an interface to plug-in different stream processing platforms.

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.

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

1.1 Big Data V's

Big data is a recent term that has appeared to define the large amount of data that surpasses the traditional storage and processing requirements. Volume, Velocity and Variety, also called the three Vs, is commonly used to characterize big data. Looking at each of the three Vs independently brings challenges to big data analysis.

Volume

The volume of data implies in scaling the storage and being able to perform distributed querying for processing. Solutions for the volume problem are either by using datawarehousing techniques or using parallel processing architecture systems such as Apache Hadoop.

Velocity

The V for velocity deals with the rate in which data is generated and flows into a system. Everyday sensors devices and applications

generate unbounded amount of information that can be used in many ways for predictive purposes and analysis. Velocity not only deals with the rate of data generation but also with the speed in which an analysis can be returned from this generated data. Having realtime feedback is crucial when dealing with fast evolving information such as stock markets, social networks, sensor networks, mobile information and many others. Aiming to process these streams of unbounded flow of data some frameworks have emerged like the Apache! S4 and the Twitter Storm platforms.

Variety

One problem in big data is the variety of data representations. Data can have many different formats depending of the source, therefore dealing with this variety of formats can be daunting. Distributed key-value stores, commonly referred as NoSQL databases, come in very handy for dealing with variety due to the unstructured way of storing data. This flexibility provides and advantage when dealing with big data. Traditional relational databases would imply in restructuring the schemas and remodeling when new formats of data appear.

1.2 *Big Data Machine Learning Frameworks*

Distributed machine learning frameworks, such as Mahout[2] and ML-Base [3], 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)[4] and Vowpal Wabbit[5], 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.

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

[3] Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael J. Franklin, and Michael Jordan. MLBase: A Distributed Machine-Learning System. In *In Conference on Innovative Data Systems Research*, 2013

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

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

1.3 *SAMOA*

Scalable Advanced Massive Online Analysis (SAMOA) is a framework that includes distributed machine learning for data streams with an interface to plug-in different stream processing platforms. SAMOA can be used in two different scenarios; data mining and machine learning

on data streams, or developers can implement their own algorithms an run them on production. Another aspect of SAMOA is the stream processing platform abstraction where developers can also add new platforms by using the API available. With these separation of roles the SAMOA project is divided into SAMOA-API and SAMOA-Platform. The SAMOA-API allows developers to develop for SAMOA without worrying about which distributed SPE is going to be used. In the case of new SPEs being released or the interest in integrating another platform, a new SAMOA-Platform module can be added. The first release of SAMOA supports two SPE that are the state or the art on the subject matter; Apache S4 and Twitter Storm.

The rest of this document is organized as follows. Chapter 2 presents the Scalable Advanced Massive Online Analysis (SAMOA) framework. Chapter 3 explains basic decision tree induction, and the SAMOA implementation of a distributed streaming decision tree induction algorithm is presented in Chapter 4. The details of the distributed clustering algorithm implemented in SAMOA are presented in Chapter 5.

2
Scalable Advanced Massive Online Analysis

Scalable Advanced Massive Online Analysis (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 2.1 discusses the high level architecture and the main design goals of SAMOA. Sections 2.2 and 2.3 discuss our implementations to satisfy the main design goals. Section 2.4 presents the Storm integration into SAMOA.

2.1 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.

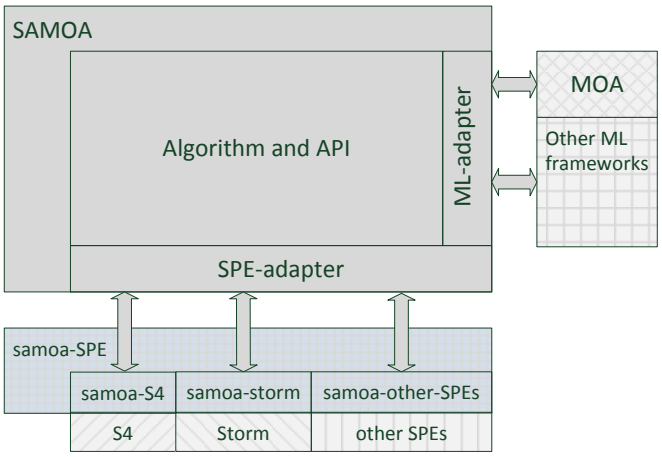


Figure 2.1: SAMOA High Level Architecture

Figure 2.1 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.

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 2.2 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 2.1. 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 2.3 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.

2.2 *SAMOA Modular Components*

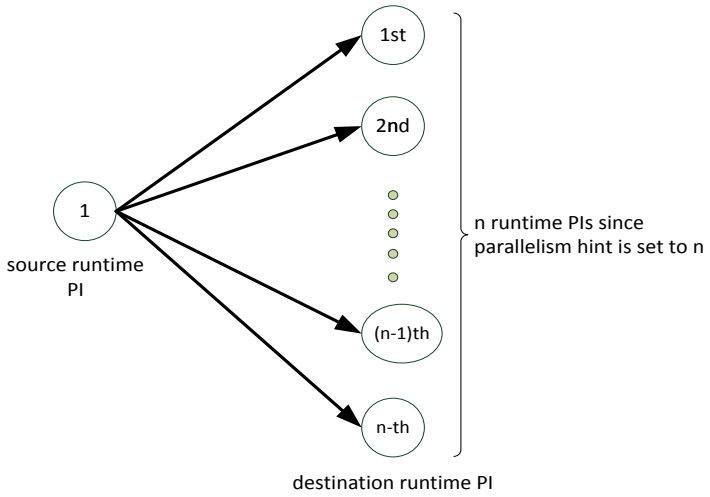
This section discusses SAMOA modular components and APIs that allow ML developers to perform rapid algorithm development. The components are: *processor*, *stream*, *content event*, *topology* and *task*.

