TECHNICAL UNIVERSITY OF CRETE

DIPLOMA THESIS

Implementation of decision trees for data streams in the Spark Streaming platform

---

Author:
Christos Ziakas

Supervisor:
Prof. Minos Garofalakis

A thesis submitted in fulfilment of the requirements
for the degree of Diploma in Electrical and Computer Engineering

in the

Software Technology And Network Applications Laboratory
School of Electrical & Computer Engineering

September 17, 2018

Abstract

In the era of big data, enormous amounts of data are created, replicated and transferred every day. The current technology for handling and analyzing vast amounts of data allows us to develop applications for various problems (e.g., DNA sequence analysis, medical imaging, traffic control) that could not previously be solved efficiently. More precisely, the time required to process large volumes of data can be minimized by using distributed computing platforms such as Apache Spark. The Apache Spark framework includes various implementations for large-scale machine learning, distributed data streaming processing and parallel graph analytics. The Spark Streaming platform provides scalable and fault-tolerant data streaming processing. However, there is only a limited number of implemented distributed incremental machine learning algorithms available in the Spark Streaming platform.

In this thesis, we propose a parallel implementation of an incremental and scalable tree learning method for classification in Spark Streaming, the Hoeffding decision tree. Our proposed implementation performs horizontal data parallelism in the shared-nothing architecture of Spark. The Hoeffding bound guarantees with high confidence that the Hoeffding decision tree is asymptotically identical to a batch-learning one. The high dimensional statistics, required for evaluating splits, are stored as sparse matrices in main memory across the Spark cluster. These statistics are instantly updated, when new training instances are available. Furthermore, distributed computations are performed in order to identify the optimal split and assess whether the splitting criterion is satisfied. The generated model is used in order to make color classification based on the spectral signature of each color. Each color has a different chemical composition, and as a consequence a different spectral signature.

Acknowledgements

Firstly, I would like to express my sincere gratitude to my advisor Prof. Minos Garofalakis for introducing me to research and for his guidance throughout this thesis. I would also like to thank the members of committee, Prof. Antonios Deligiannakis and Prof. Vasilis Samoladas, for evaluating this thesis. Furthermore, I would like to thank Prof. Costas Balas and all the members of Electronics Laboratory for providing me with the data set used in the experimental study of this thesis. Especially, Mr. Athanasios Papathanasiou who helped me understand the theoretical background of these experiments and create the data set. I would also like to thank Prof. Michail Lagoudakis and Prof. Vasilis Samoladas for their kindness and support. Last but not least, I am deeply thankful to my family and my friends for always supporting me throughout my studies.

Contents

Acknowledgements **iii**

1 Introduction **2**

2 Decision Tree Classifiers **4**
2.1 Overview of Decision Tree Classifiers 4
2.2 Scalable Decision Trees 8
2.3 Incremental Decision Trees 9
2.4 Hoeffding Decision Trees 9

3 Machine Learning in Apache Spark **12**
3.1 Apache Spark Core 13
3.2 Apache Spark Streaming 15
3.3 Decision trees in Apache Spark 17

4 Hoeffding Decision Trees in the Spark Streaming Platform **19**
4.1 Our Contribution 19
4.2 Updating the Global Statistics 22
4.3 Updating the Hoeffding Decision Tree 24
4.4 The proposed implementation in Spark Streaming 27

5 Experimental Study **31**
5.1 Hyper-spectral imaging 31
5.2 Data sets 32
5.3 Performance Evaluation 35
5.3.1 Accuracy 35
5.3.2 Scalability 37

6 Conclusion **40**

Bibliography **42**

List of Figures

2.1 An example of decision tree classifiers 5
2.2 Algorithm for constructing basic decision tree classifiers 5
2.3 An example of computing information gain metric 7
2.4 Algorithm for constructing Hoeffding tree classifiers 10

3.1 The architecture of Apache Spark 14
3.2 The architecture of the Spark Streaming platform 15
3.3 An overview of the Spark Streaming platform 16

4.1 The data flow diagram for updating the global statistics 25
4.2 The data flow diagram for updating the Hoeffding decision tree 28
4.3 An execution plan of a Spark job in the proposed implementation 29

5.1 The electromagnetic spectrum 31
5.2 Hyper-spectral data cubes 32
5.3 The X-Rite ColorChecker SG 33
5.4 Comparison between the batch and the Hoeffding decision tree for different number of attributes in terms of accuracy 36
5.5 Comparison between the batch and the Hoeffding decision tree for different number of training instances in terms of accuracy . 37
5.6 The throughput of the proposed implementation for training instances increased proportionally to the number of executors . . 38
5.7 The processing time of each Spark batch for different number of executors 39

Chapter 1

Introduction

In the era of big data, enormous amounts of data are created, replicated and transferred every day. In 2010 Eric Schmidt perfectly illustrated this new era by stating: *"From the dawn of civilization to 2003, five exabytes of data were created. The same amount was created in the last two days"*. The current technology for handling and analyzing vast amounts of data allows us to develop applications for various problems (e.g., DNA sequence analysis, medical imaging, traffic control) that could not previously be solved efficiently. However, despite the remarkable progress that has been achieved over the years, computer scientists are still facing new challenges arising from the analysis of extremely large data sets. Storing and processing massive data sets as well as applying machine learning methods and private data analytics to them are the most significant open research areas.

Machine learning is a scientific area combining the fields of computer science, optimization and statistics in order to build effective prediction models from data sets or discover patterns in them. The ultimate goal of machine learning algorithms is to approximate a function $y = f(x)$, where y is the output and x the input, in order to accurately predict the output of future inputs. Various problems can effectively be solved by machine learning algorithms (e.g., classification, regression, clustering) with supervised or unsupervised learning methods. In this thesis, we concentrate on the classification problem addressed by supervised learning algorithms. Numerous models have been proposed over the years in order to address that problem such as linear classifiers, decision trees and neural networks.

Large-scale machine learning approaches re-consider machine learning algorithms and data mining techniques in order to apply them to very large or even unbounded data sets. Distributed machine learning, online machine learning and deep reinforcement learning are just a few of the areas covered by large-scale machine learning. Moreover, additional opportunities exist in applying large-scale machine learning to various problems in healthcare, finance and cyber security.

Online machine learning is more and more popular because of the constant increase in the volumes of data. Various applications are needed to handle, process and analyze in-demand unbounded data sets known as data streams. In this regard, building synopses of data streams and effective prediction models from them remain open research areas. Furthermore, the time needed to achieve that can be minimized by using fault-tolerant and scalable cluster-computing frameworks such as Apache Hadoop, Apache Spark and Apache Flink. Plenty of batch-learning machine learning methods have been implemented in these platforms. However, despite the fact that existing distributed data streaming platforms can process quickly the unbounded data

sets, there is only a limited number of implemented distributed incremental machine learning algorithms.

In this thesis, we propose a scalable implementation of Hoeffding decision trees in the Spark Streaming platform. Storing and updating the statistics needed to evaluate the splits as well as computing the best split for each node are performed in the memory of the worker nodes. The classification model is used in order to predict the color of an object based on its spectral signature.

The remainder of this thesis is organized as follows. In chapter 2, we provide an overview of decision tree classifiers as well as we describe the Hoeffding decision tree classifier and its extensions. Next, chapter 3 presents the Spark Streaming platform and the available batch-learning decision tree classifiers in Spark. In chapter 4, we present the proposed implementation and compare it with other approaches in the literature. In chapter 5, we evaluate the proposed implementation in terms of accuracy and scalability. Finally, chapter 6 concludes the thesis.

Chapter 2

Decision Tree Classifiers

2.1 Overview of Decision Tree Classifiers

Decision tree learning methods are supervised learning methods and can effectively address classification and regression problems. Decision trees are divided into two broad categories based on their desired output. When it is a discrete variable, decision trees are called classification trees. In case of continuous output, decision trees are called regression trees. In this thesis, we concentrate on classification trees. The input of these models, known as training data set, consists of multiple training instances (records), each one containing a fixed number of variables. These variables, called attributes (features), are characterized as either categorical or numerical. Categorical attributes take values from a finite set of discrete values. When there is an ordering between these discrete values such as $\{high, low\}$, categorical attributes are called ordinal. Otherwise, they called nominal such as color and gender. Numerical or continuous attributes take values from a subset of real numbers such as integers. Each training instance contains a categorical attribute, known as dependent attribute or label, corresponding to its class. The other attributes, called independent attributes, can be either categorical or numerical.

