TECHNICAL UNIVERSITY OF CRETE
SCHOOL OF ELECTRICAL AND COMPUTER ENGINEERING

Dynamic Decision Trees in a Distributed Environment

Zafeiria Moumoulidou

Thesis Committee:

Professor Minos Garofalakis (ECE)
Associate Professor Antonios Deligiannakis (ECE)
Associate Professor Vasilis Samoladas (ECE)

August 2018

ΠΟΛΥΤΕΧΝΕΙΟ ΚΡΗΤΗΣ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Δυναμικά Δένδρα Απόφασης σε Κατανεμημένο Περιβάλλον

Ζαφειρία Μουμουλίδου

Εξεταστική Επιτροπή:

Καθ. Μίνως Γαροφαλάκης (ΗΜΜΥ)

Αναπλ. Καθ. Αντώνιος Δεληγιαννάκης (ΗΜΜΥ)

Αναπλ. Καθ. Βασίλειος Σαμολαδάς (ΗΜΜΥ)

Ιούλιος 2018

Abstract

Decision trees is one the most popular methods in data mining since the intuition behind the models produced is close to human way of thinking. In particular, we focus on the stream processing model which belongs to one of the most realistic schemes since the volume and the production rate of data most of the time make the traditional processing methods ineffective. In this thesis we study the state-of-the-art Hoeffding Tree algorithm designed for building decision tree models over high speed data streams. More precisely, one of the most significant challenges in streaming decision trees, is that each instance of data is processed only once and it is not stored in memory. Thus, any decision regarding the growth of the tree should be made based only on a subset of the original data. In paraller, we study the geometric approach for monitoring threshold functions over distributed streams. In the aformentioned distributed setting, the data needed to compute the values of a function is split among diverse processing sites. So the authors design a monitoring scheme, where the sites do not need to send their data to a central node in order to detect whether the value of a function has crossed a threshold; as a result they manage to reduce the communication load. Finally, we propose a novel distributed algorithm for mining high-speed data streams, based on the state-of-the-art Hoeffding Tree algorithm and the ideas introduced in the geometric method.

Περίληψη

Τα δένδρα απόφασης είναι μια από τις πιο διαδεδομένες τεχνικές ανάλυσης και εξόρυξης δεδομένων αφού τα μοντέλα τα οποία παράγουν συνάδουν με τον ανθρώπινο τρόπο αντίληψης. Ειδικότερα, επιλέγουμε να ασχοληθούμε με το μοντέλο ανάλυσης ροών δεδομένων δεδόμενου ότι αποτελεί ένα από τα πιο ρεαλιστικά σχήματα αφού ο όγκος και ο ρυθμός των δεδομένων στη γενική περίπτωση καθιστά τις κλασσικές μεθόδους επεξεργασίας μη αποδοτικές. Πιο συγκεκριμένα, μελετάμε τον state-of-the-art αλγόριθμο των Hoeffding Trees για την σχεδίαση δένδρων απόφασης. Για τα δένδρα απόφασης σε ροές δεδομένων μία από τις πιο σημαντικές προκλήσεις είναι ότι κάθε δεδομένο το βλέπουμε και το επεξεργαζόμαστε μόνο μία φορά χωρίς να έχουμε τη δυνατότητα να το αποθήκευσουμε στη μνήμη. Έτσι, οποιαδήποτε απόφαση σε σχέση με την ανάπτυξη του δένδρου πρέπει να ολοκληρωθεί βάσει ενός υποσυνόλου του αρχικού όγκου των δεδομένων. Παράλληλα, μελετάμε το σχήμα γεωμετρικής παρακολούθησης της τιμής μιας συνάρτησης πάνω σε κατανεμημένες ροές δεδομένων. Σε αυτό θεωρούμε ότι τα δεδομένα που χρειάζονται για τον υπολογισμό της τιμής της συνάρτησης είναι διαμοιρασμένα σε διάφορους κόμβους επεξεργασίας. Στόχος λοιπόν είναι η σχεδίαση ενός σχήματος παρακολούθησης όπου οι κόμβοι δε χρειάζεται να επικοινώνουν με κάποιο κεντρικό για να εντοπιστεί αν η τιμή της συνάρτησης ξεπέρασε κάποια τιμή έτσι ώστε να μειωθεί ο φόρτος επικοινωνίας. Τέλος, προτείνουμε ένα νέο κατανεμημένο μοντέλο σχεδίασης δενδρικών μοντέλων απόφασης συνδιάζοντας κατάλληλα ιδέες από τις δύο δουλειές.

Acknowledgements

Foremost, I would like to express my sicere gratidute to my thesis advisor Prof. Minos Garofalakis for his continuous guidance and support throught the fulfillment of the thesis. A special thanks for my thesis co-supervisor Prof. Antonios Deligiannakis for always having the time to discuss any problems that come up, his useful suggestions and our fruitful conversations. I also would like to thank Prof. Vasilis Samoladas for accepting to be in my commitee to whom I would also like to expess my admiration for being such an ispriring spirit.

Furthermore, I would like to thank Prof. Athanasios Liavas for inspiring me throughout my studies and introducing me to the world of telecommunications as well as Prof. Karystinos for being so enthusiastic and motivating in class.

I would also like to thank Vasiliki Manikaki for providing me with the code of her thesis and for being willing to help.

A big thanks goes to my family for always being supportive and their emotional support. Last but not least, I am deeply thankful my friends here in Chania for the experiences we earned together and especially Rafail-Athanasios Demertzis for his unconditional support in the last years.

Contents

List of Figures

Chapter 1

Introduction

Data mining and analytics is undoubtedly one of the most active and intriguing areas in the computer science field, which simultaneously affects diverse aspects of everyday life. Actually, human beings themselves through the regular usage of the web, social media, smartphones etc become a significant source of data, which are then processed to reveal hidden knowledge and patterns. In data mining area, classification is one of the basic concepts used; with classification term, given a set of categories (or classes), we refer to the process of identifying to which of those categories a new observation belongs based on a set of characteristics (or features). In particular, the aim of a classification task is given a training sample of data, whose class is known, to produce a general model which will be subsequently used to classify new arriving instances of data. In this thesis we choose to study *Decision Tree* classifiers, which among others like Logistic Regression classifiers, Bayesian classifiers, Support Vector Machines(SVM's), neural networks etc, are more easily human interpretable and therefore one of the most commonly used methods in data mining. Conventional decision tree learners, including ID3 and C4.5 suppose the set of training examples is stored in main memory and is available for recurrent processing at any time. However, nowadays the production rate and the amount of data, make the storage of data prohibitively costly. Thus, in order to fully take advantage of the plethora of data, a decision tree model for mining high speed data streams was proposed by Domingos and Hulten [3], who by using Hoeffding bounds managed to build a classifier which is asymptotically nearly identical to a conventional learner. Furthermore, data stream processing in remote sites is also a frequent phenomenon. As a result, extending the streaming decision tree model to a decentralized setting is of vital importance. Therefore, the fundamental aim of the current thesis is to propose a novel distributed algorithm by adequately incorporating the model introduced in the aforementioned work of Domingos and Hulten and the Geometric Approach for monitoring

the value of a function across distributed data streams which was first proposed by Sharfman et al [5] and will be extensively described later on.

1.1 Thesis Contribution

In this thesis, we choose to study the state-of-the-art algorithm of Hoeffding Trees proposed by Domingos an Hulten for mining high-speed data streams. Since distributed processing is one the most active research areas nowadays, we try to expand their model to work in a distributed and parallel manner. To the best of our knowledge, this is the first attempt to combine the Geometric Method proposed by Sharfman et al with Hoeffding Trees so as to design a novel decentralized setting for mining data streams.

1.2 Thesis Outline

In chapter 2 we describe how decision tree classifiers work along with the basic algorithms for building a conventional decision tree model. In addition, we present the state-of-the-art Hoeffding Tree algorithm for mining high spead data streams while we conclude with the overview of the Geometric Method for monitoring threshold functions over distributed data streams. In chapter 3, we explain how to adequately incorporate the Geometric method with Hoeffding Trees so as to design a distributed setting for data stream mining. In chapter 4 we provide a brief overview of Apache Storm framework [16], [15] along with the implentation details of our solution while in chapter 5 we provide an evaluation of the decentralized monitoring scheme we propose.

Chapter 2

Related Background

2.1 Classification Decision Trees

With classification term we refer to the process of building a model described of a function $y = f(x)$, where x is an example to be classified and $y \in \mathcal{C}$ where $\mathcal{C}$ is a finite set of discrete classes. Building a classifier is a supervised learning process, so we have a set of training example, based on which we try to create a general function model. For decision trees in particular, an **instance** x of the training set can be described as a vector of d **attributes** and a discrete label **class** c. For example, x could be the desciption of the day taking into account the *Weather*, *Humidity* and *Windy* conditions, therefore attributes, and class c corresponds to playing tennis or not. Each attribute, has a set of values; for instance, *Weather* might be *Rainy*, *Overcast* or *Sunny*.

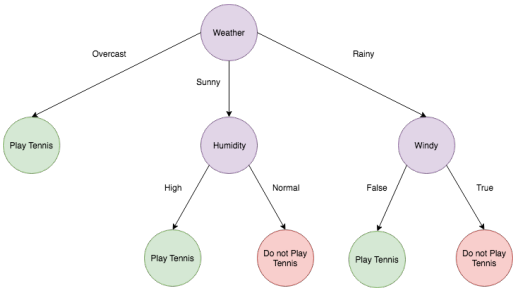


FIGURE 2.1: Decision Tree model

In order to interpret the model a decision tree classifier builds, we represent it as a graph (tree) where each internal **node** corresponds to a test on the splitting attribute, each **branch** of the node is one of the possible answers, thus possible values of the corresponding attribute, and each **leaf** contains a class prediction.

Given the decision tree model in figure 2.1, consinder an instance $x = \{Rainy, Normal, True\}$ for which we want to find a class prediction. Beginning from the root of the tree, the test to be taken is *What is the value of the incoming instance in Attribute weather?* . Since, the answer is Rainy, we follow the second branch and the following test is *What is the value of the incoming instance in Attribute Humidity?* . Since the answer is Normal, we decide that we are playing tennis.

In general, attributes may be discrete or continuous. Discrete attributes are those whose values are particular and come from a finite set of discrete values. For instance, *Weather* is a discrete attribute with categorical values. On the other hand, continuous attributes might take any value in an infinite set of numbers in a range. For example, age, height or time are continuous attributes. Next, we further describe the basic algorithms for producing a decision tree model.

2.1.1 ID3 Algorithm

ID3 Algorithm [1] proposed by Quinlan is the basic algorithm for building a decision tree using discrete attributes. ID3 builds the decision tree model recursively beginning from root and by using a set of labeled examples S. At each step, a leaf is replaced by an internal node when a splitting criterion over some Split Evaluation function 2.1.3 is satisfied. Every time a split occurs, S is divided among the branches generated depending on the attribute chosen for the split. Then, for each partition of S produced, a recursive call is made using the node that corresponds to one of the branches of the internal node as the root of the sub-tree. We then attach the sub-tree produced to the internal node. The terminating condition for the algorithm is either to run out of possible splitting attributes, or to end up with a partition of S with examples that belong to the same class. Finally, regarding the class label at each leaf, we find the cardinalities for each possible class and we assign to the leaf the one with the higher cardinality. The ID3 algorithm is presented below:

Algorithm 1 ID3 Algorithm

```
1: procedure CREATEDT(ExampleSet S, Set X of attributes)
2:     Initialize root R
3:     if S is pure or X = {} then
4:         return R
5:     for each attribute X_i ∈ X do
6:         Calculate split evaluation criteria G_i
7:     Find the G_best, decide to split on X_best and replace leaf with internal node
8:     for each value v_i of X_best do
9:         Split S as S_{v_i}
10:        Tree_{v_i} = CREATEDT(S_{v_i}, X − {X_best})
11:        Attach Tree_{v_i} to internal node
       return Tree
```

2.1.2 C4.5 Algorithm

C4.5 proposed by Quinlan [2] provides extensions to its predecessor ID3 which will be briefly descibed below:

- **Supports Continuous Attributes**
  Recall that continuous attributes are those whose values derive from an infinite set of numbers, e.g Age. In the case of continuous attributes, the split at an internal node will be binary composed of the following tests "$A < \theta$" and "$A \geq \theta$". The question that arises is how the θ parameter will be chosen.

  The solution sketch is as follows; given a sorted order of the distinct values in attribute A, declare as candidate splitting thresholds θ's the midpoint between adjacent values. Then, for each threshold θ_i calculate the split evaluation function G 2.1.3. Note that example set S of a node is divided into 2 subsets; those who satisfy "$A < \theta_i$" and those "$A \geq \theta_i$". The threshold which maximizes G function is the chosen one for the test to be taken on the internal node.

- **Handles Missing Values in Attributes**
  C4.5 provides a solution when an instance of the training set has a missing, therefore equal to '?', value in an attribute. This scenario is an actual problem since if an instance arrives at a node with no value on the test to be taken, then the model does not how to handle it. In addition, missing values affect split evaluation function G computation for the attribute for which we do not know how it is modified by the presence of the instance with missing values. In order to confront this complication, they either choose to simply

ignore the example or to replace '?' with the most frequent value of the attribute for which we have no information.

- **Avoids Overfitting**

  Recall that the terminating condition in ID3 Algorithm 1 was to either run of possible splitting attributes, or to end up with a subset of examples that belong to the same class. However, when following the above strategy the model is built tends to be too complex without the ability to produce general rules. As a result, the classifier is able to classify efficiently the examples in the training set but when examples with unknown class label arrive, the misclassification error is significant.