Processor

A *processor* in SAMOA is a unit of computation element that executes some part of the algorithm on *a specific SPE*. Processors contain the actual logic of the algorithms implemented by ML developers. *Processing items (PI)* are the internal different concrete implementation of processors for each SPE.

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 2.2. A runtime PI is an actual PI that is created by the underlying SPE during execution. SAMOA dynamically instantiates the concrete class implementation of the PI based on the underlying SPE.

Figure 2.2: Parallelism Hint in SAMOA



A PI uses composition to contain its corresponding processor and streams. 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.

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* 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.

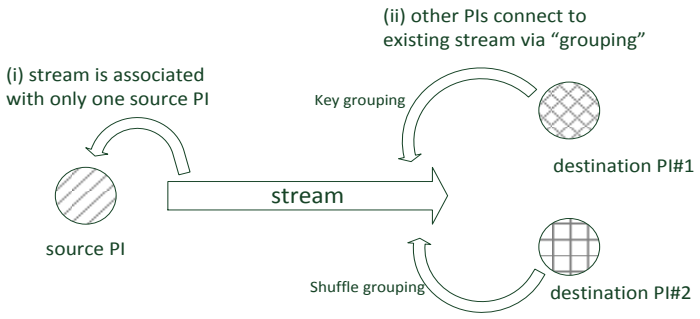


Figure 2.3: Instatiation of a Stream and Examples of Groupings in SAMOA

Refer to figure 2.2, 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.

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.

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.

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.

2.3 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 2.3, 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.

2.4 Storm Integration to SAMOA

This section explains the Storm integration into SAMOA through the aforementioned SPE-adapter layer. To start the explanation, section 2.4 presents some basic knowledge of Storm that related to the SPE-adapter

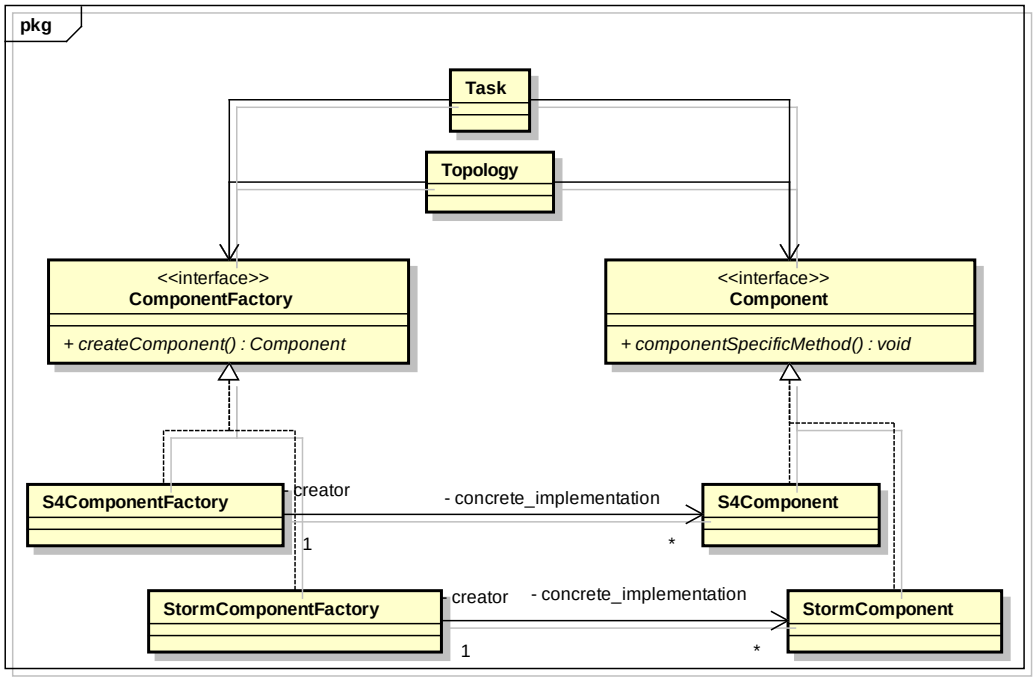


Figure 2.4: Simplified Class Diagram of SPE-adapter Layer

layer. Section 2.4 discusses our proposed design in adapting Storm components into SAMOA.

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".

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

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.

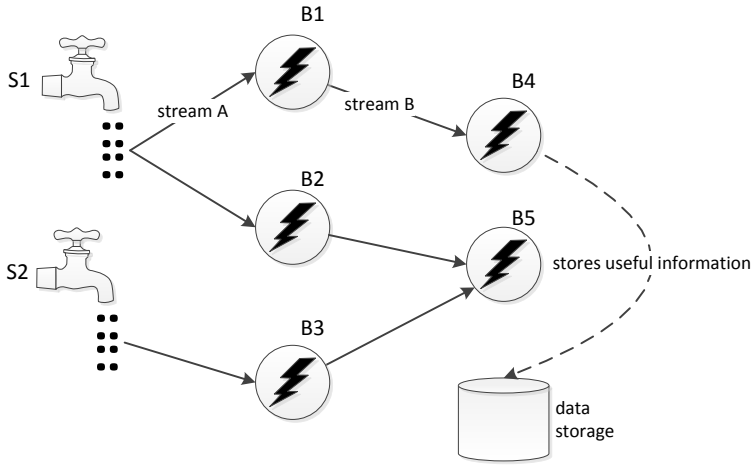


Figure 2.5: Example of Storm Topology

A topology in Storm is a network of spouts, bolts and streams that forms a directed-acyclic-graph. Figure 2.4 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 different bolts. Moreover, one or more streams can arrive to a single bolt. Refer to figure 2.4, 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 2.4 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

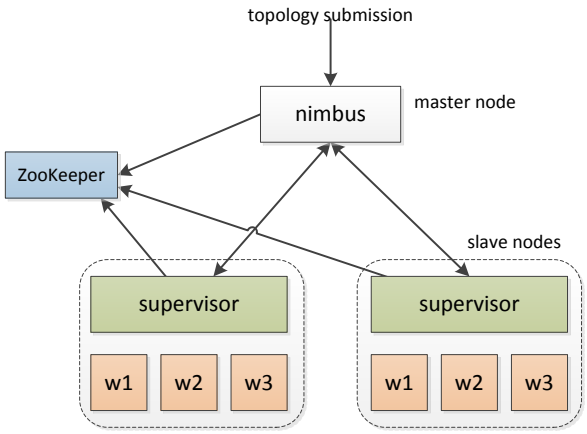


Figure 2.6: Storm Cluster

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 2.4 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.