Constructing an optimal decision tree is an NP-complete problem [34]. Thus a recursive greedy algorithm, expanded in depth-first manner, has been proposed in order to efficiently build decision trees. The basic approach for building decision tree classifiers is to expand a tree until a stopping criterion is satisfied. Decision trees are expanded by splitting each leaf node based on a splitting criterion. So, starting with only one root node, the model is built by recursively selecting the best split for each leaf node. The algorithm is halted when a stopping criterion is fulfilled for each leaf node. Each leaf node is characterized by the class containing the most instances in the node. For instance, in figure 2.1, a decision tree classifier is built in order to predict the admission decision for a PhD applicant based on his GPA and GRE performance. An algorithm for constructing basic decision tree classifiers is described in figure 2.2. Decision trees predict the output of an unseen instance, without a dependent attribute, by asking a sequence of question on independent attributes. Each internal node represents a test on an independent attribute and each leaf node represents a class. More precisely, a classification tree is capable of predicting the class of a new instance by traversing its internal nodes until a leaf node is reached. Then, the new instance is classified to the class corresponding to the leaf node class.

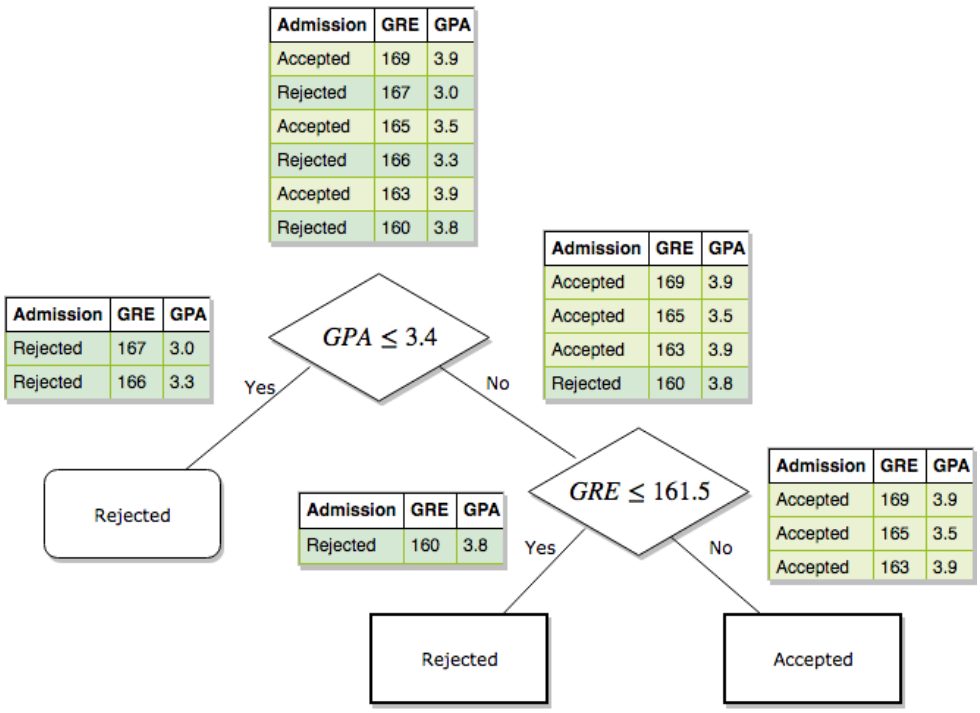


FIGURE 2.1: An example of decision tree classifiers

Data: Labeled Training Dataset D, root node N, Decision tree containing only N
Result: Decision tree classifier T
Algorithm `BuildTree(`D,N,T`)`
  if *stopping criterion is satisfied* **then**
    Assign the most frequent class to N
    Return T
  Evaluate splits of N for each attribute based on a metric
  Find best attribute split of N
  Create child nodes N_L and N_R
  Update T
  Partition D to D_L and D_R based on best split
  $BuildTree(D_L, N_L, T)$
  $BuildTree(D_R, N_R, T)$
  Return T

FIGURE 2.2: Algorithm for constructing basic decision tree classifiers

A signifianct amount of research has been conducted on decision trees classifiers over the years. THAID [52] was the first decision tree classifier extending AID [53] regression decision tree in order to address classification problems. However, decision trees were prone to adapt the patterns of their training data sets. This problem, known as overfitting, does not allow decison trees to make accurate prediction on unseen data as it is too fitted to its training data set. CART [10] proposed a novel approach known as pruning phase in order to mitigate the influence of training data sets to decision trees. The main idea of this technique is to let decision trees expand, most often with a loose stopping criterion, and prune them during the expansion phase (pre-pruning) or after that (post-pruning) by applying a pruning method to them. Numerous post-pruning strategies [16] have been introduced over the years such as cost-complexity pruning, pessimistic pruning [59], MDL pruning [60] and optimal pruning based on dynamic programming [8, 1]

Evaluating splits and finding the best one is a hard problem in terms of computational cost and in this regard many heuristics exist in the literature [63]. The most notable metrics used for that purpose are information gain, gain ratio and gini index. Further, there are many scientific works in literature for implementing multivariate splits and handling missing values [41]. Several decision tree classifiers with various pruning, splitting and stopping criteria have been implemented (e.g, ID3 [58], C4.5 [57], FACT [47], QUEST [46], CRUISE [38]) and compared [45] over the years.

In our implementation, the information gain metric is used. The information gain measures the difference in terms of entropy between a parent node and child nodes after splitting based on an attribute. The entropy [67] of a node can conceptualize the uncertainty of a node. If there are many training instances, classified in different classes, the entropy is high. On the other hand, if the most of training instances in a node belong to the same class, the entropy metric is low. Let X be a random variable corresponding to the class of an instance in a node with values in the set $E = \{0, 1, .., C\}$, where C is the number of classes. Further, let ω_i be the number of instances classified in class i in that node, the entropy $H(X)$ is

$$\begin{aligned} H(X) &= -\sum_{i=0}^{m-1} P\{X = i\} \log_2 P\{X = i\} \\ &= -\sum_{i=0}^{m-1} \frac{\omega_i}{\sum_{j=0}^{m-1} \omega_j} \log_2 \frac{\omega_i}{\sum_{j=0}^{m-1} \omega_j} \end{aligned}$$

A multi-way split S in a decision tree with degree d divides the instances of parent node N into $n_1, ..., n_d$ disjoint sets forming d child nodes. Let $H(X_i)$ be the entropy of each child node $i \in \{1, ..., d\}$. The information gain $IG(N, S)$ of split S in parent node N is

$$\begin{aligned} IG(N, S) &= H(N) - H(N|S) \\ &= H(N) - \sum_{i=1}^{d} \frac{n_i}{\sum_{j=0}^{m-1} \omega_j} H(X_i) \end{aligned}$$

The split with the highest information gain is the best one. Therefore, the decision tree is split based on that split. In figure 2.3, the information gain values of two splits are computed for the problem illustrated in 2.1. It is obvious