  In order to adress overfitting, Quinlan suggests to prune the tree after building the tree it the traditional method. So after the decision tree model is constructed, we traverse it bottom-up to find which nodes can be pruned. In particular, for each node we check whether the classsification error of its branches is greater than its error. If that's the case the branch is replaced by the node itself, therefore its subtree is pruned. In general, another method for pruning is to use pre-pruning, where we do not let a leaf to grow unless its split provides the model useful information.

2.1.3 Split Evaluation Function

2.1.3.1 Information Gain

The Split Evaluation Function used in ID3 and C4.5 is entropy and Information Gain since the success criterion of a split is decreasing the impurity for a set of training examples that arrive to a certain leaf. In the case of entropy, we care about choosing the attribute that minimizes its value whereas for Information Gain the best attribute is the one with the higher value. Below, we provide the definition for entropy and Information Gain.

At first, let us define entropy H for random variable $\mathcal{C} = \{c_1, c_2\}$ which corresponds to the discrete classes for our classification problem. Then, $H(\mathcal{C})$ over class distribution $p = \{Prob(C = c_1), Prob(C = c_2)\} = \{p_1, p_2\}$ is:

$$\begin{aligned} H(\mathcal{C}) &= -\sum_{i=1}^{|p|} p_i \log_2 p_i = -p_1 \log_2 p_1 - p_2 \log_2 p_2 \\ &= -p_1 \log_2 p_1 - (1 - p_1) \log_2 (1 - p_1) \end{aligned} \tag{2.1}$$

In the following figure, we observe how entropy behaves in relation to probability. The more pure a set is, therefore ($p_1 \to 1$ & $p_2 \to 0$) **or** ($p_2 \to 1$ & $p_1 \to 0$) respectively, the value of entropy $H \to 0$. Meanwhile, the maximal value (H = 1) occus if $p_1 = p_2 = 1/2$ when there is no win over a class in the set of data.

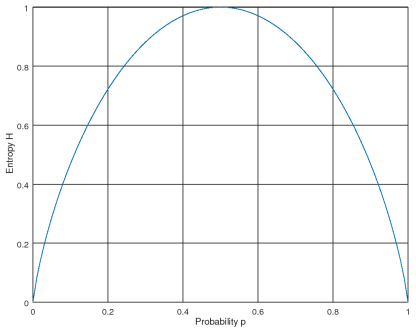


FIGURE 2.2: Entropy over Bernoulli distribution

For the multiclass setting, where $\mathcal{C} = \{c_1, c_2, ..., c_n\}$ and class distribution is defined as $p = \{p_1, p_2, ..., p_n\}$ entropy definition extends to:

$$H(\mathcal{C}) = -\sum_{i=1}^{n} p_i \log_2 p_i \tag{2.2}$$

To sum up, some properties of the entropy function are:

1. $H(\mathcal{C}) \geq 0$, since $p_i \in [0, 1]$

2. The maximal value of $H_{max}(\mathcal{C})$ occrus when $p_1 = p_2 = ... = p_n = \frac{1}{n}$ with $H_{max}(\mathcal{C}) = \log_2 |\mathcal{C}| = \log_2 n$

3. Range R of $H(\mathcal{C}) = \log_2 |\mathcal{C}|$

Below, we provide the definition of Information gain at a leaf l for a certain attribute X_α:

$$G(l, X_\alpha) = H(l) - \sum_{v \in X_{\alpha\_values}} \frac{|l_v|}{|l|} H(l_v) \tag{2.3}$$

where:

- $H(l)$ is the entropy of leaf l
- $H(l_v)$ is the entropy of leaf l_v, the leaf where all instances of l with value v in X_α arrive
- $|l_v|$ the number of instances of leaf l whose value in X_α is v
- $|l|$ the total number of instances of l

Let us illustrate an extensive example for calculating Information Gain at a certain leaf l for all possible splitting attributes. The set of examples S that we use is provided in table 2.1.

Outlook	**Temperature**	**Humidity**	**Windy**	**Class**
Rainy	Hot	High	False	No
Rainy	Hot	High	True	No
Overcast	Hot	High	False	Yes
Sunny	Mild	High	False	Yes
Sunny	Cool	Normal	False	Yes
Sunny	Cool	Normal	True	No
Overcast	Cool	Normal	True	Yes
Rainy	Mild	High	False	No
Rainy	Cool	Normal	False	Yes
Sunny	Mild	Normal	False	Yes
Rainy	Mild	Normal	True	Yes
Overcast	Mild	High	True	Yes
Overcast	Hot	Normal	False	Yes
Sunny	Mild	High	True	No

TABLE 2.1: Set of Instances S

The candidate splitting attributes set is defined as $X = \{Outlook, Temperature, Humidity, Windy\}$, the class distribution as $p_c = \{p_{yes}, p_{no}\}$ while the values for each attribute are:

- $Outlook = \{Rainy, Overcast, Sunny\}$
- $Temperature = \{Hot, Mild, Cool\}$
- $Humidity = \{High, Normal\}$
- $Windy = \{False, True\}$

In the beginning, we calculate the entropy based on S as follows:

$$H(l) = -\sum_{i=1}^{|p_c|} p_i \log_2 p_i = -p_{yes} \log_2 p_{yes} - p_{no} \log_2 p_{no}$$
$$= -\frac{9}{14} \log_2 \left(\frac{9}{14}\right) - \frac{5}{14} \log_2 \left(\frac{5}{14}\right) = 0.94\, bits$$

Next, we find how S is divided over the various branches when a certain atttribute is chosen as the splitting attribute at an internal node. The result of the procedure derives as:

Attribute	Value	Yes	No	Line Sum
Outlook	Rainy	2	3	**5**
	Overcast	4	0	**4**
	Sunny	3	2	**5**
Temperature	Hot	2	2	**4**
	Mild	4	2	**6**
	Cool	3	1	**4**
Humidity	High	3	4	**7**
	Normal	6	1	**7**
Windy	False	6	2	**8**
	True	3	3	**6**

TABLE 2.2: Partitions Of S across different attributes

Our goal is to find which of those attributes best splits our node; so we need to find which one maximizes the Information Gain criterion. We will now extensively show how Information Gain is calculated for Outlook attribute:

- $H(l) = 0.94\, bits$
- $\frac{|l_{Rainy}|}{|l|} \cdot H(Rainy) = \frac{5}{14}\, 0.97 = 0.346$ bits with

$$H(Rainy) = -\frac{2}{5}\, log_2\Big(\frac{2}{5}\Big) - \frac{3}{5}\, log_2\Big(\frac{3}{5}\Big) = 0.97\, bits$$

- $\frac{|l_{Overcast}|}{|l|} \cdot H(Overcast) = \frac{4}{14}\, 0 = 0$ bits with

$$H(Overcast) = -\frac{4}{4}\, log_2\Big(\frac{4}{4}\Big) - \frac{0}{4}\, log_2\Big(\frac{0}{4}\Big) = 0\, bits$$

- $\frac{|l_{Sunny}|}{|l|} \cdot H(Sunny) = \frac{5}{14}\, 0.97 = 0.346$ bits with

$$H(Sunny) = -\frac{3}{5}\, log_2\Big(\frac{3}{5}\Big) - \frac{2}{5}\, log_2\Big(\frac{2}{5}\Big) = 0.97\, bits$$

The Information Gain for Outlook attribute, is computed as:

$$G(l, Outlook) = 0.94 - 2 \cdot 0.346 - 0 = 0.247\, bits$$

Following similar procedure for all attributes the final values for Information Gain in set X are:

1. $G(l, Outlook) = 0.247\, bits$
2. $G(l, Temperature) = 0.029\, bits$
3. $G(l, Humidity) = 0.152\, bits$
4. $G(l, Windy) = 0.048\, bits$

As a result, the best attribute which we choose for the split is Outlook since we care about maximizing the value of Information Gain.

2.1.3.2 Gini Index

Another function which is used as Split Evaluation Function while building decision tree learners is Gini Index. The definition of Gini index at a a leaf l for a certain attribute X_α is:

$$G(l, X_\alpha) = Gini(l) - \sum_{v \in X_{\alpha\_values}} \frac{|l_v|}{|l|} Gini(l_v) \tag{2.4}$$

where:

- $Gini(l)$ is Gini function at leaf l
- $Gini(l_v)$ is Gini function at leaf l_v, the leaf where all instances of l with value v in X_α arrive
- $|l_v|$ the number of instances of leaf l whose value in X_α is v
- $|l|$the total number of instances of leaf l

Given a set of possible classes $C = \{c_1, c_2, ..., c_n\}$, Gini function at a leaf l is defined as follows:

$$Gini(l) = 1 - \sum_{c \in C} (p_c)^2$$

where p_c is the class probability at leaf l.

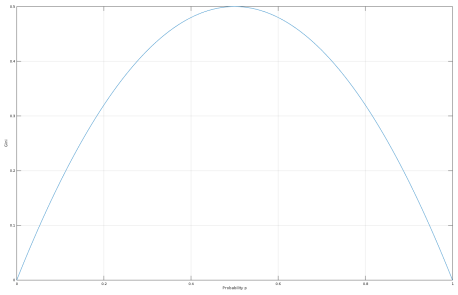


FIGURE 2.3: Gini over Bernoulli distribution

In the following figure, we observe how Gini behaves in relation to Bernoulli distribution.

Again the less pure a set is, the more increase the value of Gini while the maximum value occurs when probabilities are equal.

2.2 Streaming Decision Tree Model

The traditional methods for building decision tree models suppose that the training set used to create a classifier is located in main memory and that is available for recurrent processing. Nevertheless, in the era of Big Data, storage might become prohibitively costly and in conjunction with the rapid data production rate, the need for building a model for mining high speed data streams emerges.

Below we briefly introduce some characteristics of the stream processing model:

- Data streams arrive at a high speed in continuous manner
- Data streams are considered to be infinite
- Memory is limited, no capability of storing the streams
- Each instance of data is processed only once at the moment of arrival (Single Pass over data)
- Small processing time per record

2.2.1 Hoeffding Trees

Domingos and Hulten [3] propose a new system, which is able to process and incorporate a potentially infinite set of data without the need to store any of the instances and with the

ability to spend only a small constant time to process each one of them. Their solution is based on the observation that finding the best attribute at an internal node of the model, requires having processed only a subset of the instaces that would originally arrive to that node. In order to understand when the number of instances that arrived to a certain leaf are sufficient to make a decision regarding which attribute to split on, they use the statistical Hoeffding bound. Consequently, we describe what Hoeffding bound states; Given a random variable r whose range is R [1] and n observations of r over which the mean $\bar{r}$ of the variable is computed, Hoeffding bound guarantees that with probability $1-\delta$ the true mean of r, μ_r , differs from $\bar{r}$ at most by ϵ , where ϵ is defined as:

$$\epsilon = \sqrt{\frac{R^2 \cdot ln(1/\delta)}{2n}} \tag{2.5}$$

In other words, by using this statistical bound we are sure that $Prob\big(|\bar{r} - \mu_r| \geq \epsilon\big) \leq \delta$. Note that the bound depends only on the observations seen, confindence interval $1-\delta$ and range R. Therefore, it is independent of the distribution. However, the main disadvantage is that in the general case different number of instances may be required to reach the same ϵ and δ for data with different distributions.

In the case of decision trees, the aim is to ensure with high confidence that the attribute to be chosen after seeing only a subset of examples at a leaf l will be identical to the one that would have been chosen if the leaf had processed all of its training examples. In the traditional decision tree models, in order to decide which attribute to split on, we need to solve a maximization problem over diverse split evalution functions G's, one for each candidate splitting attribute.

In the streaming setting, since we cannot calculate the true G, we have to make a decision based on the $\bar{G}$ value which is computed over a subset of n observations of G. What Domingos and Hulten state is that by using Hoeffding bound it can be guaranteed that the correct attribute will be chosen at a leaf l. More specifically, suppose that the attribute with the best $\bar{G}$ is X_α whereas X_b is the one with the second best $\bar{G}$. Now, define the difference $\Delta\bar{G} = \bar{G}(X_\alpha) - \bar{G}(X_b) \geq 0$. Then if the leaf has processed n training examples and $\Delta\bar{G} > \epsilon$, Hoeffding bound guarantees with confidence $1-\delta$ that X_α is the correct choice since $\Delta\bar{G}$ differs from the true ΔG at most by ϵ. Thus, based on that observation they built the *Hoeffding Tree* (HT) algorithm, which is presented at 2.

Regarding the computation of the different $\bar{G}$'s, the sufficient information (statistics) each leaf needs to store is the number of examples processed for each attribute, for each value and for

[1] for Entropy $R = log\, c$, where c is the number of classes and for Gini index $R = 1$

each class; Domingos and Hulten refer to these statistics using the term n_{ijk}. Therefore, the total memory required for the model is $O(ldvc)$, where l is the number of leaves of HT, d is the number of attributes, v the maximum number of values per attribute and c the number of classes.