Design

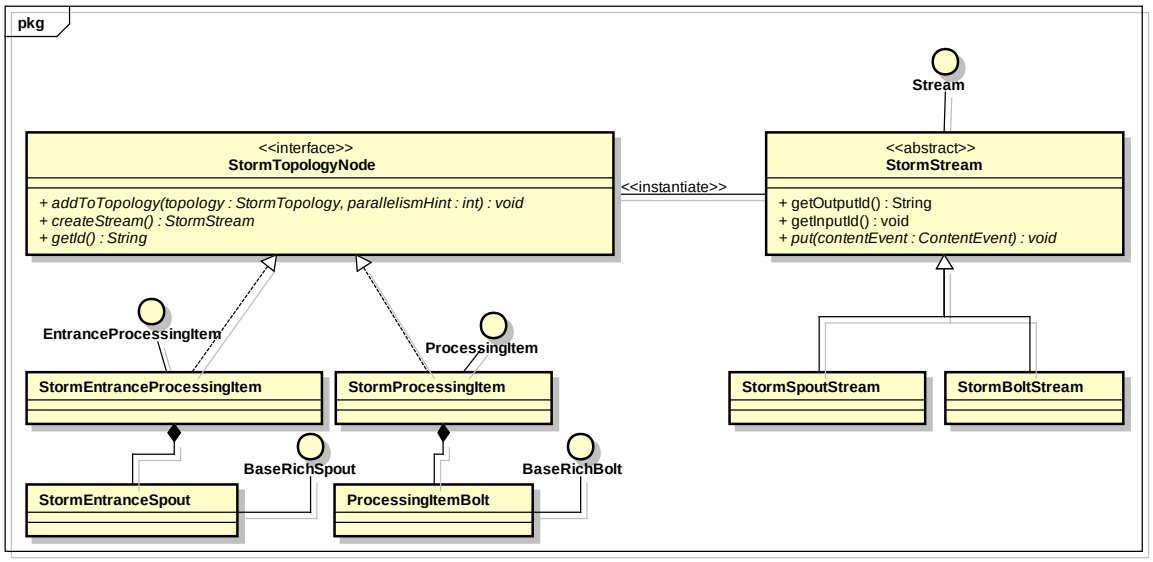


Figure 2.7: samoa-storm Class Diagram

In order to integrate Storm components to SAMOA, we need to establish relation between Storm classes and the existing SAMOA components. Figure 2.4 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 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.

2.5 *Apache S4 Integration to SAMOA*

Apache S4 (Simple Scalable Streaming System) is a distributed SPE based on the MapReduce model and uses concepts of the Actors model to process data events. It uses Processing Elements (PEs) which are independent entities that consumes and emits keyed data events in messages.

Apache S4 is intended to be an open-source general purpose stream computing platform with a simple programming interface. The basic design principles take into account high availability, scalability, low latency and decentralized architecture[2]. The distributed an symmetric design removes the need for a central master node. The design of Apache S4 borrows many concepts from IBM's Stream Processing Core (SPC) middleware, which is also used for big data analysis [3]. The messaging is implemented as push-based where events are emitted into a stream and directed to specific PEs regarding the event keys. The underlying coordination is done by Zookeeper[4] which assigns S4 tasks to physical cluster nodes. An internal view of a S4 processing node is shown in Figure 2.5. The S4 system is built and deployed

[2] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, 2010

[3] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *In Proceedings of the Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, 2006

[4] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association

in a logical hierarchical model. A deployment of S4 will contain the following components:

- Cluster: a logical cluster can be set up to deploy many S4 applications that can be interconnected by different event streams.
- Node: a node is assigned to a logical cluster and will be available to run a S4 application.
- App: the S4 App runs in a Node and encapsulates tasks that consume and produce event streams. An App contains a graph of PEs connected by Streams.
- PE Prototype: is a definition of a computation unit which contains processing logic.
- PE Instance: is a clone of a PE prototype that has a unique key and state.
- Stream: connects PE instances. A PE emits events in a stream and consumes events from a stream.

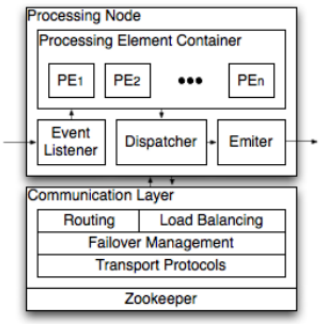


Figure 2.8: S4 processing node internals

The SAMOA-S4 is a built in module specific to plug-in the Apache S4 distributed stream processing platform. Apache S4 is a general purpose, distributed, scalable, fault-tolerant, pluggable platform that allows programmers to develop applications for continuos unbounded stream of data inspired on the MapReduce and Actors model. S4 core items for building a topology are *Processing Elements*, *Streams* and *Events*, which can be directly mapped to SAMOAs *PIs*, *Streams* and *ContentEvent*.