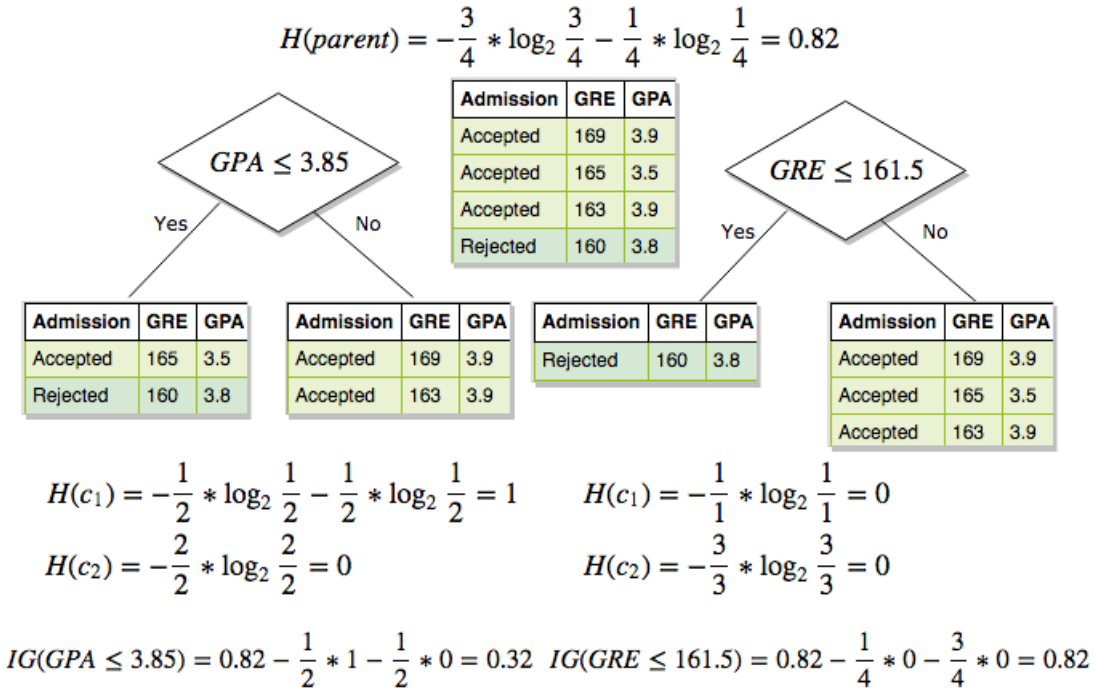


FIGURE 2.3: An example of computing information gain metric

that the second one is the best split.

In an effort to address the "overfitting" problem and as a consequence achieving more accurate predictions, several ensemble methods [14, 80] have been applied to decision trees over the years. A prominent among them is bootstrap aggregation method, also known as "bagging" [9]. The main idea of this method is to build a large number of decision trees, each one trained by an equal part of the training data set choosing with sampling with replacement. The set of decision trees predicts the class of a test instance by aggregating the outputs of each classifier and choosing the most dominant class. Moreover, several random subspace methods have been proposed over the years [30]. Random forests[11] and their variations [6] (e.g., Extra-trees [27], Rotation forest [62], Reinforcement Learning trees [81]) successfully combine the "bagging" method with a random subspace method (f.e random selection of features) in order to address the overfitting problem.

Another influential ensemble method used for improving the performance of decision trees classifiers is boosting. The objective of boosting algorithms [64, 18] is to convert a weak classifier to a strong one. This approach is conceptualized in decision trees classifiers by training multiple versions of them. Each one is constructed from a new version of the training data set which is properly weighted in order to reduce the number of its misclassified instances. Further, each version of decision trees contributes to the classification of a new instance by voting based on its performance (prediction accuracy). There are many effective implementation of boosted classification decision trees in literature such as Adaboost [19], LogitBoost [20] and BrownBoost [17].

2.2 Scalable Decision Trees

Building scalable decision trees from massive data sets is a well-studied problem in the research area of decision trees. In many cases, training data sets cannot fit in main-memory. So, several disk-based methods have been proposed over the years in order to efficiently build decision trees by limiting or removing the memory restrictions between main memory and the size of data sets. Disk-based classifcation trees, grown in breadth-first manner (level by level), require a scan over the training set per level of the tree. SLIQ [49] requires only an amount of main memory increased proportionally to the size of the training data set. In order to achieve that, a pre-sorting technique and a state-of-the-art data structure, called class list, are used. The class list is stored in main memory throughout the building phase, but the training data set is stored in secondary storage. Further, a pass over the training data set is required at each level-wise step of the building phase. However, SLIQ cannot entirely remove memory restrictions as the class list is required to be stored in main memory. SPRINT [66] removes these memory restrictions between the size of data set and main memory by proposing a different data structure, not required to be stored in main memory. Both tree learning methods use an identical MDL-based pruning method. RAINFOREST [26] framework requires a reasonable amount of main memory for evaluating a split based on attribute-value-class (AVC-group) statistics. Several well-known decision tree classifiers such as C4.5, CART, SLIQ and SPRINT can be implemented in this framework. Moreover, various statistical techniques have been applied to decision tree learning methods in order to deliver more efficient implementations (e.g., BOAT [25] , CLOUDS [61]).

Parallel implementations of decision tree learning methods can offer scalability to decision tree classifiers as well. Parallel systems consist of multiple computational nodes that can be cluster nodes, processors or threads. In shared-memory architecture, computational nodes share a main memory. On the other hand, in shared-nothing architecture, nodes have their own main and secondary memory and they communicate via a message parsing system. The most significant approaches for constructing decision trees in parallel are task parallelism, data parallelism and hybrid parallelism. In case of task parallelism, the nodes of a decision tree are distributed to computational nodes. Data parallelism can be achieved by distributing the data set to the computational nodes horizontally (instances) or vertically (attributes). These methods are known as horizontal and vertical parallelism, respectively. Hybrid parallelism approaches combine task and data parallelism in order to build decision tree classifiers.

There are several implementations of parallel decision tree classifiers in shared memory and shared nothing environments in the literature [56, 2]. Mostly, data parallelism and hybrid parallelism are preferred from task parallelism due to the latter's lack of load balance. SPRINT decision tree classifier uses vertical data parallelism in a shared-nothing environment and is one of the most influential approaches. In the same work, two parallel implementations of SLIQ decision tree classifier (SLIQ/R, SLIQ/D) are proposed. ScalPar [37] suggests an optimization of SLIQ/D by using parallel hash tables. Furhermore, an implementation of SPRINT in shared-memory systems exists in literature [79]. SPIES [35] successfully combines RAINFOREST's AVC-statistics with a sampling method in a shared-memory environment. Finally, various

implementations of decision tree classifiers (e.g., SUBTREE [79] , Pclouds [69], PLANET [54]) using hybrid parallelism in shared-memory and shared-nothing have been proposed over the years.

2.3 Incremental Decision Trees

Extensive research has been conducted in regard to develop techniques for incrementally updating decision trees when new information is available. Batch (non-incremental) learning methods require to re-construct the decision tree in order to adapt to new knowledge. Numerous incremental decision tree learning methods for classification exist in literature. A novel approach to that problem is to train a decision tree as a non-incremental model and then adjust it to the new instances. Whenever internal nodes are obsolete, they are removed (ID.4 [65]) or the decision tree is reconstructed (ID5R [70], ITI [71]) in order to produce an up-to-date model. These models assume that the training dataset fits in memory. BOAT [25] addresses that problem by building a coarse-grained decision tree from a sample of the dataset, fitting in memory. and then by refining it with the remaining dataset including new arrivals.

The most influential approach for constructing incremental decision tree classifiers is the Hoeffding tree learning method proposed by Domingos and Hulten [15]. Hoeffding decision trees and their variations are based on Hoeffding bound [31] in order to decide whether there is enough evidence for splitting a node. It can be proven that by expanding a tree model with that approach, it is likely to be identical to a batch-learning one. Moreover, only the decision tree and the statistics for evaluating the splits need to be stored in main memory. Various extensions of Hoeffding decision trees exist in the literature [21] in order to handle concept-drift [22] and numerical attributes [55] as well as minimize the time and space complexity. Furhermore, plenty ensemble methods for data streams have been applied to Hoeffding trees over the years [44].

2.4 Hoeffding Decision Trees

Hoeffding decision tree learning methods use the Hoeffding Bound [31] in order to decide whether there is enough evidence for splitting a node. The Hoeffding bound approximates the expected value of a random variable Y with range R based on the mean value $\overline{Y}$ of a sequence of n independent observations of Y. More precisely, the expected value $E[Y]$ is at least $\overline{Y} - \epsilon$, with probability $1 - \delta$, where

$$\epsilon = \sqrt{\frac{R^2 \log \frac{1}{\delta}}{2n}}$$

Intuitively, as the sample size n increases, the expected value of Y converges to the sample mean, regardless of the population distribution. In case of Hoeffding trees, the random variable is $Y = G(X_a) - G(X_b)$, where G, X_a and X_b are the evaluation metric, the attributes with the best evaluation metric and the attribute with the second best evaluation metric, respectively. Then, the mean value of Y after observing n training instances is $\overline{Y} = \overline{G}(X_a) - \overline{G}(X_b)$.