Below, we provide the Hoeffding Tree Algorithm for discrete attributes:

Algorithm 2 The Hoeffding Tree Algorithm

Input: Training examples set **S**, Set **X** of discrete attribute, Split Evaluation Function **G**, δ

procedure HoeffdingTree(S, X, G, δ)
    Initialize HT Tree with leaf l_1 as the root
    Initialize sufficient statistics n_{ijk} at leaf l_1
    Let $\bar{G}_1(X_\varnothing)$ be the $\bar{G}$ for the no-split scenario at l_1.
    for each instance s_i in training set S **do**
        Use HT to sort s_i into a leaf l
        for each attribute in X_l, for each value, for each class **do**
            Update accordingly the statistics n_{ijk} at leaf l
        Increment accordingly the number of instances n_l that leaf l has seen so far
        if l is not pure **then**
            for each attribute X_i in X_l **do**
                Calculate $\bar{G}(X_i)$ using the statistics at leaf l
                Let X_α be the attribute with the highest $\bar{G}$ at leaf l
                Let X_b be the attribute with the second highest $\bar{G}$ at leaf l
                Calculate Hoeffding Bound $\epsilon = \sqrt{\frac{R^2 ln(1/\delta)}{2n_l}}$
                if $\Delta\bar{G} = \bar{G}(X_\alpha) - \bar{G}(X_b) > \epsilon$ and $X_\alpha \neq X_\varnothing$ **then**
                    Replace leaf l with an internal node that splits on X_α
                    for each branch of the split **do**
                        Create a leaf l_m and set $X_m = X - \{X_\alpha\}$
                        Let $\bar{G}_m(X_\varnothing)$ be the $\bar{G}$ for the no-split scenario at l_m
                        Initialize statistics for l_m
    Return HT

According to the *Hoeffding Tree algorithm*, we have a model tha initially starts from a single leaf (root). For each training instance that we get, we use the model to lead the example to the correct leaf l. When l has been reached, the statistics and the number of processed examples in l are updated in addition to calculating Hoeffding bound ϵ. Notice that the value of ϵ is inversely proportional to n_l, which means that the more examples a leaf has processed, the less evidence it needs to decide which attribute is better. Moreover, for every example that arrives into a leaf l, $\bar{G}$ function for all attributes $\in X_l$ is computed and we check whether the splitting condition described above is satisfied. Note, that the solution Domingos and Hulten proposed carries out pre-pruning; thus, no split takes place if the no-splitting scenario is better than the

splitting one. Finally, when the model decides it is time to split a leaf, a new leaf is created for each branch of the split.

2.2.1.1 Very Fast Decision Tree Learner (VFDT)

Domingos and Hulten [3] present VFDT system which is based on the *Hoeffding Tree algorithm* presented above with some refinements. In particular , they notice that the probability that a single example will be the one that would provide the model the confidence it needs in order to conduct a split on a leaf l is negligible. Thus, they suggest to check whether the splitting condition is satisfied at a leaf l after processing a batch of training examples (n_{min}). Furthermore, in order to avoid consuming many examples so as to make a decision at a leaf l whose the two highest $\bar{G}$'s might have similar values, they present a tie threshold τ. As a result, the new splitting condition is $\Delta\bar{G} = \bar{G}(X_\alpha) - \bar{G}(X_b) > \epsilon$ or $\epsilon < \tau$. In other words, when ϵ makes us more confident than we need, regardless of the value of $\Delta\bar{G}$, we are ready to complete the split.

Moreover, they provide a refinement for the *Hoeffding Tree algorithm* so as to work efficiently under memory constraints. In particular, they suggest to deactivate a portion of the leaves, the least promising ones accroding to the error reduction they offer to the model. Nonetheless, they keep monitoring the promise metric for all leaves, either active or not, and in regular basis they check whether a deactivated leaf might have become more promising than an active one so as to replace it. Finally, they suggest to drop from consideration an attribute at a leaf l that is does not look promising considering that data distribution is supposed to be random.

2.2.2 Extensions of Hoeffding Trees and VFDT System

G. Hulten et al [4] propose an extension to the basic algorithm of VFDT for mining concept-adaptive data streams. In particular, CVFDT uses a sliding window model so as to detect changes in the concept of input streams and to evaluate the current decision tree model. In contrary to VFDT where once a split decision is made is permanent, in CVFDT they monitor whether the attribute chosen at an internal node remains the best choice while the stream evolves. In case that another attribute becomes better than the one on which a split was made, that subtree is deactivated and replaced by an alternative subtree which splits on the new best attribute.

Moreover, Gama et al [13] propose $VFDT_c$ an extension of VFDT so as to efficiently handle numeric attributes. More precisely, each possible split is a node in a binary tree while a new

split point is inserted into the tree only when there is a significant proportion od input data with that value. Finally, they propose a method for computing the split evaluation function for every split point in the binary tree while they suggest to use a Naive Bayes classifier into the leaves of the tree model so as to determine the class label instead of assigning to leaf the most representative class.

2.3 Distributed Streaming Decision Trees

In the beginnining, recall that in the distributed streaming setting we consider a set of processing sites $S = \{s_1, s_2, ..., s_n\}$ which process a set of data streams $D = \{d_1, d_2, ..., d_n\}$. The partitioning of the data across the sites might be horizontal or vertical. In horizontal partitioning the data are equally divided across the sites whereas in vertical partitinioning, a site is responsible for monitoring some features, thereby columns of the data. For example, in decision trees an example of vertical partinioning is that each site monitors a subset of the attributes.

Ben-Haim et al [10] propose an approximate streaming parallel algorithm for decision trees with horizontal partitioning. In their solution, there is a set of *slave* processing sites which build histograms over the bucket of data they process so as to maintain the statistics needed for building the decision tree while they have access to the classification tree built till the current moment. Periodically, the sites send their histograms to a *master* site, which then aggregates them accordingly so as to use them for growing the tree model.

In the work of A. Murdopo et al [10], [12] they design a distributed streaming algorithm for learning decision trees using vertical partinioning across data on top of Apache SAMOA, a platform for mining big data streams. More precisely, in the work of Murdopo et al there is a model-aggregator component which is responsible for maintaining and growing the decision tree model, while the processing sites are responsible for maintaining the local statistics for certain attributes for the different leaves based on the updates they receive from the tree model. In other words, each of the processing sites is able to compute the split evaluation function at leaf l for the attribute it monitors independently. Therefore, when the model aggregator decides it is time to check if the splitting condition at a leaf l is satisfied, the sites compute the diverse split evaluation functions for the attributes they monitor, find the two local best and inform the model aggregator. Once the model aggregator receives the information needed, it is ready to find the two global best attributes among the local best and checks whether the splitting condition is satisfied.

2.4 Geometric Function Monitoring Over Distributed Streams

The work of Sharfman et al [5] addresses the problem of monitoring the value of a function over distributed streams. They propose a novel method for detecting whether the value of function has crossed a predetermined threshold T , namely whether $f(\cdot) > T$ or $f(\cdot) < T$. While the problem formulation might be trivial for linear functions that is not the case for the non-linear. Thus, in the general case no conclusions for the value of f can be made from seeing only a subset of its data.

For the problem formulation, consider a set of processing sites $S = \{s_1, s_2, ..., s_n\}$ where a set $D = \{d_1, d_2, ..., d_n\}$ of n data streams arrives and a Coordinator site. Each site $s_i \in S$ maintains a d-dimensional local statistics vector $v_i(t)$. We define the global statistics vector as follows:

$$v(t) = \frac{\sum_{i=1}^{n} w_i v_i(t)}{\sum_{i=1}^{n} w_i} \tag{2.6}$$

where $w_1, w_2, ..., w_n$ are positive weights which can be constant or time-varying. What we want to know at any time is whether $f\big(v(t)\big)$, namely the value of f expressed over the global statistics vector, is above/below a predetermined threshold T.

Sharfman et al noticed that although local f values in sites do not provide any information for the global f value, we can make safe conclusions by monitoring the domain of f. The solution they propose is divided into two phases; the monitoring and the synchronization phase. In the latter, the sites communicate with the coordinator site which gathers all local statistics vector $v_i'(t)$, where $v_i'(t)$ refers to the last sent local statistics vector of site i. As soon as the coordinator receives all $v_i'(t)$'s calculates the current $v(t)$, which we will refer to as estimate vector $e(t)$ and sends it back to the sites. In other words, the estimate vector holds the last known global vector $v(t)$ that was calculated in the last sychronization phase. Along with $e(t)$ and its local statistics vector, each site holds and updates some extra vectors all of which are gathered below:

- **Local statistics vector:** $v_i(t)$
- **Estimate vector:** $e(t) = \dfrac{\sum_{i=1}^{n} w_i v_i'(t)}{\sum_{i=1}^{n} w_i}$
- **Drift Vector:** $u_i(t) = e(t) + \Delta v_i(t)$ where $\Delta v_i(t) = v_i(t) - v_i'(t)$

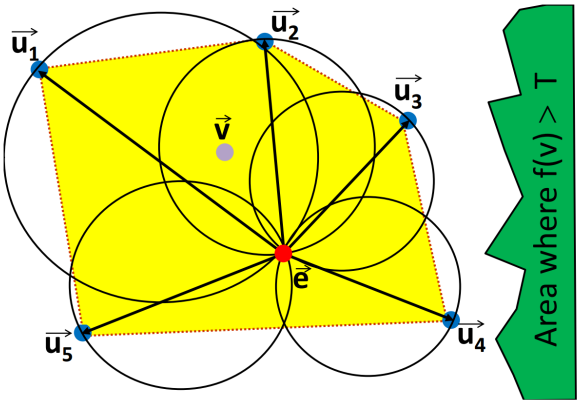


FIGURE 2.4: Estimate vector $e(t)$, Drift vectors $u_i(t)$, current global vector $v(t)$ and Bouncing balls $B(e(t), u_i(t))$, Figure Source: [6]

In [5] they prove that:

$$v(t) = \frac{\sum_{i=1}^{n} w_i u_i(t)}{\sum_{i=1}^{n} w_i} \tag{2.7}$$

which declares that at any time the true global statistics vector lies within the convex hull the drift vectors $\{u_1, u_2, ..., u_n\}$ held by the individual sites form. The geometric interpretation of the current observation can be depicted in figure 2.4. Hence, we know that if we guarantee that all f-values over all the points within the convex hull lie in the admissible area, then $f(v(t))$ also lies there. In order to achieve this, Sharfman et al make use of the following theorem:

Theorem 2.1. *Let $\vec{x}, \vec{y_1}, \vec{y_2}, ..., \vec{y_n}$ be a set of vectors in $\mathbb{R}^d$. Let $Conv(\vec{x}, \vec{y_1}, \vec{y_2}, ..., \vec{y_n})$ be the convex hull of $\vec{x}, \vec{y_1}, \vec{y_2}, ..., \vec{y_n}$. Let $B(\vec{x}, \vec{y_i})$ be a ball centered centered at $\frac{\vec{x}+\vec{y_i}}{2}$ with a radius of $\left\|\frac{\vec{x}-\vec{y_i}}{2}\right\|$. Then $Conv(\vec{x}, \vec{y_1}, \vec{y_2}, ..., \vec{y_n}) \subset \bigcup_{i=1}^{n} B(\vec{x}, \vec{y_i})$.*

In our problem formulation, $\vec{x}$ is the estimate vector $e(t)$ while y_i's are the drift vectors u_i's. So we conclude that any time the convex hull of the drift vectors is bounded by the union of the n balls $B(e(t), u_i(t))$ centered at $\frac{e(t)+u_i(t)}{2}$ with a radius of $\frac{\|e(t)-u_i(t)\|}{2}$. Note that each site i is able to construct its ball independently without the need for communication, since both $e(t)$ and $u_i(t)$ are known. Using the above theorem, a way to check whether all f-values over the convex hull of u_i's lie in the admissible area, therefore $f(v(t))$ lies in there, is to check whether the points of all spheres also lie in the admissible area.

Subsequently, let the area in $\mathbb{R}^d$, with $y \in \mathbb{R}^d$, where $f(\vec{y}) > T$ be green while the area where $f(\vec{y}) \leq T$ be red. Then each site, has to check whether its Ball is monochromatic. As long as all balls are monochromatic, it is guaranteed that the convex hull of u_i's, therefore $v(t)$, has not moved towards the inadmissible area.

Given the above observation, in the monitoring phase each site i checks whether its ball $B(e(t), u_i(t))$ remains monochromatic by calculating the minimal and maximal values of f within B. In the event of a non-monochromatic Ball, the site communicates with the coordinator who then initializes a sychronization process in order to detect whether $f(v(t))$ has crossed the threshold. Note that since $Conv\big(e(t), u_1(t), u_2(t), ..., u_n(t)\big) \subset \bigcup_{i=1}^{n} B(e(t), u_i(t))$ there are cases when a local violation occurs but the convex hasn't crossed the threshold surface.

Monitoring Skylines over Distributed Streams

In the work of Papapetrou and Garofalakis [6] they use the Geometric Method for monitoring a fragmented continuous skyline over distributed streams. They observe that in order to maintain the skyline the following events should be monitored:

1. The entrance of a non-skyline object into skyline
2. The rank between skyline objects, changes in which might cause an object to exit the skyline

In order to monitor the above problems they define pivot points as the midpoint between the f-values of the objects. For the first monitoring problem, they observe that a non skyline object cannot enter the skyline if there is at least one skyline object that is better than it while for the second monitoring problem they observe that not all pairs of skyline objects should be monitored since some introduce tighter constraints than other, which will be violated first.

Chapter 3

Solution Sketch

In this chapter we show how we can incorporate the geometric monitoring method proposed by Sharfman et al [5] so as to design a novel decentralized model for the *Hoeffding Tree algorithm* presented in 2.

3.1 Our Approach

3.1.1 Formulation of Split Evaluation Function

In the beginning, recall that the main concept of the *Hoeffding Tree algorithm* is to decide with high confidence which attribute is the best for a leaf to split on. In order to do so, Domingos and Hulten suggest that each leaf maintain the statistics needed for its split evaluation functions whose values are computed at arrival of an instance x in order to find the top-2 functions. The split occurs when the diferrence between the two functions **crosses** a **threshold** ϵ.