2.6 Example

In practice the use of SAMOA is quite similar as the underlying platforms where the developer implements the business rules in the processing items and connect them by streams. A different aspect is that the logic implemented in the Processor instead of the PI directly; this is a design pattern where the PI is a wrapper allowing the processor to run in any platform. One detail to keep in mind is that once implemented for SAMOA the code can run on any platform which has

an adapter. The following code snippet demonstrates the simplicity of the SAMOA-API. This code builds a simple source entrance PI and one other PI connected by a stream.

```
TopologyBuilder topologyBuilder = new TopologyBuilder();

sourceProcessor = new SourceProcessor();
TopologyStarter starter = new TopologyStarter(sourceProcessor);
topologyBuilder.addEntranceProcessor(sourceProcessor, starter);

Stream stream = topologyBuilder.createStream(sourceProcessor);
Processor processor = new new Processor();
topologyBuilder.addProcessor(processor);
processor.connectInputKey(stream);
```

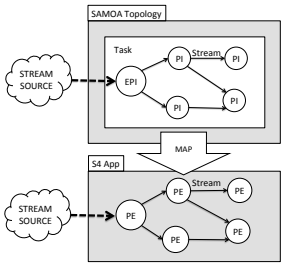


Figure 2.9: Mapping of SAMOA elements to S4 platform.

With this simple API the topology can be extended to create larger graphs of data flow. This model provides a greater flexibility for designing distributed algorithms than the MapReduce model, which can only have sequences of mappers and reducers. Therefore it is easier to adapt common algorithms to S4, consequently to SAMOA. The Figure 2.6 illustrates how the SAMOA framework maps to the S4 system. Notice that the task present in SAMOA is a wrapper for the PIs and stream graph, whereas in S4 this would be implemented directly on the App component.

3
Decision Tree Induction

This chapter presents some backgrounds in decision trees. 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 **??** explains some extensions for the streaming decision tree induction algorithm.

A decision tree consists of *nodes*, *branches* and *leaves* as shown in Figure 3. A *node* consists of a question regarding a value of an attribute, for example node n in Figure 3 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.

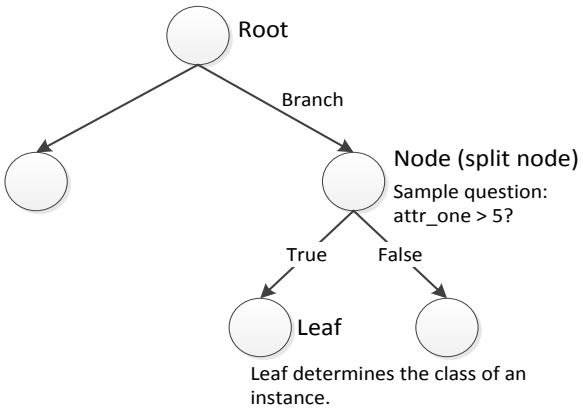


Figure 3.1: Components of Decision Tree

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 and 3.1 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 $a \in D$ **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** $Tree$

Sample Dataset

Given the weather dataset [1] 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

[1] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005

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 3.1. Section 3.1 and 3.1 explain how the algorithm chooses the best attribute.

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 3.1, 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 $info([2,3]) =$

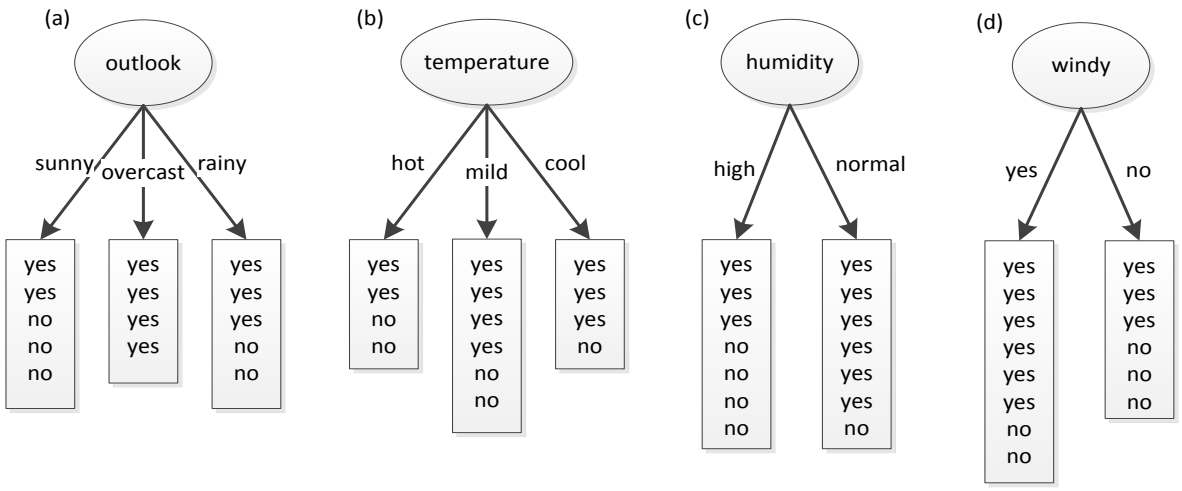


Figure 3.2: Possible Splits for the Root in Weather Dataset

$-p_1 log_2 p_1 - p_2 log_2 p_2$ 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 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 3.1, the algorithm produces these following gains for each split:

- $gain(Outlook) = 0.247$ bits

- $gain(Temperature) = 0.029$ bits
- $gain(Humidity) = 0.152$ bits
- $gain(Windy) = 0.048$ 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).

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 3.1.