$\overline{G}(X_i)$ and $\overline{G}(X_b)$ are the attributes with the best and second best evaluation metric after observing n training instances in a leaf node. As described previously, $E[Y] \geq \overline{Y} - \epsilon$ with probability $1 - \delta$. When $\overline{Y} > \epsilon$, then $E[Y] > 0$ with probability $1 - \delta$. Sequentially, $E[Y] > 0 \Leftrightarrow G(X_a) - G(X_b) > 0 \Leftrightarrow G(X_a) > G(X_b)$, which means that the split on X_a is indeed the best one with $1 - \delta$ probability. In our implementation, the information gain metric, described previously, is used for evaluating splits. X_a and X_b are the attribute with the highest and second highest information gain. The range of the information gain metric is $R = \log_2 c$, where c is the number of classes. The Hoeffding decision tree classifier is a binary decision tree. Thus only binary splits are evaluated. Further, a null split with $IG = 0$ is considered in order to ensure that the split with the highest information gain is better than not splitting.

So, the Hoeffding tree learning method starts with a root node and expands based on the Hoeffding Bound. If $\overline{Y} = \overline{G}(X_a) - \overline{G}(X_b) > \epsilon$, the node is split into two new nodes. For each leaf node is essential to hold some statistics in order to compute the information gain of each split. The required statistics are the frequency of each class-value of each attribute in each node of a tree. Therefore, the space complexity of Hoeffding tree algorithm is $\mathcal{O}(n * a * v * c)$, where n, a, v and c are the leaf nodes, the attributes of each leaf node, the values of each attribute and the possible classes, respectively. In addition, the tree model needs to be stored in main memory as well. Each leaf node is assigned with its majority class. An algorithm for constructing Hoeffding decision tree classifiers is described in figure 2.4.

Data: Data stream of instances.
Result: Hoeffding tree classifier H

```
1  Algorithm BuildTree(Training Dataset, δ)
2      Create H containing only the root node
3      while true do
4          for each new instance do
5              Traverse H until a leaf node l
6              for each attribute a ∈ l do
7                  Update the counts of attribute a
8          for each leaf node l ∈ H do
9              Evaluate the splits based on Information Gain ratio
10             Find the best split G̅(X_a)
11             Find the second best split G̅(X_b)
12             Compute ε = sqrt(R² log(1/δ) / 2n)
13             if (G̅(X_a) − G̅(X_b) > ε) and (G̅(X_a) > 0) then
14                 Split H based on the best split
15         for each leaf node l ∈ H do
16             Assign the most frequent class to the leaf node l
17 return H
```

FIGURE 2.4: Algorithm for constructing Hoeffding tree classifiers

A system based on the Hoeffding tree called VFDT [15] includes some

mechanisms in order to improve its performance. Methods for handling attributes with similar information gain values (ties) and with poor contribution to prediction are presented. Moreover, the evaluation of splits as well as whether the Hoeffding bound is satisfied is executed when a great number of instances has arrived. However, this implementation assumes that the data generated by a stationary distribution. In this regard, CVFDT [33] extends VFDT system in order to handle concept-drift. CVFDT achieves that by holding statistics for both interior and leaf nodes, considering only training instances arrived during a sliding window. When a split is no longer the most informative, an alternative sub-tree is grown based on the new best split. The outdated sub-tree is replaced by the new one when the latter become more accurate.

Furthermore, significant amount of research has been conducted in order to handle numerical attributes. In batch tree learning methods, when a numerical attribute takes n different values, $n-1$ splits are examined in order to find the best one. The split is formed as $attr \leq mid$, where mid is the midpoint value of two successive values of the numerical attribute $attr$. However, in streaming algorithms the values of numerical attributes are not known a priori. The implementation of VFDT in VFML package [32] addresses that problem by constructing a fixed-size histogram with N bins. The boundaries of each bin are determined by the first N values arriving at the system.

VFDTc system [23] stores all values that have not been seen previously in a B-tree data structure. However, this approach is inefficient because B-tree is constantly increasing in size when numerical attributes take many values. Furhermore, it proposes to classify test instances by using a Naive bayes classifier in each leaf node instead of assigning them with the majority class. Another apporach is Numerical Interval Pruning (NIP)[36] which builds a number of intervals and then it prunes them by applying statistical tests. Furthermore, a method for solving ties by combining Hoeffding trees with Option trees [12] exists in the literature [39]. Whenever there are more than one good choices for splitting a node, a set of binary trees is built for each good choice. The training and test instances traverse more than one decision trees. A test instance is classified by aggregating the votes of each traversed tree. So, Hoeffding trees can handle a tie by using the strategy of Option trees.

UFFT system [24] follows a more sophisticated approach in an effort to improve the time and space complexity of the tree learning algorithm. The main idea is to hold only the incremental mean and variance of the values of numerical attributes for each class in each leaf node. Then, the values per class in each leaf node are modelled as a normal distribution and the best split is found based on a discriminant analysis. This approach is only applied to binary classification problems. Thus, in case of a multi-class problem with n classes, UTTF divides the n-class problem into k binary classification problems, where $k = \binom{n}{2}$ is the number of pairs of classes. For each binary classification problem, a Hoeffding binary tree is constructed. When a new training instance arrives, it traverses only the binary trees containing its class label. The final model is a forest of decision trees and makes predictions by traversing all trees. Each tree classifies the test instance in a class and assigns a probability to its decision. The test instance is classified in the class taking the highest aggregated probability based on the sum rule [40]. Furtheromore, an extension of UTTF system [55] proposes to find the best split in a multi-class problem by considering many splitting points and choosing the best one.

Chapter 3

Machine Learning in Apache Spark

In the era of big data, the time needed to analyze vast amounts of data is an important factor for many applications. This time can be reduced by using distributed processing of data. Numerous cluster-computing frameworks, achieving fault-tolerance and scalability, have been proposed over the past few years. One of the most well-known cluster-computing framework is Apache Hadoop [73] drawing high attention from both industry and academia. Hadoop ecosystem consists of an execution engine, a distributed file storage system (HDFS [68]) and a cluster manager. Hadoop is based on a master/slave shared-nothing architecture. The execution engine of Apache Hadoop is an implementation of MapReduce programming model [13] providing parallelism in distributed environments.

A MapReduce program is a sequence of MapReduce jobs. Each MapReduce job is executed in two phases, known as map and reduce phase, respectively. In the map phase, a mapper is created in each worker node of the cluster. Each mapper takes as an input a partition of data. To this end, data are partitioned across worker nodes. Then, all mappers apply a map function to their corresponding partitions in parallel. A map function outputs information in key-value pair format. All key-values pairs are sorted based on key and sent across worker nodes based on a partition function. In order to achieve that, shuffling of data is required in most cases. All information, regarding a key, is accumulated in the same worker node. In reduce phase, a reducer is created for each worker node containing accumulated information of a key. All values of a key are inserted in a list. Therefore, each reducer takes as an input at least one key-value pair, where the value is aggregated information of each key. Then, a reduce function is applied to each pair in parallel. The results are stored in HDFS and they can be either consumed or used as an input to another MapReduce job.

Despite the wide range of applications that Apache Hadoop can efficiently be used, the necessity for implementing a map and a reduce phase is a bit restrictive. Moreover, storing the output of each MapReduce job in secondary storage is an expensive operation in terms of time and space complexity. In this regard, Apache Spark [78] developed at Berkeley's AMPLab can achieve distributed processing of data more efficiently than Apache Hadoop, especially for iterative algorithms such as machine learning algorithms. Spark can*"run workloads 100X faster"* than Hadoop. Spark extends the MapReduce programming model by allowing multiple passes of data in main memory based on the Resilient Distributed Dataset (RDD) abstraction [77]. An RDD is a distributed, read-only and immutable collection of serializible objects. Further,

Spark achieves high performance for both batch and streaming data, using a state-of-the-art DAG scheduler, a query optimizer, and a physical execution engine.

3.1 Apache Spark Core

Apache Spark is a cluster-computing framework providing APIs in Scala, Java, Python and R programming languages. Apache Core, implemented in Scala, is the basic component of Apache Spark. Spark Core is responsible for task scheduling, providing fault-tolerance and memory management. Various high-level libraries have been developed on top of Spark Core over the past few years. Spark SQL [3] library provides execution of SQL queries as well as Dataset and Data-frame structured data structures. Machine learning (MLlib) library [50] includes machine learning algorithms for RDDs and for structured inputs. Further, there are two libraries for distributed streaming processing. Spark Streaming library [76] is based on RDDs and Structured Streaming library built on Spark SQL library. Finally, Apache Spark contains GraphX [75] library for parallel graph analytics and SparkR [72] for using Spark from R.