From the procedure described above, we notice that we could use the Geometric Method for monitoring that difference in a distributed and continuous manner. However, we have to express the split evaluation function $\bar{G}$ over the mean of the local statistics vectors of n processing sites as Burdakis and Deligiannakis do in [7] for other functions however. At first, note that the sufficient statistics to calculate $\bar{G}$ function at a leaf l for a discrete attribute X_α are the counts for each class, for each value of X_α.

Now, consider a set of processing nodes $S = \{s_1, s_2, ..., s_n\}$ where a set of data streams $D = \{d_1, d_2, ..., d_n\}$ arrives in a distributed manner. In the beggining, suppose we have a discrete attribute X_α whose values, for reason of simplicity, belong in set $\mathcal{V} = \{\alpha_1, \alpha_2\}$ and the class

labels $\mathcal{C} = \{c_1, c_2\}$. In order to compute $\bar{G}(l, X_\alpha)$ over a global statistics vector $v(t)$, each site i should maintain the following local statistics vector:

$$v_i(t) = \left[n^{c_1}_{\alpha_1,i}(t) \quad n^{c_2}_{\alpha_1,i}(t) \quad n^{c_1}_{\alpha_2,i}(t) \quad n^{c_2}_{\alpha_2,i}(t) \right]^T$$

where $n^{c_1}_{\alpha_1,i}(t)$ refers to the number of the examples which site i read and which arrived at leaf l with value α_1 in attribute X_α and class label c_1. In a similar manner the notation extends to all the vector elements.

In general, $v_i(t)$ will be a **d**-dimensional vector, with $\mathbf{d} = m \times n$, where **m**: number of distinct values for the monitoring attribute ($|\mathcal{V}|$) and **n** : number of distinct classes used in the classification ($|\mathcal{C}|$). The global statistics vector, given the definition in (2.4) is:

$$\begin{aligned} v(t) &= \frac{\sum_i w_i \cdot v_i(t)}{\sum_i w_i} \xrightarrow{w_i = 1 \; \forall \, i} \\ &= \frac{\sum_i v_i(t)}{N} \quad where \, N = |P| \end{aligned}$$

In vector form we get:

$$v(t) = \frac{1}{N} \cdot \left[\sum_i n^{c_1}_{\alpha_1,i}(t) \quad \sum_i n^{c_2}_{\alpha_1,i}(t) \quad \sum_i n^{c_1}_{\alpha_2,i}(t) \quad \sum_i n^{c_2}_{\alpha_2,i}(t) \right]^T \tag{3.1}$$

Our purpose now is to express split evaluation function G over $v(t)$. Next, we show how both Information Gain and Gini Index are formulated given the global statistcs vector $v(t)$.

3.1.1.1 Information Gain as Split Evaluation Function G

Recall that the definition of Information gain at a leaf l for a certain attribute X_α is:

$$\bar{G}(l, X_\alpha) = H(l) - \sum_{v \, \in \, \mathcal{V} = \{\alpha_1, \alpha_2\}} \frac{|l_v|}{|l|} H(l_v) \tag{3.2}$$

where:

- $H(l)$ is the entropy of leaf l
- $H(l_v)$ is the entropy of leaf l_v, the leaf where all instances of l with value v in X_α arrive
- $|l_v|$ the number of instances of leaf l whose value in X_α is v
- $|l|$ the total number of instances of l

Subsequently we express each term of $\bar{G}(l, X_\alpha)$ over the global statistics vector $v(t)$ (3.1). Given class distribution $p = \{Prob(\mathcal{C} = c_1), Prob(\mathcal{C} = c_2)\} = \{p_1, p_2\}$ we have:

$$\begin{aligned} H(l) &= -p_1 \log_2 p_1 - p_2 \log_2 p_2 \\ &= -\frac{\sum n^{c_1}}{|l|} \cdot \log_2 \left(\frac{\sum n^{c_1}}{|l|} \right) - \frac{\sum n^{c_2}}{|l|} \cdot \log_2 \left(\frac{\sum n^{c_2}}{|l|} \right) \\ &= -\frac{v(1) + v(3)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(1) + v(3)}{\sum_{i=1}^{4} v(i)} \right) - \frac{v(2) + v(4)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(2) + v(4)}{\sum_{i=1}^{4} v(i)} \right) \end{aligned} \tag{3.3}$$

Given the class distribution at leaf l_{α_1}, $p_{L_1} = \{Prob(\mathcal{C} = c_1 | \mathcal{V} = v_1), Prob(\mathcal{C} = c_2 | \mathcal{V} = v_1)\} = \{p_{L_{11}}, p_{L_{12}}\}$ we have:

$$\begin{aligned} H(l_{\alpha_1}) &= -p_{L_{11}} \log_2 p_{L_{11}} - p_{L_{12}} \log_2 p_{L_{12}} \\ &= -\frac{\sum n_{\alpha_1}^{c_1}}{|l_{\alpha_1}|} \cdot \log_2 \left(\frac{\sum n_{\alpha_1}^{c_1}}{|l_{\alpha_1}|} \right) - \frac{\sum n_{\alpha_1}^{c_2}}{|l_{\alpha_1}|} \cdot \log_2 \left(\frac{\sum n_{\alpha_1}^{c_2}}{|l_{\alpha_1}|} \right) \\ &= -\frac{v(1)}{\sum_{i=1}^{2} v(i)} \cdot \log_2 \left(\frac{v(1)}{\sum_{i=1}^{2} v(i)} \right) - \frac{v(2)}{\sum_{i=1}^{2} v(i)} \cdot \log_2 \left(\frac{v(2)}{\sum_{i=1}^{2} v(i)} \right) \end{aligned} \tag{3.4}$$

with the weighted value:

$$\frac{|l_{\alpha_1}|}{|l|} \cdot H(l_{\alpha_1}) = \frac{\sum_{i=1}^{2} v(i)}{\sum_{i=1}^{4} v(i)} \cdot \left[-\frac{v(1)}{\sum_{i=1}^{2} v(i)} \cdot \log_2 \left(\frac{v(1)}{\sum_{i=1}^{2} v(i)} \right) - \frac{v(2)}{\sum_{i=1}^{2} v(i)} \cdot \log_2 \left(\frac{v(2)}{\sum_{i=1}^{2} v(i)} \right) \right]$$
$$= -\frac{v(1)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(1)}{\sum_{i=1}^{2} v(i)} \right) - \frac{v(2)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(2)}{\sum_{i=1}^{2} v(i)} \right) \tag{3.5}$$

Similarly at leaf l_{α_2} given the class distribution $p_{L_2} = \{Prob(\mathcal{C} = c_1|\mathcal{V} = v_2), Prob(\mathcal{C} = c_2|\mathcal{V} = v_2)\} = \{p_{L_{21}}, p_{L_{22}}\}$ we have:

$$\begin{aligned} H(l_{\alpha_2}) &= -p_{L_{21}} \log_2 p_{L_{21}} - p_{L_{22}} \log_2 p_{L_{22}} \\ &= -\frac{\sum n_{\alpha_2}^{c_1}}{|l_{\alpha_2}|} \cdot \log_2 \left(\frac{\sum n_{\alpha_2}^{c_1}}{|l_{\alpha_2}|} \right) - \frac{\sum n_{\alpha_2}^{c_2}}{|l_{\alpha_2}|} \cdot \log_2 \left(\frac{\sum n_{\alpha_2}^{c_2}}{|l_{\alpha_2}|} \right) \\ &= -\frac{v(3)}{\sum_{i=3}^{4} v(i)} \cdot \log_2 \left(\frac{v(3)}{\sum_{i=3}^{4} v(i)} \right) - \frac{v(4)}{\sum_{i=3}^{4} v(i)} \cdot \log_2 \left(\frac{v(4)}{\sum_{i=3}^{4} v(i)} \right) \end{aligned} \tag{3.6}$$

with the weighted value:

$$\frac{|l_{\alpha_2}|}{|l|} \cdot H(l_{\alpha_2}) = \frac{\sum_{i=3}^{4} v(i)}{\sum_{i=1}^{4} v(i)} \cdot \left[-\frac{v(3)}{\sum_{i=3}^{4} v(i)} \cdot \log_2 \left(\frac{v(3)}{\sum_{i=3}^{4} v(i)} \right) - \frac{v(4)}{\sum_{i=3}^{4} v(i)} \cdot \log_2 \left(\frac{v(4)}{\sum_{i=3}^{4} v(i)} \right) \right]$$
$$= -\frac{v(3)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(3)}{\sum_{i=3}^{4} v(i)} \right) - \frac{v(4)}{\sum_{i=1}^{4} v(i)} \cdot \log_2 \left(\frac{v(4)}{\sum_{i=3}^{4} v(i)} \right) \tag{3.7}$$

Combining (3.3) (3.5) and (3.7) $\bar{G}(l, X_\alpha)$ expressed over $v(t)$ is:

$$\bar{G}(l, X_\alpha) = H(l) - \sum_{v \in \mathcal{V}=\{\alpha_1,\alpha_2\}} \frac{|l_v|}{|l|} H(l_v) =$$

$$\underbrace{-\frac{v(1)+v(3)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(1)+v(3)}{\sum_{i=1}^{4} v(i)}\right) - \frac{v(2)+v(4)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(2)+v(4)}{\sum_{i=1}^{4} v(i)}\right)}_{\text{H(l)}}$$

$$\underbrace{+\frac{v(1)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(1)}{\sum_{i=1}^{2} v(i)}\right) + \frac{v(2)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(2)}{\sum_{i=1}^{2} v(i)}\right)}_{-\frac{|l_{\alpha_1}|}{|l|} \cdot H(l_{\alpha_1})}$$

$$\underbrace{+\frac{v(3)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(3)}{\sum_{i=3}^{4} v(i)}\right) + \frac{v(4)}{\sum_{i=1}^{4} v(i)} \cdot \log_2\left(\frac{v(4)}{\sum_{i=3}^{4} v(i)}\right)}_{-\frac{|l_{\alpha_2}|}{|l|} \cdot H(l_{\alpha_2})} \tag{3.8}$$

3.1.1.2 Gini Index as Split Evaluation Function G

Recall that the definition of Gini index at a leaf l for a certain attribute X_α is:

$$\bar{G}(l, X_\alpha) = Gini(l) - \sum_{v \in \mathcal{V}=\{\alpha_1,\alpha_2\}} \frac{|l_v|}{|l|} Gini(l_v) \tag{3.9}$$

where:

- $Gini(l)$ is Gini function at leaf$l$
- $Gini(l_v)$ is Gini function at leaf l_v, of leaf l_v, the leaf where all instances of l with value v in X_α
- $|l_v|$ he number of instances of leaf l whose value in X_α is v
- $|l|$ the total number of instances of l

Given a set of possible classes $\mathcal{C} = \{c_1, c_2, ..., c_n\}$, Gini function at a leaf l is defined as follows:

$$Gini(l) = 1 - \sum_{c \in \mathcal{C}} (p_c)^2$$

where p_c is the class probability at leaf l.

We express each term of $\bar{G}(l, X_\alpha)$ over the global statistics vector $v(t)$ 3.1. Given class distribution $p = \{Prob(\mathcal{C} = c_1), Prob(\mathcal{C} = c_2)\} = \{p_1, p_2\}$ we have:

$$\begin{aligned} Gini(l) &= 1 - p_1^2 - p_2^2 \\ &= 1 - \left(\frac{\sum n^{c_1}}{|l|}\right)^2 - \left(\frac{\sum n^{c_2}}{|l|}\right)^2 \\ &= 1 - \left(\frac{v(1) + v(3)}{\sum_{i=1}^{4} v(i)}\right)^2 - \left(\frac{v(2) + v(4)}{\sum_{i=1}^{4} v(i)}\right)^2 \end{aligned} \tag{3.10}$$

Given class distribution at leaf l_{α_1}, $p_{L_1} = \{Prob(\mathcal{C} = c_1|V = v_1), Prob(\mathcal{C} = c_2|V = v_1)\} = \{p_{L_{11}}, p_{L_{12}}\}$ we have:

$$\begin{aligned} Gini(l_{\alpha_1}) &= 1 - p_{L_{11}}^2 - p_{L_{12}}^2 \\ &= 1 - \left(\frac{\sum n_{\alpha_1}^{c_1}}{|l_{\alpha_1}|}\right)^2 - \left(\frac{\sum n_{\alpha_1}^{c_2}}{|l_{\alpha_1}|}\right)^2 \\ &= 1 - \left(\frac{v(1)}{\sum_{i=1}^{2} v(i)}\right)^2 - \left(\frac{v(2)}{\sum_{i=1}^{2} v(i)}\right)^2 \end{aligned} \tag{3.11}$$

with weighted value:

$$\frac{|l_{\alpha_1}|}{|l|} \cdot Gini(l_{\alpha_1}) = \frac{\sum_{i=1}^{2} v(i)}{\sum_{i=1}^{4} v(i)} \cdot \left[1 - \left(\frac{v(1)}{\sum_{i=1}^{2} v(i)}\right)^2 - \left(\frac{v(2)}{\sum_{i=1}^{2} v(i)}\right)^2\right]$$

$$= \frac{\sum_{i=1}^{2} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(1)^2}{\sum_{i=1}^{2} v(i)} \cdot \frac{1}{\sum_{i=1}^{4} v(i)} - \frac{v(2)^2}{\sum_{i=1}^{2} v(i)} \cdot \frac{1}{\sum_{i=1}^{4} v(i)} \tag{3.12}$$

$$= \frac{\sum_{i=1}^{2} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(1)^2 + v(2)^2}{\sum_{i=1}^{2} v(i) \cdot \sum_{i=1}^{4} v(i)}$$