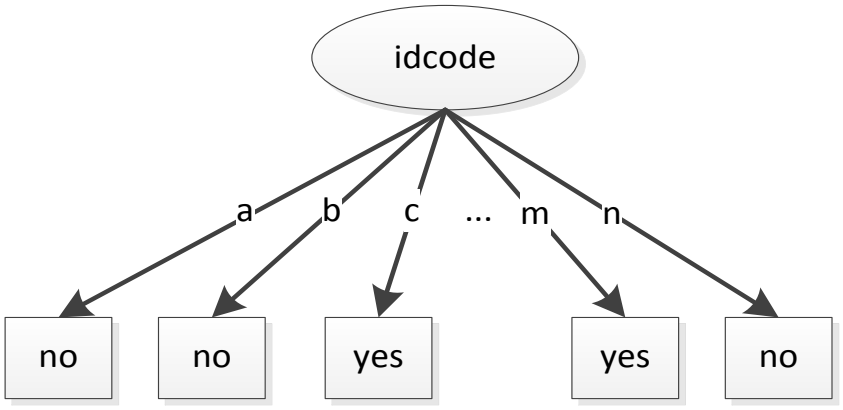


Figure 3.3: Possible Split for *ID code* Attribute

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 3.1, *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 [2]. This section explain further techniques in C4.5 to make the algorithm suitable for production settings.

[2] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993

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 [3] and [4] 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] John Ross Quinlan. Decision Trees as Probabilistic Classifiers. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 31–37, 1987

[4] John Ross Quinlan. Unknown Attribute Values in Induction. In *Proceedings of the sixth international workshop on Machine learning*, pages 164–168, 1989

3.3 *Very Fast Decision Tree (VFDT) Induction*

Very Fast Decision Tree (VFDT) [5] was 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.

[5] Pedro Domingos and Geoff Hulten. Mining High-Speed Data Streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM

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

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**

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 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.

4
Distributed Streaming Decision Tree Induction

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

4.1 Parallelism Type

In this text, 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 to 4.1, 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.

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.

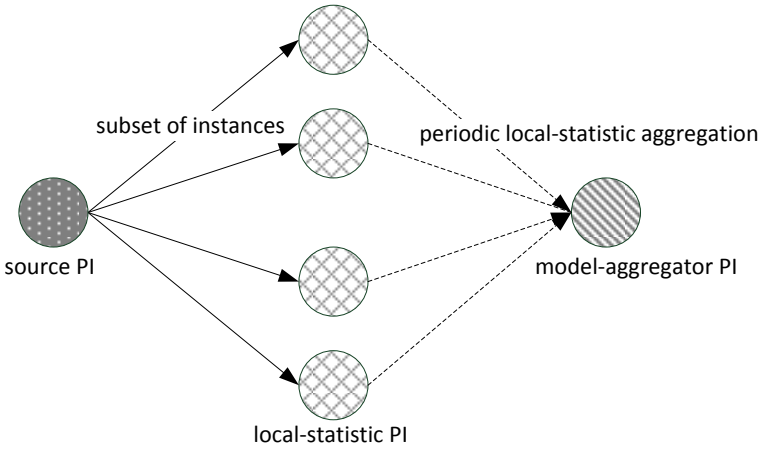


Figure 4.1: Horizontal Parallelism

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.

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 addi-

tional 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.

Vertical Parallelism

Figure 4.2: Vertical Parallelism

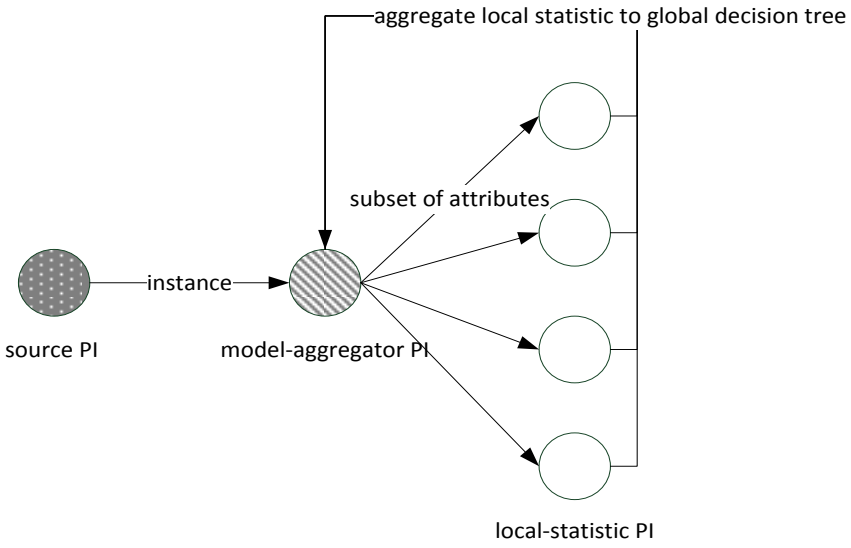


Figure 4.1 shows the implementation of vertical parallelism for distributed streaming decision tree induction. Source processing item serves the same purpose as the one in horizontal parallelism (section 4.1). However, model-aggregator PI and local-statistic PI have different roles compared to the ones in section 4.1.

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

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.

Task Parallelism

Figure 4.1 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 4.1. 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 4.1 shows the induction based on task parallelism. This in-