Each Spark application consists of a driver program, a cluster manager and many worker nodes. The driver program contains the main program and a SparkContext object. Spark applications acquire computational resources, defined by SparkContext, through a cluster manager (e.g., standalone, Mesos, YARN). A worker node is a cluster node that executes application programs on it. Each worker node contains a number of executors. An executor is responsible for running tasks and storing data. A task is a unit of work, sent to one executor by SparkContext. In this regard, the driver program schedules the executors in order to execute their tasks. The figure 3.1 (*source: www.spark.apache.org*) illustrates Spark's architecture described above.

A Resilient Distributed Dataset (RDD) is an immutable distributed collection of objects. An RDD is always re-computable (resilient) and partitioned across worker nodes (distributed). It also contains any serializable data type (dataset). An RDD can be generated by four different ways. An RDD can be created by external data (e.g., HDFS), by transforming other existing RDDs, by parallelizing a collection in the driver program or by changing the persistence of an existing RDD. Further, each RDD holds information regarding its parent RDDs or the external data from which it was created, known as dependencies. However, all RDDs should be stored or derived from an RDD in stable storage. In case of a node failure, an operation on an RDD can be re-executed by recovering an ancestor RDD. An ancestor RDD can be found by traversing a lineage graph of an RDD. A lineage graph of each RDD can be created by its set of parent dependencies. Therefore, this mechanism achieves fault-tolerant implementations. Apart from a set of dependencies on parents RDDs, each RDD consists of a set of partitions and a function for computing data partitions given dependencies. Further, each RDD holds meta-data in regard to preferred data placement (data locality) and partition function (e.g., hash-partitioning, range-partitioning).

There are various operations in Spark and are divided into two broad categories, transformations and actions. Generally, transformations construct a new RDD form previous ones by applying a function to them. On the other

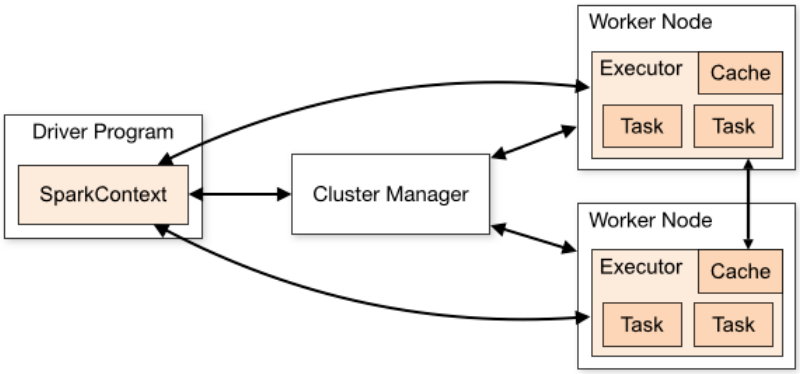


FIGURE 3.1: The architecture of Apache Spark

hand, actions make some computations on RDDs. The results of these computations are either sent to the driver program or stored in external storage (e.g., HDFS). When a spark application is launched, the scheduler creates one or more Spark jobs. More, precisely, the scheduler creates a job for each action of an application. Therefore, a job contains numerous transformations and exactly one action. In order to execute a job, the scheduler builds a directed acyclic graph (DAG) of stages based on the lineage graphs of all RDDs used in this job. A new stage is started when a transformation of the previous stage requires shuffling of data. Therefore, a stage can contain multiple transformations executed in pipeline, but at most one transformations requiring shuffling of data. Most of the transformations can be executed in pipeline. However, there are transformations that can be executed in pipeline when each partition of parent RDD is used by at most one partition of child RDD (narrow-dependency). However, when the partitions of the parent RDD is used by more than one partition of child RDD (wide-dependency), shuffling of data is required. Finally, each stage is divided into tasks. A task is created for each partition of parent RDD and operation of a stage. Each task contains a partition of an RDD and an operations of its stage. Then, the scheduler assigns each task to one executor of a worker node based on data locality.

Moreover, Spark allows persisting RDDs in worker nodes in order to avoid re-computing them in future stages. Users can define the desired storage level (e.g., Memory only, Disk only) based on his needs. In addition, Spark supports sharing variables from the driver program to all worker nodes. There are two types of shared variables available in Spark, the broadcast variables and the accumulators, both cached in main memory of all worker nodes. Broadcast variables can be read from an operation without being sent from the driver program. On the other hand, accumulators are used for aggregating values across all worker nodes and sending them to the driver program. Accumulators cannot provide only-one guarantee when they are updated with a transformation. For this reason, accumulators are not reliable when they used with transformations. Local variables, required for a transformation, are sent from the driver program to all worker nodes. Finally, users can specify how to partition an RDD across worker nodes by declaring the number of partitions and the partition function.

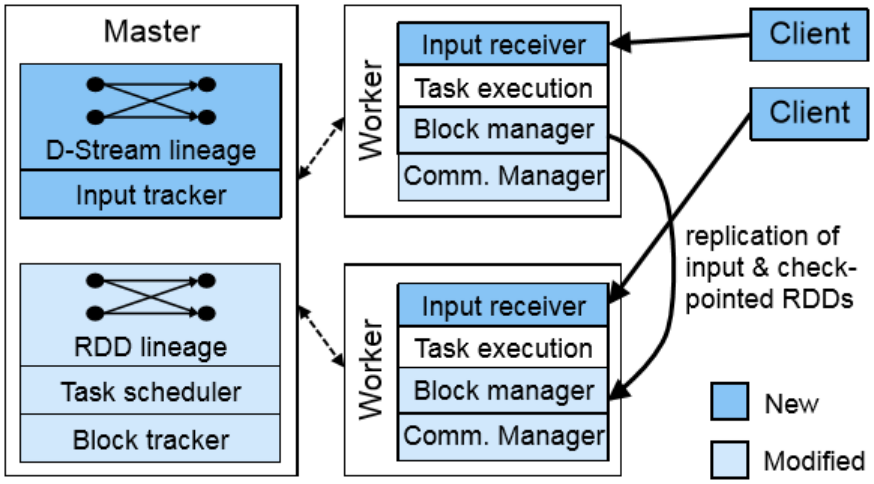


FIGURE 3.2: The architecture of the Spark Streaming platform

3.2 Apache Spark Streaming

The Spark Streaming platform is built on top of Apache Core and provides scalable, high-throughput and fault-tolerant distributed processing of data streams. Data streams are considered as an unbounded sequence of RDDs, called Discretized Streams (DStreams). New information of data streams is received from receivers and is stored in main memory of worker nodes. Then, at each fixed user-defined time interval, known as batch interval, an RDD is created in order to include new information in a DStream. Therefore, a DStream is a collection (HashMap) of (Time,RDD) records. Each RDD in a Dstream contains data from a certain interval. Numerous operations can be applied on DStreams and, similarly to Core, are divided into transformations and output operations. A DStream is created either form external data sources or from a transformation on other DStreams. The architecture of Spark Streaming illustrated in figure 5.1 (*source:* [76]) is based on Spark's architecture. To this end, at each batch interval, a job is created for each output operation of a Spark Streaming application. Each job contains numerous transformations and exactly one output operation action. The scheduler builds a DAG of states based on lineage graphs. Therefore, a stage can contain multiple transformations executed in pipeline, but at most one transformations requiring shuffling of data. Finally, each stage is divided into tasks. Each task contains a partition of an RDD and all operations of its stage. Then, the scheduler assigns a task to an executor of a worker node based on data locality. Then, the scheduler assigns a task to an executor of a worker node based on data locality.

Since a DStream is a sequence of RDDs, transformations can be performed either on a fraction of RDDs or only on the new one. In this regard, transformations are divided into stateless and stateful. Stateless transformations are applied separately on each new RDD. Stateful transformations demand information of previous RDDs. Window stateful transformations require previous RDDs during a sliding window. State tracking stateful transformations require only the previous RDD, representing the state. Moreover, DStreams can be consumed from an external source by applying output operations on them.