Similarly at leaf l_{α_2} given the class distribution $p_{L_2} = \{Prob(\mathcal{C} = c_1|\mathcal{V} = v_2), Prob(\mathcal{C} = c_2|\mathcal{V} = v_2)\} = \{p_{L_{21}}, p_{L_{22}}\}$ we have:

$$\begin{aligned} Gini(l_{\alpha_2}) &= 1 - p_{L_{21}}^2 - p_{L_{22}}^2 \\ &= 1 - \left(\frac{\sum n_{\alpha_2}^{c_1}}{|l_{\alpha_2}|}\right)^2 - \left(\frac{\sum n_{\alpha_2}^{c_2}}{|l_{\alpha_2}|}\right)^2 \\ &= 1 - \left(\frac{v(3)}{\sum_{i=3}^{4} v(i)}\right)^2 - \left(\frac{v(4)}{\sum_{i=3}^{4} v(i)}\right)^2 \end{aligned} \tag{3.13}$$

with weighted value:

$$\frac{|l_{\alpha_2}|}{|l|} \cdot Gini(l_{\alpha_2}) = \frac{\sum_{i=3}^{4} v(i)}{\sum_{i=1}^{4} v(i)} \cdot \left[1 - \left(\frac{v(3)}{\sum_{i=3}^{4} v(i)}\right)^2 - \left(\frac{v(4)}{\sum_{i=3}^{4} v(i)}\right)^2\right]$$

$$= \frac{\sum_{i=3}^{4} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(3)^2}{\sum_{i=3}^{4} v(i)} \cdot \frac{1}{\sum_{i=1}^{4} v(i)} - \frac{v(4)^2}{\sum_{i=3}^{4} v(i)} \cdot \frac{1}{\sum_{i=1}^{4} v(i)} \tag{3.14}$$

$$= \frac{\sum_{i=3}^{4} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(3)^2 + v(4)^2}{\sum_{i=3}^{4} v(i) \cdot \sum_{i=1}^{4} v(i)}$$

Combining (3.10), (3.12) and (3.14), $\bar{G}(l, X_\alpha)$ expressed over $v(t)$ is:

$$G(l, X_\alpha) = Gini(l) - \sum_{v \in \mathcal{V}=\{\alpha_1,\alpha_2\}} \frac{|l_v|}{|l|} Gini(l_v) =$$

$$\underbrace{1 - \left(\frac{v(1)+v(3)}{\sum_{i=1}^{4} v(i)}\right)^2 - \left(\frac{v(2)+v(4)}{\sum_{i=1}^{4} v(i)}\right)^2}_{\text{Gini(l)}}$$

$$\underbrace{-\left(\frac{\sum_{i=1}^{2} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(1)^2 + v(2)^2}{\sum_{i=1}^{2} v(i) \cdot \sum_{i=1}^{4} v(i)}\right)}_{\frac{|l_{\alpha_1}|}{|l|} \cdot Gini(l_{\alpha_1})}$$

$$\underbrace{-\left(\frac{\sum_{i=3}^{4} v(i)}{\sum_{i=1}^{4} v(i)} - \frac{v(3)^2 + v(4)^2}{\sum_{i=3}^{4} v(i) \cdot \sum_{i=1}^{4} v(i)}\right)}_{\frac{|l_{\alpha_2}|}{|l|} \cdot Gini(l_{\alpha_2})} \tag{3.15}$$

3.1.2 Monitoring the Splitting Condition in Hoeffding Trees

Consider a leaf l in a Hoeffding tree with the set of candidate splitting attributes $X_l = \{X_1, X_2, ..., X_n\} \subseteq \mathbf{X}$, with equality at the root. Given the two attributes $X_\alpha, X_b \in X_l$, which maximize the value of split evaluation function G the splitting condition at l is expressed as: (see 2)

$$\Delta\bar{G}_l = \bar{G}(l, X_\alpha) - \bar{G}(l, X_b) > \epsilon \tag{3.16}$$

Our purpose now is to express the above monitoring problem using the appropriate and the minimum number of thresholds. At first, notice that the problem is divided into two subproblems:

- Find the top-2 $\bar{G}'s$ at leaf l over a set on n attributes
- Check whether $f > \epsilon$

In parallel with the work briefly described in 2.4 for us the skyline is monitoring the top-2 objects ($\bar{G} : object$); thus, we should monitor when an object that was not in top-2, may become better that the first or second. Note however, that in order to become better than the first, an object has to primarily become better from the second, an idea introduced for the skyline monitoring in [6]. Moreover, we want to know when diffence between the top-2 objects crosses ϵ without caring if the order between the first and second changes at any given point.

At first, observe that G functions are defined as $f(v(t)) : \mathbb{R}_+^d \mapsto \mathbb{R}$ and that G is not a monotonic function over its domain. Given an initial ranking between attributes $X_i \in X_l$ and the values of $\bar{G}(l, X_\alpha)$ and $\bar{G}(l, X_b)$, with $\bar{G}(l, X_\alpha) \geq \bar{G}(l, X_b)$, we use their midpoint as a *pivot* point, idea also introduced in [6], between the two with :

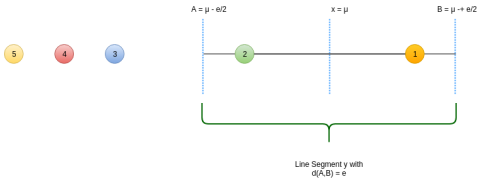
$$\mu = \frac{\bar{G}(l, X_\alpha) + \bar{G}(l, X_b)}{2}$$

Using μ, we define a line segment $\mathcal{Y}$ of length ϵ with bounds $[\mu - \frac{\epsilon}{2}, \mu + \frac{\epsilon}{2}]$. Note that as long as both values belong in $\mathcal{Y}$ then it is guaranteed that $\Delta\bar{G}_l = \bar{G}(l, X_\alpha) - \bar{G}(l, X_b) \leq \epsilon$. As a result, the thresholds for monitoring when the difference between the two best functions crosses ϵ are the following:

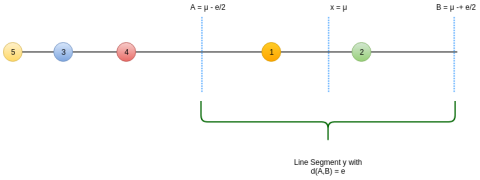
- $\bar{G}(l, X_\alpha) > \mu + \frac{\epsilon}{2}$
- $\bar{G}(l, X_\alpha) < \mu - \frac{\epsilon}{2}$
- $\bar{G}(l, X_b) > \mu + \frac{\epsilon}{2}$
- $\bar{G}(l, X_b) < \mu - \frac{\epsilon}{2}$

Subsequently, we need to define a threshold so as to detect when a non top-2 $\bar{G}(l, X_i)$ with $X_i \in X_l - \{X_\alpha, X_b\}$ might become one. Notice that, all non top-2 $\bar{G}(l, X_i)$ should check at any given point if their value dominates the value of the second best attribute. So, a first approach is to check whether $\bar{G}(l, X_i) > \mu - \frac{\epsilon}{2}$. On second thought, however, we realize that as long as $\bar{G}$ values move in $\mathcal{Y}$ then despite their current rank it is guaranteed that the splitting condition (3.16) at leaf l is not satisfied, since the distance between any points in $\mathcal{Y}$ is not greater than ϵ. Therefore, the optimal threshold for all $X_i \in X_l - \{X_\alpha, X_b\}$ is to check whether $\bar{G}(l, X_i) > \mu + \frac{\epsilon}{2}$.

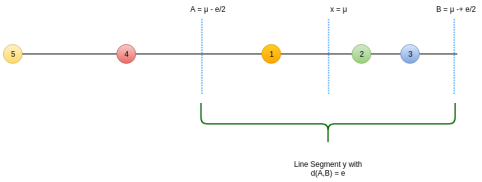
Below, we provide the geometric representation of the ideas described above where $X_l = \{X_1, X_2, ..., X_5\}$ with initial rank $\bar{G}(l, X_1) \geq \bar{G}(l, X_2) \geq ... \geq \bar{G}(l, X_5)$. The notation used for the objects (functions) is $I \in \{1, 2, ..., 5\} = \bar{G}_I = \bar{G}(l, X_I) \in \mathbb{R}$.



(A) The initial rank between $\bar{G}$ functions



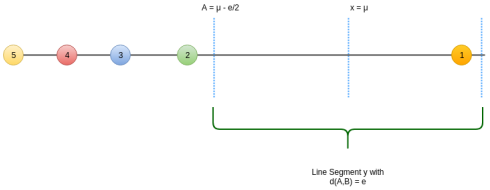
(B) $\bar{G}_1$ and $\bar{G}_2$ change positions but no violation occurs since their difference is guaranteed to be within ϵ



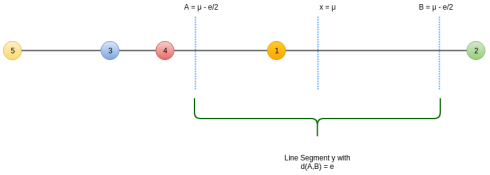
(C) $\bar{G}_3$ becomes better than both $\bar{G}_1$, $\bar{G}_2$ but within $\mathcal{Y}$ it is guaranteed that the difference between **new** X_α (X_3) and **new** X_b (X_2) is less than ϵ

FIGURE 3.1: Geometric Monitoring- No violation Schemes

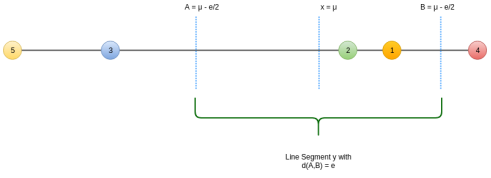
Note that we do not care whether the ranking between any any non-top 2 $\bar{G}$ changes (e.g whether $\bar{G}_5$ gets better than $\bar{G}_3$) . Below, are presented some cases tha would cause a violation to the scheme proposed:



(A) $\bar{G}_2 < \mu - \frac{\epsilon}{2}$ so we have to check the new ranking of X_i's and if the splitting condition is satisfied



(B) $\bar{G}_2 > \mu + \frac{\epsilon}{2}$ so we have to check the new ranking of X_i's and if the splitting condition is satisfied



(C) $\bar{G}_4 > \mu + \frac{\epsilon}{2}$ so we have to check the new ranking of X_i's and if the splitting condition is satisfied

FIGURE 3.2: Geometric Monitoring- Violation Schemes

All in all, the monitoring problems are the following:

- $\bar{G}(l, X_\alpha) > \mu + \frac{\epsilon}{2}$ **or** $\bar{G}(l, X_\alpha) < \mu - \frac{\epsilon}{2}$
- $\bar{G}(l, X_b) > \mu + \frac{\epsilon}{2}$ **or** $\bar{G}(l, X_b) < \mu - \frac{\epsilon}{2}$
- $\bar{G}(l, X_i) > \mu + \frac{\epsilon}{2}$ with $X_i \in X_l - \{X_\alpha, X_b\}$

Therefore at a leaf l, given its set X_l and initial rank between $\bar{G}$'s, the threshold monitoring problems that describe the splitting condition (3.16) are $N = 4 + |X_l| - 2 = 2 + |X_l|$, which is $O(|\mathbf{X}| + 2) = O(|\mathbf{X}|)$ while the theshold monitoring problems for a Hoeffding decision tree will be $O(\sum l \cdot |\mathbf{X}|)$. As long as none of these problems at aleaf l cause a violation it is guaranteed that the splitting condition at l is not satisfied. Nevertheless, if a monitoring problem gives a local violation, the ranking of $\bar{G}$'s at l should be recomputed and we should also check if the splitting condition is satisfied.

Chapter 4

Implementation

For the implementation of the decentralized model for Hoeffding Trees using Geometric Method, we used Apache Storm framework [16] while the programming language we chose was Java. In the following section, we provide a brief overview of Storm:

4.1 Storm Overview

Storm is a free, open sourse distributed realtime computation system designed by Nathan Marz [1] and the BackType mostly used for stream processing. Below, we provide a brief overview of its main concepts.

4.1.1 Storm Components

4.1.1.1 Spouts

Data streams are a possibly infinite sequence of tuples, which arrive at Spout component of Storm. Each tuple is composed of a set of fields, which are primitive data types or any serializable objects. Spouts are the ones responsible for emitting the tuples to the upcoming processing stages via streams. Input sources for Spouts might be plain files, APIs like Twitter etc.

[1] https://github.com/nathanmarz

4.1.1.2 Bolts

The Bolts are the main processing units in Storm Architecture. They receive tuples either from Spouts' or other Bolts' streams. For each tuple, they conduct a type of processing and potentially emit a new tuple to a stream.