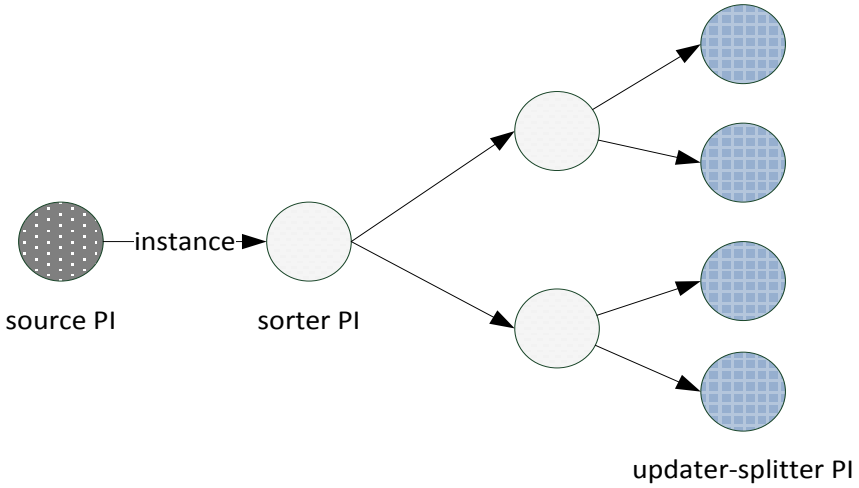


Figure 4.3: Task Parallelism

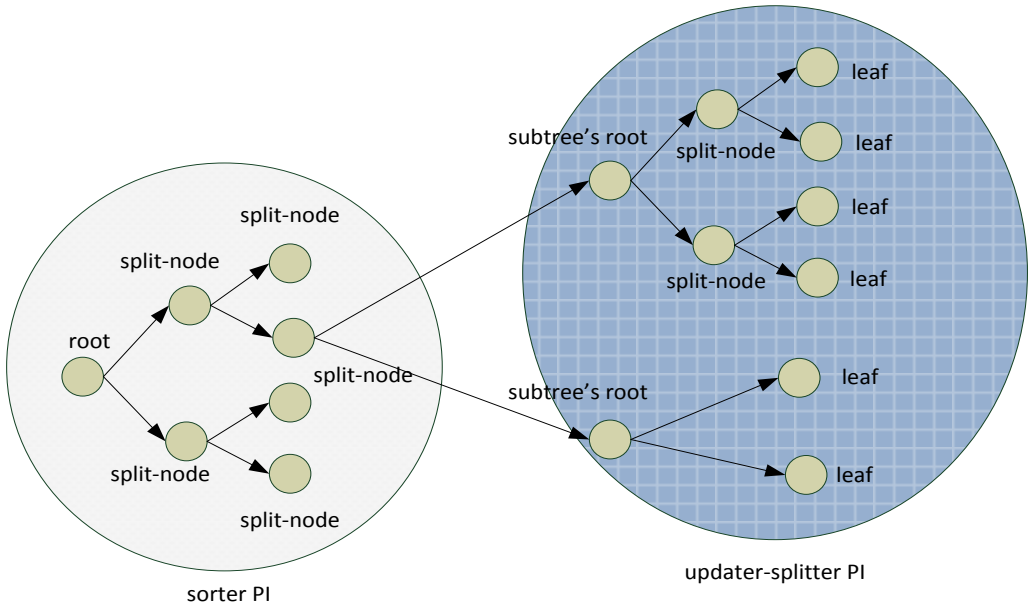


Figure 4.4: Sorter PI in terms of Decision Tree Model

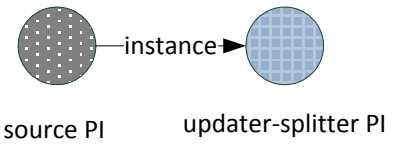
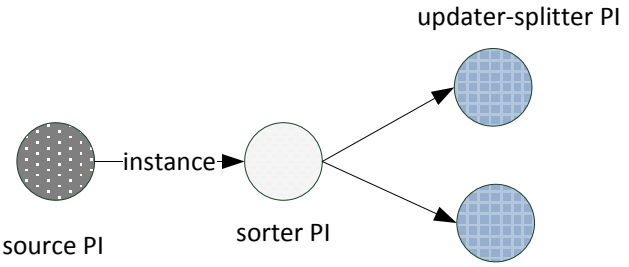


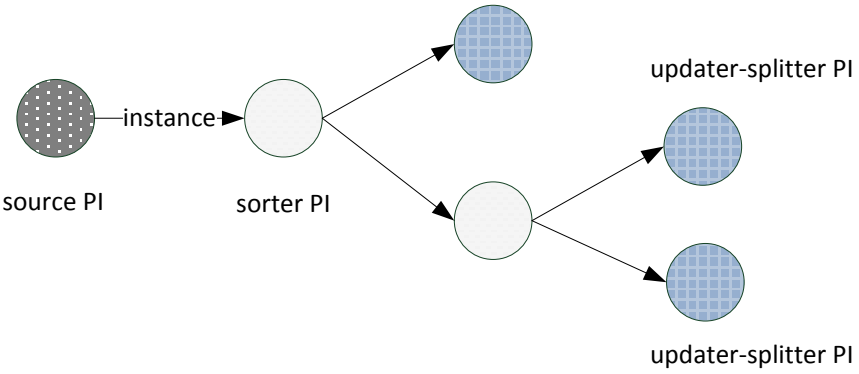
Figure 4.5: Decision Tree Induction based on Task Parallelism

(i) induction starts with an updater-splitter 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



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

duction 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.

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.

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.

The use-case for our distributed streaming tree induction algorithm is to perform document-streaming and text-mining classification. As section 4.1 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.

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 4.2 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.

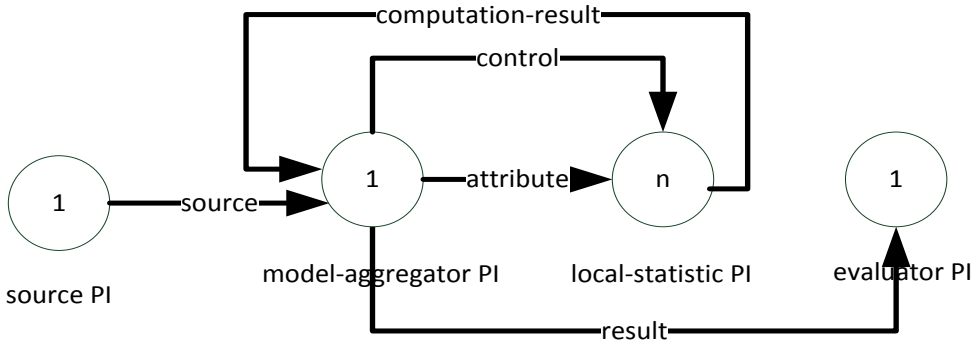


Figure 4.6: Vertical Hoeffding Tree

As we described in section 4.1 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` 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

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**

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`

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

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 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
Distributed Clustering Design

SAMOA has a distributed clustering algorithm based on the CluStream framework presented by Aggarwal et. al in [1]. The decision for choosing CluStream was due to its good clustering accuracy, dataset scalability and handling of evolving data streams. The goal was to design a distributed clustering algorithm in SAMOA that is accurate and performs as good as the CluStream implementations in other machine learning tools, such as MOA.