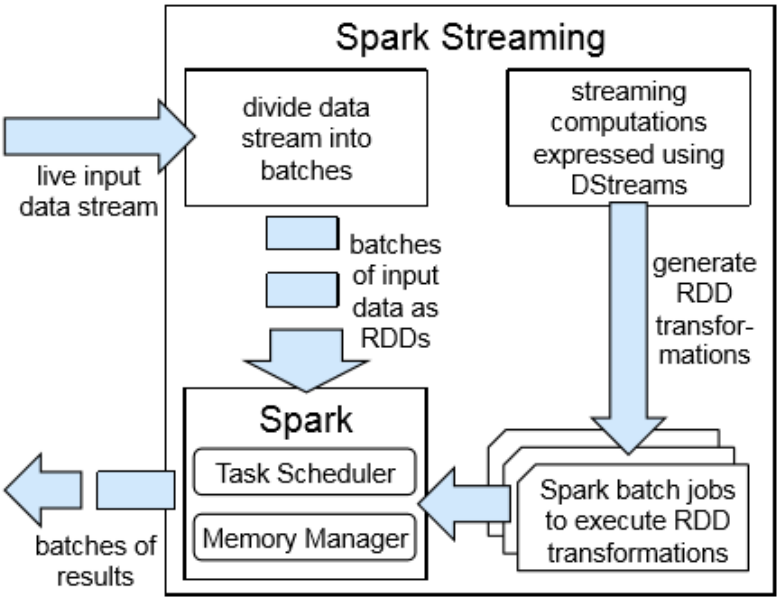


FIGURE 3.3: An overview of the Spark Streaming platform

Therefore, stateless transformations and output operations on DStreams are similar to transformations and actions on RDDs, respectively. In this regard, all transformations on DStreams follow a lazy evaluation triggered by an output operation. Shared variables and persisting DStreams in a user-defined level storage are also available. Further, each RDD recover from its lineage graph in stateless transformations. However, lineage graphs may be very large in stateful transformations and thus traversing a lineage graph is inefficient. DStreams recover based on checkpoint mechanism in case of node failure. Checkpoint mechanism allows storing periodically in reliable storage all RDDs of DStream required in case of node failure. An overview of the Spark Streaming platform is illustrated in figure 3.3 (*source:* [76])

The stateful transformation *UpdateStateByKey* generates a new DStream by combining previous values of a state with values of a new RDD. In order to achieve that, new information is formed as a DStream key-value pair, identical to the DStream pair of a state. All records of a state are stored in main memory of worker nodes and in a reliable storage (checkpoint mechanism). All new information, arrived at each batch interval, is shuffled over the cluster's network in order to update the previous version of a state based on a user-defined function. An update function is applied to each record of a state, regardless of whether new information contains a value for a record. To this end, an optimization of *UpdateStateByKey* is available in the Spark Streaming platform, the stateful *MapWithState* transformation. In contrast with the *UpdateStateByKey* transformation, *MapWithState* applies an update function only to the records of a state that their keys are contained in a new RDD. Consequently, updating a state with *MapWithState* is significantly less time-consuming than updating it with *UpdateStateByKey*. *MapWithState* can support a greater number of state values as well. Therefore, *"MapWithState can provide 6X lower latency and maintain state for 10X more keys than when using UpdateStateByKey"* according to the

Databricks company.

In our implementation we use the *MapWithState* stateful transformation as well as numerous operations described below. The stateless *Map* transformation applies an 1-1 function to all elements of a new RDD in a DStream. The stateless *FlatMap* transformation is similar to *Map*, but it returns zero or more outputs for each element. The stateless *ReduceByKey* transformation can be used only for DStream pairs and aggregates the values of each key for each new RDD based on a reduce function. When *ReduceByKey* is applied on an RDD, which is narrow dependent with its parent RDD (e.g., hash-partitioned parent RDD), shuffling of data is not required. When an RDD is wide dependent with its parent RDD, then *ReduceByKey* demands shuffling of data over the cluster whether the parent RDD is not hash-partitioned based on the key. The stateless *GroupByKey* transformation follows the same approach of *ReduceByKey*, without applying a reduce function to aggregated values. The *GroupByKey* also requires shuffling when the parent RDD is not hash-partitioned based on key. Finally, *ForeachRDD* output operation applies a user-defined RDD action to all RDDs of a DStream.

The Apache Spark framework includes another library for distributed data streaming processing called Structured Streaming. Structured Streaming is built on the Spark SQL engine and capitalizes on its execution optimizations. The approach of Structure Streaming is to handle a data stream as an unbounded Dataframe. An unbounded Dataframe is increased horizontally in size when a new instance arrives. In contrast, Spark Streaming handles data streams as a sequence of RDDs. For this reason, Structured Streaming is more high-level and accessible from programmers than Spark Streaming. Moreover, Structured Streaming handles out-of-order data based on watermarks and event-time analysis. Triggers and batch-stream joins are also available. Despite the advantages of Structure Streaming, it does not contain any implementation of machine learning algorithms. On the other hand, Spark Streaming includes various implementations of incremental machine learning algorithms. Therefore, we preferred to implement the Hoeffding decision tree classifier in the Spark Streaming platform in order to extend the existing work of Spark's community.

3.3 Decision trees in Apache Spark

Spark is efficient at iterative computations as RDDs can be persisted in main memory for future use. Therefeore, Spark is well-suited for large-scale machine learning applications. To this end, a distributed machine learning library, called Machine Learning library [50] (MLlib), has been implemented on top of Spark Core. Various machine learning algorithms for supervised, unsupervised and deep learning are included in MLlib library. Numerous methods for data-preprocessing, distributed linear algebra and statistical analysis are included in MLlib as well. MLlib library allows programming on two different APIs. The first one is based on RDD abstraction and it is more suitable for programming by scratch; since it contains various data structures for local and distributed matrices and vectors. Linear algebra operators are executed by Breeze library. Also, RDD-based API contains streaming implementations of significant testing, linear regression and k-means algorithms. The other API is based on DataFrame data structure and follows a more high-level approach

for programming machine learning algorithms influenced by scikit-learn library. It is able to program a machine learning work-flow by constructing a pipeline. Each machine learning pipeline consists of a chain of transformers and estimators as well as parameters and DataFrame inputs. Transformer algorithms transform a Dataframe into a different one. An Estimator algorithm receives a training data set and produces a machine learning model. However, there is no implementation of streaming algorithms in this API since the Spark Streaming library is based on RDDs.

Various supervised machine learning methods for classification are available in both libraries such as support vector machines, logistic regression and decision trees. Decision trees classifiers and their ensemble methods, Random forest and Gradient-boosted trees, have been implemented in both libraries. These implementations are based on PLANET framework [54] and allow users to define a variety of parameters (e.g., evaluation metric, stopping criterion). PLANET proposes a parallel implementation of decision trees in a cluster computer platform based on MapReduce programming model. Classifications trees grow in breadth-first manner (level-by-level).

As described previously, a MapReduce job consists of a map and a reduce phase. A MapReduce job is created on each iteration of the tree learning algorithm. On each iteration, each worker node of the cluster creates a mapper. Each mapper consists of the up-to-date classification tree and an horizontal partition of a training set. A mapper scans from HDFS its corresponding partition. Then, it finds the leaf node of each training instance by traversing the tree. A counter for each possible combination of leaf nodes, attributes, values, and classes ($\langle leaf, attr, val, class\rangle$ quadruple) is computed. Finally, each mapper outputs a key value pair with $key : \langle leaf, attr\rangle$ and $value : \langle val, class, aggr\rangle$. All information for each possible leaf node and attribute is sent to the same reducer. Then, each reducer find the best split for its corresponding $\langle leaf, attr\rangle$ double. Each reducer outputs information about the best split for its corresponding attribute $attr$ in a leaf node $leaf$. This information is stored in HDFS. Finally, the master node scans all this information and find the best split for each leaf node. Then, it decides when to split a node based on a splitting criterion and updates the classification tree. The up-to-date classification tree is distributed to all worker nodes in order to repeat the procedure until a stopping criterion is satisfied.