4.1.1.3 Topologies

A topology in Storm is a graph which comprises of diverse Spouts and Bolts that communicate the one with another via a set of streams. Each Spout or Bolt might emit to multiple streams, which will be consumed from the following Bolts since it is frequent that topologies have the form of a DAG. Note that the components of a topology might run in parallel while the parallelism level of each unit is defined at declaration. Below, we provide an example of a Topology:

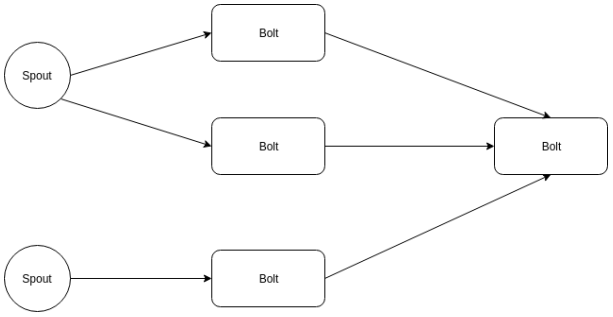


FIGURE 4.1: A topology in Storm

4.1.1.4 Stream Grouping

Considering that the parallelism of a component declares the number of tasks that perform execution, in order to declare how streams are distributed between different tasks, Storm provides multiple grouping methods some of which are presented below:

- **Shuffle grouping**
  Tuples are distributed among different tasks in a round robin fashion while it is guaranteed that tuples will be equally divided between them. For example, if a bolt has parallelism level '2' and the tuples to be processed are 10, it is guaranteed that each task will process 5 tuples.

- **Fields grouping**
  Tuples are distributed based on the value of one or more tuple fields and it reminds us of

a "Group by" functionality. For example, consider a tuple with fields $\{name, age\}$ with fieldsGrouping on name. Then it is guaranteed that the same name will always be led to the same task.

- **All grouping**
  Tuples are sent to all tasks.
- **Global grouping**
  All tuples of the stream result to a single task, the one with the lowest ID.
- **None grouping**
  Currenlty noneGrouping works as shuffleGrouping. However they plan to execute bolts with this grouping in the same thread as the bolt or spout they subscribe from.

4.1.2 Our Topology

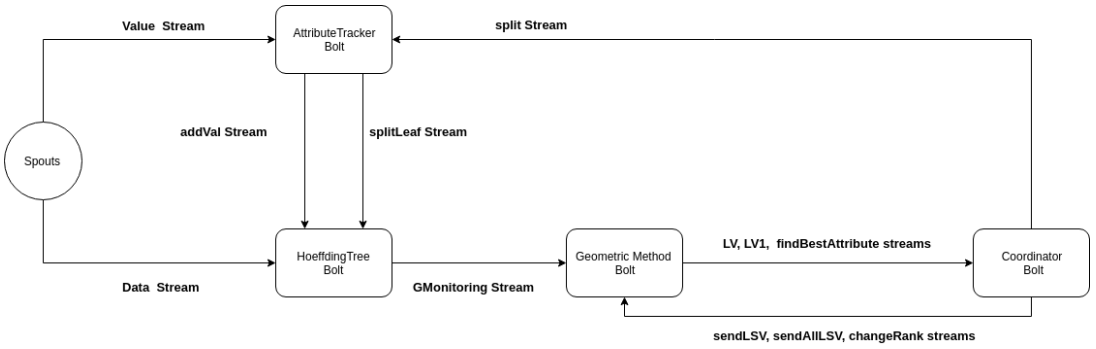


FIGURE 4.2: Topology in Storm

Spouts

In the topology Spouts are responsible for providing the next processing stages with the tuple of information they receive from text files in our case.

In particular, each line of the input file corresponds to a training example which contains the values for each attribute, the class label as well as a siteID identifier so as to simulate the set of processing sites S. For instance, a typical input line for the dataset in table 2.1, would be "1|Rainy|Hot|Normal|False|No", which declares that site 1 receives an instance $x = \{Rainy, Hot, Normal, False\}$ with class label $c = No$. The responsibility of the Spouts, is to produce for each line an object of type "Example", which holds information regarding the site that receives the line, its value on each attribute and its class so as to facilitate its classification

into the tree model. In order to model the attributes, we assign to each one an ID which represents the attribute-column of the dataset their values are located at (e.g Outlook is assigned ID '0') while during the processing we consider that class label is at last column of the input line. Furthermore, a similar procedure is followed for converting a class label to an integer(e.g Yes - '0' while No-'1'). So an Example Object for instance x will contain information as: {< 0, Rainy >, < 1, Hot >, < 2, Normal >, < 3, False >, < class, 1 >, < siteID, 1 > }.

Spouts emit the Example objects produced using shuffleGrouping to *Data* stream, from which *Hoeffding Tree* Bolts consume tuples. In addition, for each attribute they emit the value that the input has to *Value* stream with tuple fields "AttributeID,value" using fieldsGrouping on attribute field. *AttributeTracker* Bolts are the ones who process the tuples from *Value* stream.

AttributeTracker Bolt

Algorithm 3 shows the pseudocode for the processing AttributeTracker Bolt conducts on the tuples it receives.

Algorithm 3 AttributeTracker Bolt

Input: Tuple **T**

```
1: procedure PROCESSTUPLE(T)
2:     if T.source equals Value stream then
3:         if T.value is not monitored for T.AttributeID then
4:             Assign a branchID for the new value
5:             Inform the Hoeffding Tree for the new value on T.AttributeID and its branchID
   via addVal stream
6:     else if T.source equals split stream then
7:         Inform the Hoeffding Tree for the split of T.leafID on T.attribute via splitLeaf stream
```

The responsibility of the *AttributeTracker* Bolt is to monitor the distinct values for each attribute and assign to them an ID corresponding to the branch (child of an internal node) the instances should follow while traversing the model tree. For example, for attribute *Outlook* AttributeTracker would produce the following scheme: {*Rainy* → 0, *Overcast* → 1, *Sunny* → 2}. For each new value they detect for an attribute, they emit a tuple with fields "attributeToBeUpdated, newValue, branchID" to *addVal* stream using allGrouping as all *Hoeffding Tree* Bolts should update their information. Also, note that it is important that this branchID assignment procedure for an attribute should take place in a centralized manner, thus each AttributeTracker

task is responsible for certain attributes, and that all Tree Bolts receive the same scheme so they produce consistent tree models.

Finally, the AttributeTracker Bolt informs the *Hoeffding Tree* Bolts which leaf to split on which attribute when they receive notification from the Coordinator site via *split* stream. They emit tuples with fields "leafID, attribute" to the *splitLeaf* stream using allGrouping. Again, allGrouping is necessary so as to produce the same tree model across the different tasks for the *Hoeffding Tree* Bolts.

Hoeffding Tree Bolt

The *Hoeffding Tree* Bolt is responsible for building the Decision Tree model based on the instances received from Spouts, the information for monitoring the diverse Attributes and split decision events received from *AttributeTracker* Bolt. Below in algorithm 4 we provide a pseudocode for the behaviour of the *Hoeffding Tree* Bolt:

Algorithm 4 Hoeffding Tree Bolt

Input: Tuple **T**

1: **procedure** PROCESSTUPLE(**T**)
2: **for** each attribute in X **do**
3: Initiliaze map <Value, BranchID>
4: Initialize HT Tree with leaf r as the root
5: Initialize List of Leaves L
6: **if** T.source equals *addVal* stream **then**
7: Update branches' ID's for the T.attributeToBeUpdated
8: HT $\leftarrow$ UPDATECHILDREN(r, A, L)
9: Sort any stored Examples using the updated HT
10: **else if** T.source equals *Data* stream **then**
11: Sort example to leaf l where l using HT
12: **if** $l \neq \varnothing$ **then**
13: **for** each attribute $X_i \in X_l$ **do**
14: Emit updates for the local statistics vector of T.siteID for $\bar{G}(l, X_i)$ via *Gmonitoring* stream
15: **else**
16: Store example to process it later
17: **else if** T.source equals *splitLeaf* stream **then**
18: Replace T.leafID by an internal node that splits on T.attribute and remove it from the list of the leaves of the tree
19: **for** each branch of the split **do**
20: Create leaf l_m with $X_m = X - \{T.attribute\}$
21: Add l_m to the list of the leaves of the tree

Algorithm 5 UpdateChilren

Input: Node HT.C of Tree HT, Attribute A for which we monitored new value, List of Leaves L

Output: Tree HT

```
1: procedure UpdateChildren(HT.C, A, L)
2:     if HT.C == leaf then
3:         return HT
4:     else if HT.C splits on A then
5:         Create leaf l_m with X_m = X_{HT.C} − {A}
6:         Add l_m to L
7:         return HT
8:     else
9:         for each branch (node) b_i of internal node HT.C do
10:            return UpdateChildren(b_i, A, L)
       return HT
```

The *Hoeffding Tree* Bolts for each Example object they receive from Spouts, they use the current Tree model to lead the instance to the correct leaf. While traversing the tree model, in order to find how the instance answers the test to be taken, they need to know on which attribute each internal nodes splits on as well as the value of the instance on that attribute(which is stored inside Example object). Then, for the current Example and for each attribute in X_l X they emit a tuple via *Gmonitoring* stream with fields "siteID, numOfInstances, timestamp, functionG, value, class, n" using fieldsGrouping on "siteID" field. More specifically, they inform the Bolts implementing the Geometric Monitoring how the current instance x affects the local statistics vector for each function $\bar{G}(l, X_j)$ where $X_j \in X_l$ of the site s_i which processed x. In particular, notice that each instance affects only one index in $v_i(t)$, which is equal to index = value ·#classes + class. Since they continuously send the updates to the the Bolts implementing the Geometric Monitoring, numOfInstances = 1. Otherwise, for each leaf they should aggregate for each value for each class for each node how many examples they have seen since the last time they emitted to *Gmonitoring* stream. Regarding n, it is a variable that holds the number of examples a leaf l since the last time we sent the updates for the local statistics vector so that the bolts for geometric monitoring update n_l accordingly. Thus, for each example that arrives at leaf l they send a batch of updates $|X_l|$, but n_l is incremented only once.

Furthemore, note that since we depend on a streaming model, we assume that the values of the attributes are not known well in advance. Thus, there is a chance that the instance to

be processed might have a value on attribute of which the model was not aware of the time a split decision was made. As a result, while traversing the tree the corresponsing branch does not exist. In order to support this event, we assume that the *Hoeffding Tree* Bolts store such instances so as to process them when they get a new-value event from *AttributeTracker* Bolt which is the responsible one for monitoring the attributes. When a new-value event occurs, they update the model the Bolt stores for the branches' ID's for the attribute to be updated. Moreover, they call **UpdateChildren** procedure 5 , where they recursively traverse the tree until they find all the nodes that split on the attribute for which a new value has been monitored and add a new branch for it. The terminating condition is either reach a leaf or find the parent of the sub-tree that splits on the attribute with the new value. Finally, when a *splitLeaf* event occurs, they find the corresponding leaf from the list and split it on the attribute they are told to and assign a unique ID to all of its children.

Geometric Method Bolt

For the implementation of the Geometric Monitoring of the Split Evaluation Functions at a leaf l, we use the system that Vasiliki Manikaki developed on Storm in her diploma thesis [8]. Recall that the distributed computing environment comprises of a set of processing sites $S = \{s_1, s_2, ..., s_n\}$ which receive a set of data streams $D = \{d_1, d_2, ..., d_n\}$ and a coordinator site. Below, we will briefly describe Manikaki's approach along with the additions we needed so as to implement the monitoring scheme described in 3.1.2.

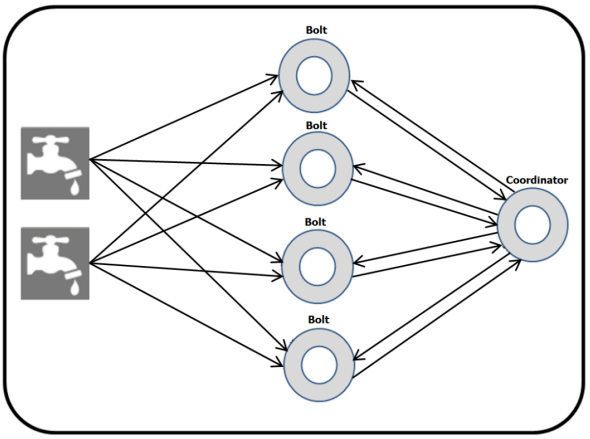


FIGURE 4.3: Manikaki's Topology in Storm

In our topology, Spouts are eliminated since the tuples the Geometric Method Bolts receive come from the *GMonitoring stream*. Thus, the Bolts' main responsibility is to implement the Geometric Method for monitoring whether a function f will cross a threshold T. The Bolts are

responsible for processing the data received from *GMonitoring stream*, check whether a local violation occurs at a site after the receipt of the updates for a certain function and inform the Coordinator if a violation is detected. In addition, in synchronization phase for a monitoring problem they should send their computed local statistics vectors to the Coordinator. The system developed by Manikaki supports monitoring multiple functions, while each Bolt is responsible for representing a subset of the processing S sites. For example for a set S, with $|S| = 4$ and a parallelism level in Bolts equal to 2, Bolt 1 will be responsible for two sites and Bolt 2 for the other two. In particular, each Bolt maintains a HashMap of the functions it monitors, while each function object contains all the information (e.g local statistics vector, estimate vector etc) needed for the geometric monitoring distributed among the processing sites S. In our problem formulation, each site might receive examples for multiple leaves. Therefore, the Bolts should maintain a Hashmap of the necessary functions for each leaf for which a site that they represent receives an example. Below we will describe the geometric monitoring of split evaluation functions at a certain leaf l.