[1] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003

5.1 CluStream Algorithm

CluStream is a framework for clustering evolving data streams efficiently. As mentioned before, the clustering problem aims to partition a data set into one or more groups of similar objects by using a distance measure. Since stream clustering cannot maintain all the information used due to memory limitation and neither revisit past information, the algorithm has to keep a small summary of the data received. CluStream efficiently deals with this problem by using an online component and an offline component. The online component analyses the data in one-pass and stores the summary statistics, whereas the offline component can be used by the user for querying the cluster evolution in time. In order to maintain these summary statistics CluStream uses a micro-clustering technique. These micro-clusters are further used by the offline component to create higher level macro-clusters [2].

Every algorithm uses specific data structures to work; in stream clustering algorithms it is not different. CluStream uses specific data structures called *Cluster Feature*(*CF*) vectors to summarize the large amount of data on its online phase. This data structure was first introduced in the *BIRCH* algorithm [3]. *CF* vectors are composed of the number of data objects (*N*), the linear sum of data objects (*LS*) and the sum of squares of data objects (*SS*). In more details the *LS* and *SS* are *n*-dimensional arrays. The *CF* vectors conserve the properties of incrementality and additivity, which are used to add points to the *CF*

[2] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003

[3] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases, 1996

vector or merge CF vectors. In CluStream the CF vectors are referred as micro-clusters with additional time components and called $\overline{CFT}(\mathcal{C})$ for a set of points $\mathcal{C}$. The time components are the sum of timestamps (LST) and the sum of squares timestamp (SST). The incrementality and additivity properties are presented bellow.

1. Incrementality: A new arriving point $x \in \mathbb{R}^d$ can be inserted into a CF vector by updating its statistics summary as follows:

$$\begin{gathered} LS \leftarrow LS + x \\ SS \leftarrow SS + x^2 \\ N \leftarrow N + 1 \end{gathered}$$

2. Additivity: A new CF vector can be created by merging two disjoint vectors CF_1 and CF_2 as follows:

$$\begin{gathered} N = N_1 + N_2 \\ LS = LS_1 + LS_2 \\ SS = SS_1 + SS_2 \end{gathered}$$

With these data structures in hand it is possible to compute the means, the radius and the diameter of clusters as it is represented in the equations 5.1, 5.2, 5.3.

$$centroid = \frac{LS}{N} \tag{5.1}$$

$$radius = \sqrt{\frac{SS}{N} - \left(\frac{LS}{N}\right)^2} \tag{5.2}$$

$$radius = \sqrt{\left(\frac{2N * SS - 2(LS)^2}{N(N-1)}\right)} \tag{5.3}$$

The aforementioned properties are used in CluStream to merge or create micro-clusters. The decision on creating or merging is based on a boundary factor that is relative to its distance to the closest cluster. In the case of creating a new micro-cluster the number o clusters has to be reduced by one to maintain a desired amount of cluster. This is done to save memory and can be achieved by either removing or joining old clusters. In this way CluStream can maintain statistical information of a large amount of dominant micro-clusters over a time horizon. Further on these micro-clusters are used on an offline phase of the algorithm to generate higher level macro-clusters.

The offline phase of the algorithm takes as input a time-horizon h and a number of higher level clusters k. The value of h will determine how much history has to be covered by the algorithm and the value k will determine the granularity of the final clusters. Lower h values retrieves more recent information and higher k renders more detailed clusters. In this phase the macro-clusters are determined by

the use of a modified k-means algorithm. The k-means algorithm uses the micro-clusters centroids as *pseudo-points* which are the clustered in higher level clusters. The initialization of the k-means is modified to sample seeds with probability proportional to the amount of points in the micro-cluster. The new seed for a macro-cluster is defined as a weighted centroid for that partition.

5.2 SAMOA Clustering Algorithm

In SAMOA a distributed version of CluStream was implemented using the SAMOA API. The general design of the clustering data flow was divided in three major components represented by the following processing items: data distribution (*DistributrionPI*), local clustering (*LocalClusteringPI*) and global clustering (*GlobalClusteringPI*). In addition to the clustering components a separate module for evaluation was created with a sampling processing item (*SamplingPI*) and an evaluation processing item (*EvaluatorPI*). The evaluation components are used to asses the clustering algorithms by using pre-defined measures, as will be explained in Section 5.3. These components are illustrated in Figure 5.2. This design allows pluggable clustering algorithms to be used in both the local clustering phase and the global clustering phase, where the global clustering uses the output of the local clustering. The definition of each processing item is as follows:

- *DistributionPI*: this processing item can either generate a source stream or receive an external stream to distribute between the *LocalClusteringPI* instances. The distribution will depend in the connection type defined in Section **??**.

- *LocalClusteringPI*: this processing item applies the desired clustering algorithm to the arriving stream and outputs the results as events in the stream connecting the *GlobalClusteringPI*.

- *GlobalClusteringPI*: the global clustering is responsible to aggregate the events from the local clustering and generate a final global clustering, which can be outputted to the *EvaluatorPI* for evaluation or to any output (file, console or other applications).