Classification trees, implemented in Spark MLlib, are similar to PLANET ones. However, a partition of a training set, fitting in main memory, is persisted in main memory of a worker nodes. Therefore, scanning of a partition from secondary storage (HDFS), which is a very expensive operation in terms of I/O costs, is omitted. In case of a partition does not fit in memory, a fraction of it is persisted in secondary storage. Furthermore, all information regarding a leaf node is accumulated in a worker node through *ReduceByKey*. Sequentially, splitting decisions are executed in parallel on worker nodes, since information required to decide a split on a leaf node is in the same worker node. Therefore, the master node collects all splitting decisions and update the classification tree. Further, splitting points of numerical attributes are defined by an equidepth histogram computed before the tree learning method.

Chapter 4

Hoeffding Decision Trees in the Spark Streaming Platform

4.1 Our Contribution

In this thesis, we propose a parallel implementation of Hoeffding decision trees in the Spark Streaming platform. In our approach, all information, required for evaluating splits, is stored in main memory of worker nodes. This information, called global statistics, contains a counter for each possible combination of leaf nodes, attributes, values of attribute, and classes. Global statistics can be also considered as counters for all $\langle leaf, attr, val, class \rangle$ quadruples. All new training instances, arrived during a batch interval, are horizontally partitioned across worker nodes. Each worker node computes a counter for each $\langle leaf, attr, val, class \rangle$ quadruple, appearing in its partition. These counters, appeared in a worker node during a batch interval, are called local statistics. Therefore, each worker node, containing new training instances, has its own local statistics. Then, all local statistics are aggregated in order to obtain aggregated local statistics during a batch interval. Global statistics are updated through a stateful operations, in order to include aggregated local statistics, generated during a batch interval. All computations for evaluating splits and finding the best one for each leaf node and attribute ($\langle leaf, attr \rangle$ double) are locally executed in parallel. Splitting decisions based on the Hoeffding bound are made for each leaf node in parallel as well. All splitting decisions are sent to the driver program (master node) in order to update the Hoeffding tree. The driver program holds only an up-to-date tree-structure of the Hoeffding tree. The driver program splits the Hoeffding tree and updates the majority classes whenever needed. Unseen instances are horizontally partitioned across worker nodes. All worker nodes use the up-to-date Hoeffding tree in order to classify the unseen instances of their partition.

The proposed implementation stores global statistics of each leaf node and attribute in a sparse data structure. The counters of each $\langle leaf, attr \rangle$ double is stored in a sparse matrix. Global statistics are partitioned across worker nodes based on $\langle leaf, attr \rangle$ doubles (hash-partitioning). The number of rows and columns of a sparse matrix for a $\langle leaf, attr \rangle$ double correspond to the number of values taking the attribute and the number of classes, respectively. Training instances, labelled with the same class, are more likely to be classified in the same leaf node as the Hoeffding tree becomes more and more accurate. This assumption is based on that a Hoeffding tree is asymptotically identical to a batch-learning one. In this regard, a matrix containing all possible combinations of values and classes for an attribute in a leaf node, become more and more sparse as the classification tree grows. Further, only a limited fraction

of global statistics is updated during a time interval. Holding global statistics in main memory is expensive in terms of space complexity and demands large amounts of main memory. Especially, for high-dimensional data sets (e.g., hyper-spectral data cubes) and multi-class classifications problems (e.g., color classification), sparse matrices can significantly reduce the size of main memory required for storing global statistics.

In our implementation, a stateful *MapWithState* transformation is used for updating global statistics. This operation, described in the previous chapter, allows us to maintain a state of global statistics in main memory and update it when new aggregated local statistics are available. At each batch interval, aggregated local statistics are added to global statistics. However, only a limited fraction of global statistics is updated during a time interval since aggregated local statistics rarely include information of all $\langle leaf, attr \rangle$ doubles. Therefore, all information of each $\langle leaf, attr \rangle$ double is stored as a sparse matrix in main memory. Further, global statistics of a $\langle leaf, attr \rangle$ double are deleted from main memory when no new information of this double has arrived during a user-defined time interval. Sequentially, global statistics of interior nodes and obsolete leaf nodes are deleted from main memory. In case of an obsolete $\langle leaf, attr \rangle$ double, its global statistics are re-initialized when new information of this double is available. Therefore, the deactivation mechanism, proposed in VFDT, is implemented without communicating with the master node. Our implementation includes the tie-breaking mechanism, proposed in VFDT, as well. Numerical attributes are handled by constructing a fixed-size histogram with B bins based on the first B observations (values) of each numerical attribute. Finally, the user-defined parameters, required for splitting decision based on the Hoeffding Bound (δ, R) and on the tie-breaking mechanism (τ, n_{ties}), are distributed across worker nodes only once (broadcast variables).

Our implementation follows a similar approach to the batch-learning classification tree learning method, implemented in Spark MLlib. In our case, a training set is a fraction of a data stream of training instances. Both implementations partition a training set horizontally. Moreover, in both implementations splitting decisions for each leaf node are made in parallel. However, the evaluation of attribute splits is executed in parallel in the proposed implementation. In contrast, the evaluation of leaf nodes splits is executed in parallel in the batch-learning implementation. Therefore, our implementation is more scalable than the batch-learning. However, the proposed implementation demands shuffling of data in order to accumulate all best attribute splits in the same worker node. In contrast, the batch learning implementation avoids this shuffling of data, since all best attributes splits of a leaf node are in the same worker node. Further, numerical attributes are handled by constructing a fixed-size histogram before training the model in batch-learning method. In this regard, only the boundary values of the fixed-size histogram are considered as candidate splitting points and they are immutable. On the other hand, numerical attributes are handled by constructing a fixed-size histogram with B bins based on the first B observations of each numerical attribute.

Furthermore, our implementation is more scalable compared to the other existing parallel implementation of classification trees for data streams, known as SPDT [5]. In this implementation, new training instances are also horizontally partitioned across worker nodes. However, each worker node approximates its local statistics by building an on-line histogram for all values of

each $\langle leaf, attr, class \rangle$ triple. Then, all approximations of local statistics are sent to the master node. The master node holds an approximation of global statistics, consisting of a histogram for each $\langle leaf, attr, class \rangle$ triple. Therefore, the histograms of global statistics are merged with the histograms of all local statistics. Candidate splits are estimated based on approximated global statistics. The master node evaluates all estimated candidate splits and decides when to split a node. In contrast, the proposed implementation stores global statistics, based on a sparse data structure, across worker nodes and only a splitting decision is sent to the master node. Further, our implementation enumerates all possible candidate splits based on exact global statistics. As numerical attributes are handled by constructing a fixed-size histogram during the building phase, only the boundary values are considered as candidate splitting points and they are immutable.

Therefore, the proposed implementation scales out more efficiently than SPDT due to the fact that only splitting decisions are sent to the master node. In the SPDT, as the classification tree grows, data shuffling and the workload of the master node increase as well. Therefore, it demands a large volume of data shuffling over the cluster's network and a large number of computations in the master node. Moreover, estimating candidate splits, used in SPDT, is more efficient than the exact approach, used in our implementation. However, enumerating all possible candidate splits based on exact global statistics is more effective in terms of accuracy. SPDT also stores only an approximation of global statistics in the master node. In contrast, exact global statistics are stored across worker nodes in our implementation. Further, the size of main memory, needed to store exact global statistics, is significantly reduced by using sparse matrices based on the assumption that a Hoeffding tree becomes more and more accurate as the Hoeffding tree grows.

The proposed implementation follows a similar approach with VHT [43]. However, VHT is based on a different shared-nothing architecture, the Apache SAMOA distributed streaming platform [51, 42]. In this implementation, new training instances are vertically partitioned across worker nodes. Only one worker node holds the counter for each $\langle leaf, attr, val, class \rangle$ quadruple. Thus, global statistics are computed in parallel when a splitting decision is required. The evaluation of splits is performed in parallel as well. However, the splitting decision is made in the master node. In our implementation, training instances are horizontally partitioned, but global statistics are informed instantly for each leaf node and attribute by aggregating local statistics. Moreover, splitting decisions are made in parallel. A splitting decision is made whenever a new batch of training instances arrives. The batch interval is a user-defined variable.