At first, each example that a site s_i process is classified to a certain leaf l using the tree model developed by Hoeffding Tree Bolts who are responsible for sending the necessary information to the Geometric Method Bolts so as to monitor the G functions at l. More specifically, for each instance that arrives into l, Geometric Method Bolts receive $|X_l|$ tuples containing the following information:

1. The siteID that processed the example
2. The value for the function
3. Function F, whose name is generated as "function.InfoGain_attributeID_leaf"
4. The valueID the example had in the F.attributeID
5. The label classID of the example

In order to illustrate how valueID and classID are used in our implementation, consider the dataset in 2.1 and an instance $x = \{Rainy, Mild, Normal, True, Yes\}$ which arrives at site s_1 and arrives to l with ID=0. Below, we show how the values for each attribute and classes are mapped to IDs :

- Outlook (Attribute_0) : { Rainy → 0, Overcast → 1, Sunny → 2 }
- Temperature (Attribute_1) : { Hot → 0, Mild → 1, Cool → 2 }

- Humidity (Attribute_2) : { High → 0, Normal → 1 }
- Windy (Attribute_3) : { False → 0, True → 1 }
- Class labels : { Yes → 0, No → 1 }

Combining the above information, the tuples that the Geometric Method Bolt receive so as to update the local statistics vector of site s_1 for each attribute in $X_l = X$ for x instance are:

1. "1|1|function.InfoGain_0_0 |0|0"
2. "1|1|function.InfoGain_1_0 |1|0"
3. "1|1|function.InfoGain_2_0 |1|0"
4. "1|1|function.InfoGain_3_0 |1|0"

The Bolt for each tuple, thus for each function $\bar{G}$, updates the local statistics vector v_1 since $siteID = 1$. Notice, that each instance affects somehow only one count of those stored in $v_1(t)$, index of which is defined as $j = valueID \cdot \#numOfClasses + class$. For instance , j for $\bar{G}(l, Attribute\_1)$ is equal to $j = 1 \cdot 2 + 0 = 1$, with $v_1(j) += 1$ when continuously sending the updates each instance produces for the $\bar{G}$ functions. Above, we described the procedure for updating the local statistics vector at a processing site s_i for a function $\bar{G}$. To sum up, the Bolts for every tuple they receive, they update the appropriate local statistics vector and check whether a local violation occurs at the site which processed the tuple.

In our monitoring scheme, the threshold chosen depends on the rank of the function for which we received an update. At initialization, we randomly define the two best functions, with rank '1' and '2' respectively, across different $\bar{G}(l, X_i)$ while we set the *pivot* point $\mu_l = 0$ between the 2 best $\bar{G}$'s since we do not have any statistics. In general, the Bolts for every function, despite its rank, have to check whether a local violation occurs for the monitoring problem $\bar{G} > \mu_l + \epsilon_l/2$. If no local violation occurs, and the rank of the function for which the Bolts received an update is equal to '1' or '2', we also need to check whether a local violation occurs for the monitoring problem $\bar{G} < \mu_l - \epsilon_l/2$. Note that ϵ_l changes for every new example a site receives and the Bolts recalculate it accordingly based on the equation (2.5). If they detect a local violation they inform the Coordinator Bolt via *LV* streams with a tuple containing information about the site, the function and the leaf for which the violation occured.

Furthermore, the Bolts might receive a message from Coordinator requesting the local statistics vector of a function $\bar{G}$ at l via *SendLSV* stream. In that case, the Bolts aggregate the

local statistics vectors of function $\bar{G}$ for the sites they are responsible and send it to the Coordinator Bolt via *LV1* stream. In addition, each time a violation for a unction $\bar{G}$ at l occurs the Coordinator calculates the new estimate vector and sends it back to the Bolts via *GlobalV* stream.

Subsequenlty, in our scheme in case a monitoring condition is satisfied, thus local violation was not a false positive, the Bolts receive a message from Coordinator requesting their local statistics vector for all $\bar{G}$ functions at leaf l via *SendAllLSV* stream. Then, the Bolts after they receive the batch of updates for the instance that caused the violation, for each $\bar{G}$ at leaf l aggregate the local statistics vectors for each site they are responsible and send it to the Coordinator via *findBestAttribute* stream. Finally, they may also receive a message from Coordinator via *changeRank* stream so as to update the ranking for the $\bar{G}$ functions at l along with the value of μ_l.

Below we provide the pseudocode describing the procedure described:

Algorithm 6 Geometric Method Bolt

Input: Tuple **T**

procedure PROCESSTUPLE(**T**)
  if T.source equals *GMonitoring* stream **then** ▷ instance classified into leaf l
    $\bar{G}_l \leftarrow$ T.function
    site $\leftarrow$ T.siteID
    Update Local Statistics Vector for $\bar{G}_l$
    $\epsilon_l \leftarrow \sqrt{\frac{R^2 \cdot ln(1/\delta)}{2 \cdot n_l}}$
    Threshold $t \leftarrow \mu_l + \frac{\epsilon_l}{2}$
    if site.hasLocalViolation for $\bar{G}_l > t$ **then**
      Report local violation to Coordinator via *LV* stream
    else if $\bar{G}_l$.rank == '1' or $\bar{G}_l$.rank == '2' **then**
      Threshold $t \leftarrow \mu_l - \frac{\epsilon_l}{2}$
      if site.hasLocalViolation for $\bar{G}_l < t$ **then**
        Report local violation to Coordinator via *LV* stream
  else if T.source equals *SendLSV* stream **then**
    $\bar{G}_l \leftarrow$ T.function
    Send local statistics vector for of $\bar{G}_l$ via *LV1* stream
  else if T.source equals *GlobalV* stream **then**
    $\bar{G}_l \leftarrow$ T.function
    Update estimate vector for $\bar{G}_l$
  else if T.source equals *SendAllLSV* stream **then** ▷ Time to check whether Splitting condition satisfies
    for each $\bar{G}_l$ **do**
      Send local statistics vector via *findBestAttribute* stream
  else if T.source equals *changeRank* stream **then** ▷ Rank among $\bar{G}$'s and μ_l changed
    $\mu_l \leftarrow$ T.μ_{new}
    for each $\bar{G}_l$ **do**
      if $\bar{G}_l$ is in top-2 $\bar{G}$'s **then**
        update accordingly $\bar{G}_l$.rank
      else
        $\bar{G}_l$.rank $\leftarrow 0$

Coordinator Bolt

The Coordinator Bolt is responsible for handling the local violations that occur at diverse leaves l. Below, we will explain the behaviour of the Coordinator Bolt when a local violation at a certain leaf l occurs. To begin with, the Coordinator Bolt might receive a local violation event via *LV* stream for a function $\bar{G}_l$. In that case, it requests from the Geometric method Bolts via *SendLSV* stream using allGrouping, to send their local statisics vector for $\bar{G}_l$. When it receives the local statistics vector from all Bolts, it calculates new estimate vector for $\bar{G}_l$, and checks whether its value given the updated estimate vector satisfies the monitoring problem condition. Recall, that for all $\bar{G}_l$'s despite their rank we have to check whether $\bar{G}_l > \mu_l + \frac{\epsilon_l}{2}$. Then for the two best, if the above condition is not satisfied, we have to check whether $\bar{G}_l < \mu_l - \frac{\epsilon_l}{2}$. In the event that a monitoring condition is true, the Coordinator via *SendAllLSV* stream using allGrouping informs the Bolts that they should send the local statistics vectors for all $\bar{G}_l$ as soon as they receive the batch of updates for a training example (recall that an instance produce $|X_l|$ updates). Finally, when the Coordinator receives the local statistics vector from all Bolts for all $\bar{G}_l$'s, it calculates for each one of them the true estimate vector, finds the ranking order between them and if the leaf is not pure it checks whether the splitting condition is satisfied.

Below we provide a pseudocode for the procedure described:

Algorithm 7 Coordinator Bolt

Input: Tuple **T**

1: **procedure** PROCESSTUPLE(**T**)
2: **if** T.source equals *LV* stream **then**
3: $\bar{G}_l \leftarrow$ T.function
4: Request from Bolts to send their local statistics vector for $\bar{G}_l$ via *SendLSV* stream
5: **else if** T.source equals *LV1* stream **then**
6: $\bar{G}_l \leftarrow$ T.function
7: Update estimate vector $e(t)$ for $\bar{G}_l$ using the local statistics vector received
8: **if** all local statistics vectors arrived **then**
9: Calculate value $myVal$ of $\bar{G}_l$ using estimate vector
10: **if** $myVal > \mu_l + \frac{e_l}{2}$ **then**
11: Request from Bolts to send local statistics vector for all $\bar{G}_l$'s via *SendAllLSV* stream.
12: **else if** $\bar{G}_l.rank ==$'1' or $\bar{G}_l.rank ==$'2' **then**
13: **if** $myVal < \mu_l + \frac{e_l}{2}$ **then**
14: Request from Bolts to send local statistics vector for all $\bar{G}_l$'s via *SendAllLSV* stream.
15: **else if** T.source equals *findBestAttribute* stream **then**
16: **if** local statistics vector for all $\bar{G}_l$ arrived from all Bolts and l is not pure **then**
17: **for** each $\bar{G}_l$ **do**
18: Calculate its value val_j using its estimate vector $e_j(t)$
19: Find the two best $\bar{G}_l$, $\bar{G}(l, X_\alpha)$ & $\bar{G}(l, X_b)$
20: **if** $(\Delta\bar{G} = \bar{G}(X_\alpha) - \bar{G}(X_b) > \epsilon_l$ or $\epsilon_l < \tau)$ and $X_\alpha \neq X_\varnothing$ **then**
21: Decide to split l on X_α
22: Inform the model of the current decision via *split* stream
23: **else**
24: Calculate new $\mu_l = \frac{\bar{G}(l, X_\alpha) + \bar{G}(l, X_b)}{2}$
25: Inform the Bolts for new new μ_l and the new top-2 objects via *changeRank* stream

Chapter 5

Experimental Evaluation

In this chapter we describe the experiments we conducted in order to evaluate the monitoring scheme we propose for the distributed environment. We compare our solution, with the centralized scheme, lets call it *Naive*, a notation also used in [5]. In *Naive* scheme, we suppose that all local statistics vector arrive at a central node who from the data it receives, computes the values for each $\bar{G}_l$ in order to find if the splitting condition at l is satisfied. This procedure takes place for every training example that arrives at l, so each site should communicate with the central node to inform it in regard with its local statistics vector.

The scope of our experiments is to examine the communication load of the scheme we propose in relation to $Naive$ scheme. For the experiments, we created a synthetic dataset following the idea developed in [3], [9]. In particular, we generate decision trees by randomly choosing the splitting attributes for each node and class labels for the leaves. Then we use those trees to the assign class label to the instances of the synthetic dataset. More specifically, we consider a set of 300 training examples that arrive at l, with $|X_l| = 4$ discrete binary attributes and binary class labels while we set δ parameter equal to 10^{-3}. For counting the number of exchanged messages in each of the above monitoring schemes for a leaf l, we consider the following formulas:

- $Naive$ scheme: $\#sites \cdot \#function\bar{G}_l \cdot \#instances$
- Our scheme: $(2 \cdot \#sites + 1) \cdot \#violations + 2 \cdot \#sites \cdot \#function\bar{G}_l \cdot \#calculateRank$

Regarding our scheme, as it can be examined from the formula provided, we care about calculating the messages needed for the clarification of a local violation as well as those needed for computing the new ranking of $\bar{G}_l$ in case one of the monitoring conditions is satisfied. For the

clarification process, the site which detects the violation informs the coordinator (1 message) which then requests the local statistics of the processing sites (thus n messages). Finally, the sites send back to the coordinator what it needs (thus n messages). Similar thinking applies for the other term of the formula provided. At the figures below, we show the communication load monitored for our scheme in relation to *Naive* scheme while the number of processsing sites increases and how the number of violations increase in relation to the number of sites:

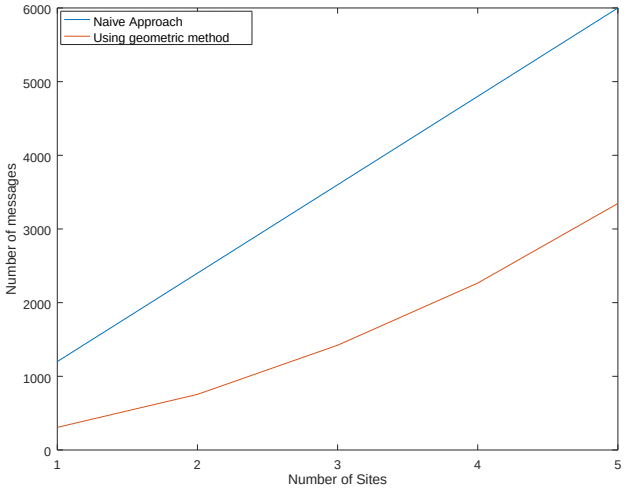


FIGURE 5.1: Communication Load for the Proposed Monitoring Scheme

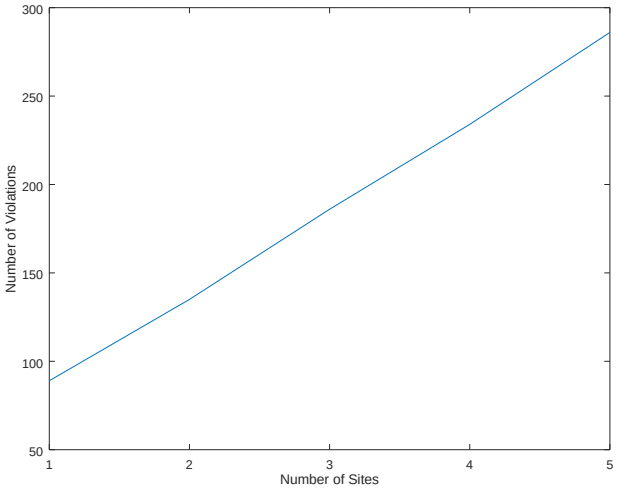


FIGURE 5.2: Increase in Violations

Note that the number of updates that arrive to the sites are equal to $\#sites \cdot |X_l|$ (linear to the number of sites) although each training instance affects only $|X_l|$ local statistics vector in

a certain site. Since threshold e_l for every instance that arrives in l is modified, we have to check whether the balls remain monochromatic for the new threshold even if the local statistics vector of a site remains the same. That in conjunction with the fact that a change in estimate vector of a function G_l might trigger local violations in sites whose local statistics vector is not modified because of the arrival of an instance, explains why the number of violations increases with the number of sites. Finally, the local statistics vector for a function G_l in general are similar across the sites, so if a local violations occurs in one site frequently occurs in others as well.