- *SamplingPI*: the sampling processing item is part of the evaluation module and will only be activated when the evaluation mode is active. It is responsible for sampling the source stream by a chosen threshold from 0 to 1, where 1 forward all the incoming events to the evaluation processing item. Another feature of the *SamplingPI* is that it also generates the ground truth clustering and ground truth evaluation measures to be used as external evaluations. This is only possible when the ground truth is available in the source stream.

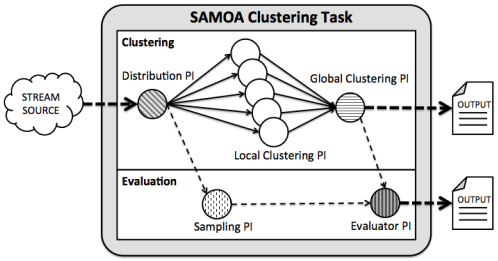


Figure 5.1: SAMOA Clustering Task topology.

- *EvaluatorPI*: the evaluator processing item can apply any desired measures to the clustering result. The measures adopted for SAMOA Clustering are of intra cluster cohesion and inter cluster separation, which are detailed in the next section.

5.3 Evaluation Criteria

Assessing and evaluating clustering algorithms can be a very challenging task. Jain and Dubes mention in 1988 that "the validating of clustering is the most difficult and frustrating part of a cluster analysis" [4]. This statement reflects what Estvill-Castro said about the clustering having different interpretations depending on the perspective. Clustering can be evaluated in two main categories: *external* and *internal* evaluation. The main difference between them is that external validation takes into account the matching against some external structure, whereas the internal only uses its internal attributes for validation. An external structure can be represented by the ground truth, present on synthetic data but not often found on real data. In the literature there are many different measures to validate clustering and a review of some external measures can be found in [5]. For the purpose of this project the measures chosen was that of *cohesion* and *separation* that can both be used as internal and external evaluations by following some conditions. The following items describe in details the measures:

[4] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988

[5] Junjie Wu, Hui Xiong, and Jian Chen. Adapting the right measures for k-means clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 877–886, 2009

- *Cohesion* is the measure of how closely related are the items in a cluster. It is measured by the sum of square error (SSE) between each point x in a cluster C and the cluster mean m_i.

$$SSE = \sum_i \sum_{x \in C_i} (x - m_i)^2 \tag{5.4}$$

- *Separation* measures how distant the clusters are from each other. This is achieved by calculating the between-cluster sum of squares (BSS), taking into account the overall clustering mean m. This measure is also directly related to the weight (amount of points) in a cluster C_i.

$$BSS = \sum_i |C_i|(m - m_i)^2 \tag{5.5}$$

Highly cohesive clusters might be considered better than less cohesive. With these measures some actions can be taken, such as splitting less cohesive clusters and merging not so separate ones. An interesting property of these measures is that they are complementary and their sum renders a constant. Therefore the measures for a clustering with $k = 1$ will give a value of BSS equal to zero, therefore the value of SSE will be the constant. Taking this into account the measures are good candidates for evaluating the SAMOA distributed clustering design. Since the cohesion can only be measured if all the points available, it can only be assessed when the evaluation components are active. On the global phase of the clustering the separation factor (BSS) can be found because the final clustering are weighted clusters. Further on the cohesion (SSE) can be inferred by calculating the difference between the constant and BSS. When the ground truth is available it is possible to find the value of SSE and BSS by using the real clustering centroids. More on the evaluation method will be discussed in the next chapter.

6
Bibliography

[1] Charu C. Aggarwal, Jiawei Han, Jianyong Wang, and Philip S. Yu. A framework for clustering evolving data streams. In *Proceedings of the 29th international conference on Very large data bases - Volume 29*, VLDB '03, pages 81–92. VLDB Endowment, 2003.

[2] Lisa Amini, Henrique Andrade, Ranjita Bhagwan, Frank Eskesen, Richard King, Yoonho Park, and Chitra Venkatramani. Spc: A distributed, scalable platform for data mining. In *In Proceedings of the Workshop on Data Mining Standards, Services and Platforms, DM-SSP*, 2006.

[3] Pedro Domingos and Geoff Hulten. Mining High-Speed Data Streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, KDD '00, pages 71–80, New York, NY, USA, 2000. ACM.

[4] Patrick Hunt, Mahadev Konar, Flavio P. Junqueira, and Benjamin Reed. Zookeeper: wait-free coordination for internet-scale systems. In *Proceedings of the 2010 USENIX conference on USENIX annual technical conference*, USENIXATC'10, pages 11–11, Berkeley, CA, USA, 2010. USENIX Association.

[5] Anil K. Jain and Richard C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[6] Tim Kraska, Ameet Talwalkar, John Duchi, Rean Griffith, Michael J. Franklin, and Michael Jordan. MLBase: A Distributed Machine-Learning System. In *In Conference on Innovative Data Systems Research*, 2013.

[7] L. Neumeyer, B. Robbins, A. Nair, and A. Kesari. S4: Distributed stream computing platform. In *Data Mining Workshops (ICDMW), 2010 IEEE International Conference on*, pages 170–177, 2010.

[8] John Ross Quinlan. Decision Trees as Probabilistic Classifiers. In *Proceedings of the Fourth International Workshop on Machine Learning*, pages 31–37, 1987.

[9] John Ross Quinlan. Unknown Attribute Values in Induction. In *Proceedings of the sixth international workshop on Machine learning*, pages 164–168, 1989.

[10] John Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann, 1993.

[11] Ian H Witten and Eibe Frank. *Data Mining: Practical machine learning tools and techniques*. Morgan Kaufmann, 2005.

[12] Junjie Wu, Hui Xiong, and Jian Chen. Adapting the right measures for k-means clustering. In *Proceedings of the 15th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 877–886, 2009.

[13] Tian Zhang, Raghu Ramakrishnan, and Miron Livny. Birch: An efficient data clustering method for very large databases, 1996.