Furthermore, we wanted to see how violations spread while the stream evolves, in other words see when they occur. In parallel, we check the behaviour of the monitoring scheme in case that the processing sites check whether a local violation occurs after processing a batch of training examples in an effort to simulate the n_{min} parameter descibed in [3]. We set the number of nodes equal to 2 and $\delta = 10^{-3}$ and the results of the experiments conducted are:

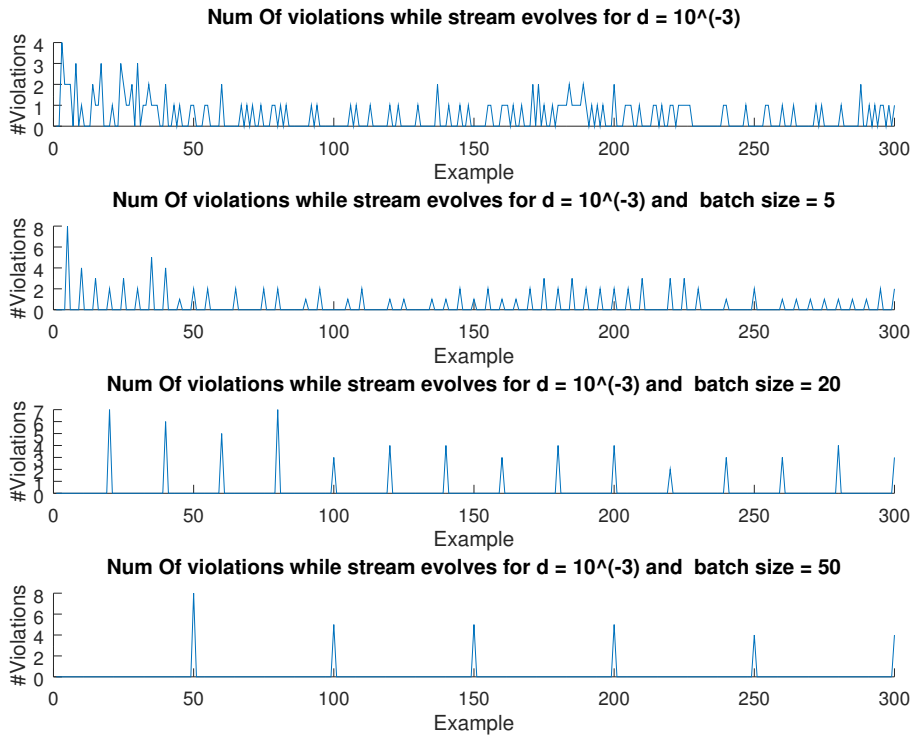


FIGURE 5.3: Number of violations in relation to the evolution of the stream

In the first subfigure, we see how the violations occur when the nodes check whether there is a local violation after processing each instance. Recall, that each instance produces $|X_l|$ changes in $\bar{G}_l$ functions and that each instance arrives to a certain site. Taking into account the above

observations, notice that while stream evolves and $\bar{G}_l$'s tend to stabilize around a certain range, the number of violations reduce. In particular, we observe that after the initial examples which tend to fire local violations for most of $\bar{G}_l$, then most of the time violations refer to only one function. That behaviour triggered our desire to understand which function(s), characterized by their rank, are the ones which cause the violations.

So, we provide a percental distribution of the violations with reference to the ranking of the function that produced them for diverse schemes which comprise of different number of sites:

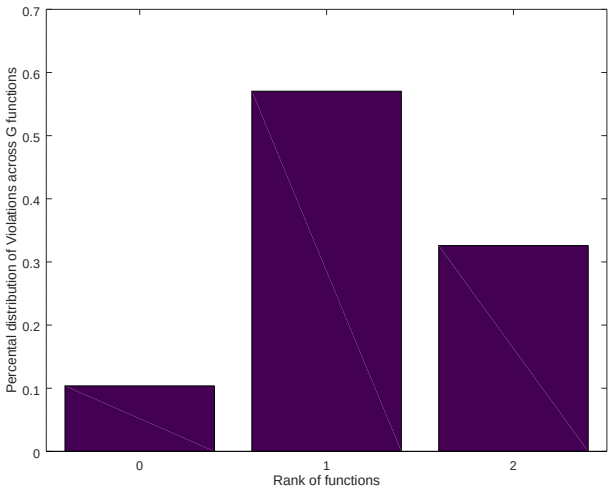


FIGURE 5.4: Distribution of violations in regard to function ranking and number of processing sites

From the results, we notice that the functions that caused the majority of violations are produced from the top-2 functions. Thus, the interesting part is that most of the time non-top objects do not cause violations. So we rarely need to actually compute their values in contrary to the naive approach where for each example we need to compute all $\bar{G}_l$ functions. In fact, in the monitoring scheme we propose, the occasions for which we need to compute the value of a $\bar{G}_l$ is either when it causes a local violation, or when the coordinator decides to find the new ranking between the functions. The latter occasion is represented in our subfigure by the red line, which declares at which example a new ranking was computed by the Coordinator. Notice, that in the beginning of the stream where the value of $\bar{G}_l$'s are computed on a small subset examples and are not stabilized around a certain range, such occasions are more frequent. Finally, for the batch approach, at first note that the maximum number of violations now are $\#sites \cdot |X_l|$ since after a set of n_l examples, we check whether a local violation occurs in any of the nodes for any of the monitored functions. We observe, that while the length of the batch increases, the rhythm with which the violations decrease deteriorates since sites do not frequently receive

information for the global estimate vector and the changes in $\bar{G}_l$ values are rapid. Note for example, that when batch size is equal to '5' there are batches that cause no violations which is not the case for batch size equal to '20' or '50', where most of the time at least '2' functions trrigger a synchronization phase. Nonetheless, since the number of updates are less we manage to reduce the communication load missing however any example inside the batch that might have caused a violation which could lead to a synchronization phase where a split decision could be made.

Furthermore, we conducted a set of experiments with 4 discrete attributes, 2 of which were ternary and the other two binary, while class label was binary. We consider that leaf l processes a set of 400 training examples, and we evaluate how violations behave for different values of δ parameter:

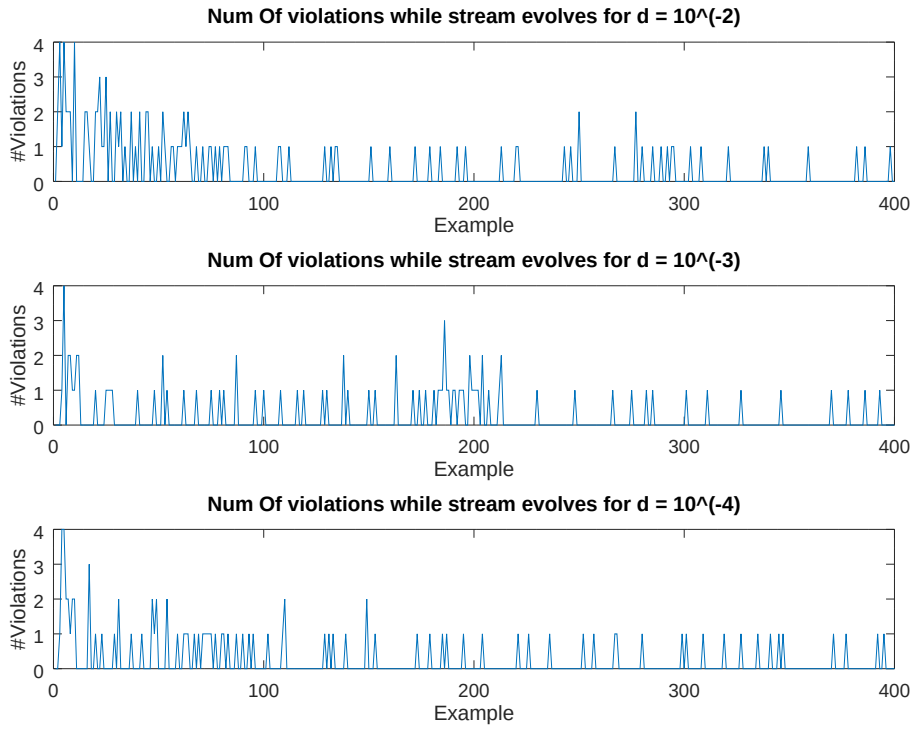


FIGURE 5.5: Number of Violations in regard to δ parameter

Again, for all of the monitoring schemes, while stream evolves the number as well as the frequency of local violation decreases. Nonetheless, while δ decreases, thus ϵ decreases and takes value in $\bar{G}_l$ function range more quickly, the number of violations in the beginning of the stream are more.

To sum up, from the experiments conducted we observe that the monitoring scheme presented in 3.1.2 reduces the communication load in relation to the *Naive* approach. At the same time, we observed that the number of needless computation of G function at leaf l is significanlty reduced since most of the time non-top functions trigger no violations and computing the new ranking between G functions occurs only when there is a chance that the splitting condition is satisfied.

Chapter 6

Conclusions and Future Work

6.1 Conclusions

In traditional Hoeffding trees every time a training instance arrives at a leaf l, the system should compute for every candidate splitting attribute its G value, considering G is the information gain. For the distributed setting, by using horizontal partinioning across processing sites we consider that for calculating these G values the sites send their data to a central node who then executes the computations needed. In this thesis, we designed a novel distributed scheme for monitoring the splitting condition of a leaf in streaming decision trees using Geometric Monitoring in an effort to reduce communication load. From the experiments conducted, we observe that in comparison to the *Naive* approach, the communication load is reduced especially for datasets where some attributes become clearly better than others. At the same time, when the information gain of the attributes in the dataset is significantly different, we rarely need to compute the information gain for less important attributes, knowledge which cannot be acquired while using traditional methods.

6.2 Future Work

The current work can be extended in various ways. First of all, note that testing for monochromicity was implemented by computing a grid in $\mathbb{R}^d$ inside the ball constructed by each site. Thus, the computational cost is significant high and as a result it is important to find an optimized way to find the maximal and minimum value of the monitored function so as to be able to further evaluate the monitoring scheme proposed based on large data. In addition, another idea

is to find how to incorporate the method proposed in the work of Keren et al [14] so as to declare safe zones for information gain method, which is not an immediate procedure and compare how the behavior and the communication load of the monitoring system changes. Another extension of this work would be to study how to incorporate continuous attributes in our scheme and how to find an optimized way so as to find the best splitting point for them. Finally, we could examine if the proposed solution can be extended for time-changing streams.

References

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

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

[3] P. Domingos and G. Hulten, "Mining High-Speed Data Streams", in Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data mining, KDD 2000.

[4] G. Hulten, L.Spencer and P.Domingos, "Mining Time-Changing Data Streams", in Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data mining, KDD 2001.

[5] I. Sharfman, A.Schuster and D. Keren, "A Geometric Approach to Monitoring Threshold Functions Over Distributed Data Streams", in Proceedings of ACM SIGMOD International Conference on Management of Data, SIGMOD 2006.

[6] O. Papapetrou and M.Garofalakis, "Continuous fragmented skylines over distributed streams", in IEEE 30th International Conference on Data Engineering, 2014.

[7] S. Burdakis, A. Deligiannakis, "Detecting Outliers in Sensor Networks using the Geometric Approach", in 2012 IEEE 28th International Conference on Data Engineering

[8] Manikaki Vasiliki, "Distributed Event Detection using the STORM System", Diploma Thesis, Technical University Of Crete, 2014, http://purl.tuc.gr/dl/dias/C88659C3-1CF1-4F4E-992E-4B55658B011C

[9] G. Hulten, P. Domingos and L. Spencer, "Mining Massive Data Streams", Journal of Machine Learning Research

[10] Y. Ben-Haim and E. Tom-Tov, "A Streaming Parallel Decision Tree Algorithm", Journal of Machine Learning Research, pp. 849-872, 2010

[11] A. Murdopo, "Distributed Decision Tree Learning for Mining Big Data Streams", Master Thesis, UPC Universitat Politècnica de Catalunya, 2013, http://people.ac.upc.edu/leandro/emdc/arinto-emdc-thesis.pdf

[12] N. Kourtellis, G. De Francisci Morales, A. Bifet and A. Murdopo, "VHT: Vertical Hoeffding Tree", IEEE International Conference on Big Data, 2016

[13] J. Gama, R. Rocha and P.Medas, "Accurate Decision Trees for Mining High-speed Data Stream", in Proceedings of ACM SIGKDD International Conference on Knowledge Discovery and Data mining, KDD 2003.

[14] D. Keren, I. Sharfman, A. Schuster and A.Livne, "Shape Sensitive Geometric Monitoring", IEEE Transactions On Knowledge and Engineering, Vol. 24, No.8, 2012

[15] J. Leibiusky, G. Eisbruch and D. Simonasi, "Getting Started with Storm", O'Reilly, 2012

[16] Apache Storm, http://storm.apache.org/