In the below sections, the proposed implementation is presented in more detail. More precisely, the procedure of updating the global statistics, whether a new batch of training instances is available, is presented in 4.2. The distributed computations for evaluating the splits, finding the best one and deciding whether to split a node are presented in 4.3. Finally, the implementation in the Spark Streaming platform is presented in 4.4.

4.2 Updating the Global Statistics

In the proposed implementation, a data stream of training instances is modeled as a DStream. A Dstream, as described in the previous chapter, is a sequence of RDDs. Each RDD contains all training instances arrived at the receivers of worker nodes during a batch interval. At each batch interval, training instances of the new RDD are horizontally partitioned across the cluster. A training instance, included in a partition of a worker node, traverses the up-to-date Hoeffding tree H, until a leaf node $leaf$. To this end, the master node sends all information, required for traversing the tree, to all worker nodes. All training instances contain values for their fixed-size n attributes and a class label. Therefore, a training instance, labeled as $class_j$, generates n $\langle leaf, attr, val, class_j \rangle$ quadruples. A *FlatMap* operation transforms a training instances to n key-value pairs. Each key-value pair consists of a key : $\langle leaf, attr_n, val_{ni}, class_j \rangle$ and a $value$: $\langle 1 \rangle$, where $attr_n$ represents the attribute of a training instance and val_{ni} its value. Each key-value pair of a training instance shows that this training instance belongs to leaf node $leaf$ and is labeled as $class_j$. Moreover, it shows that it contains an attribute $attr_n$ with value val_{ni}.

The counters of all $\langle leaf, attr, val, class \rangle$ quadruples in the batch are computed by applying a *ReduceByKey* transformation with hash-partitioning based on key to the DStream. Firstly, the *ReduceByKey* operation aggregates locally all appearances of each $\langle leaf, attr, val, class \rangle$ quadruple in a worker node during the current batch interval. Therefore, a counter for each $\langle leaf, attr, val, class \rangle$ quadruple is computed. These counters are the local statistics of each worker node during the current batch interval. Then, *ReduceByKey* partitions (hash-partition) all local aggregated key-value pairs based on their keys across the cluster. All key-value pairs with the same key are sent to the same worker node. A counter of each $\langle leaf, attr, val, class \rangle$ quadruple is computed by aggregating the local aggregated key-value pairs. These counters are the aggregated local statistics of all worker nodes during the current batch interval and data shuffling is required in order to compute them. Consequently, *ReduceByKey* transforms each key : $\langle leaf, attr_n, val_{ni}, class_j \rangle, value$: $\langle 1 \rangle$ to key : $\langle leaf, attr_n, val_{ni}, class_j \rangle, value$: $\langle aggr_n \rangle$, where $aggr_n$ is the number of occurrences of $\langle leaf, attr_n, val_{ni}, class_j \rangle$ quadruple during the current batch interval.

The proposed implementation stores all counters of each $\langle leaf, attr \rangle$ double, computed throughout the building phase, as a sparse matrix. In order to achieve that, global statistics are stored as a DStream pair, where key : $\langle leaf, attr \rangle$ and $value$: $SparseMatrix(val, class)$. For example in a binary classification problem, the counter of all training instances in the 1st leaf node, classified to the 4th class, where its 2nd attribute takes the 3rd value, is 5 throughout the building phase. This information, included in global statistics, is the entry value $SparseMatrix(3,4) = 5$ of $\langle 1,2 \rangle$ key. The key-value pair containing this entry value is key : $\langle 1,2 \rangle, value$: $SparseMatrix(3,4)$. However, if the above example corresponds to the counter in all worker node during a batch interval, then this information is formed as key : $\langle 1,2 \rangle, value$: $\langle 3,4,5 \rangle$.

Sequentially, a *Map* operation is applied to the DStream, containing the aggregated local statistics during the current time interval, in order to change the key of all key-value pairs. Since the key is changed, all partitioning information, generated by *ReduceByKey*, is forgotten. Next, all information regarding each possible combination of leaf nodes and attributes is accumulated, as a

list of $\langle val, class, aggr \rangle$ triples, in the same worker node by applying a *GroupByKey* operation based on key. Shuffling of data is required in order to accumulate all information of each $\langle leaf, attr \rangle$ to the same worker node. Data shuffling could be avoided in another architecture by hash-partitioning based either on $\langle leaf, attr_n \rangle$ or on $\langle leaf \rangle$ in both *ReduceByKey* and *GroupByKey* operations. However, partitioning information of RDDs, produced by *ReduceByKey*, has been forgotten in Spark, since *map* operation changes the key. Generally, a *map* transformation does not preserve partitioning information, even if it does not change the key. In this regard, the stateless *MapValues* transformations exists in order to preserve partitioning information of pair RDDs.

Global statistics, needed to evaluate all attribute splits for each $\langle leaf, attr_n \rangle$ double, are stored as a state DStream. The state represents the occurrences of each $\langle leaf, attr, val, class \rangle$ quadruple, until the previous batch interval. A state is represented by key-value pairs, where $key : \langle leaf, attr \rangle$ and $value : SparseMatrix(val, class)$. Each entry (i, j) of a sparse matrix, containing in a pair with key $key : \langle leaf, attr \rangle$, represents the counter of $\langle leaf, attr, i, j \rangle$ quadruple. A state DStream is needed to be stored as $HashMap[Time, RDD]$ in main memory across worker nodes and be processed in main memory as well. The states are only stored in secondary storage for achieving fault-tolerance as described in the previous chapter. Therefore, holding the global statistics across the cluster demands large amounts of main memory. In order to reduce the size of main memory, required for storing the global statistics, they are stored as sparse matrices. The entry values of a $SparseMatrix(val, class)$ are stored in Compressed Sparse Column (CSC) format. For example, consider a multiclass problem with 4 classes (A,B,C,D) and an attribute of a leaf node taking values from $\{1, 2, 3, 4, 5\}$. Suppose that 60 training instances, labelled as A, and 10 training instances, labelled as C, have been classified to that leaf node. In case of the training instances labelled as A, the attribute takes 45 times the value 1 and 15 times the value 5. In case of the training instances labelled as C, the attribute takes 10 times the value 3. The global statistics of this attribute are described by the the following sparse matrix $SparseMatrix(5, 4)$

$$P = \begin{bmatrix} 45 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 10 & 0 \\ 0 & 0 & 0 & 0 \\ 15 & 0 & 0 & 0 \end{bmatrix}$$

However, this matrix is stored by holding only $A = \begin{bmatrix} 45 & 15 & 10 \end{bmatrix}$, $Row = \begin{bmatrix} 0 & 2 & 4 \end{bmatrix}$, $Col = \begin{bmatrix} 0 & 2 & 2 & 3 & 3 \end{bmatrix}$. A matrix contains the non-zero values in column-major order. *Row* matrix contains the corresponding rows of each non-zero values. Finally, let n_j be the number of non-zero values on $j - th$ column and the number of columns be k (4 in our example), then

$$Col[j] = \begin{cases} 0 & for\ j = 0 \\ Col[j-1] + n_{j-1} & for\ j = 1, ..., k \end{cases}$$

The aggregated local statistics in the current batch interval, generated by

GroupByKey, as well as the global statistics, throughout all previous batch intervals, can be merged with a *MapWithState* stateful transformation. More precisely, the aggregated local statistics are represented by key-value pairs, with $key : \langle leaf, attr \rangle$ and $value : List[val, class, aggr]$ for all combinations of leaf nodes, attributes, values, and classes, appeared in the current batch. The global statistics are also represented by key-value pairs, with $key : \langle leaf, attr \rangle$ and $value : SparseMatrix(val, class)$ for all possible combinations of leaf nodes, attributes, values, and classes. The aggregated local statistics of a leaf node and an attribute are stored in main memory of the worker node, in which the state of the leaf node and the attribute is stored. Therefore, data shuffling is not required. The update function adds the counter $aggr_n$ of a $\langle leaf, attr_i, val_{ni}, class_j \rangle$, contained in the list, to the entry value of $SparseMatrix(val_{ni}, class_j)$ of the pair with $key : \langle leaf, attr_i \rangle$, contained in the state. In order to update a $SparseMatrix$, it is converted to a $DenseMatrix$. In $DenseMatrix$, the entry values are stored in a single array of doubles with columns listed in sequence. When the update is finished, $DenseMatrix$ converts to $SparseMatrix$ again. Further, when a $key : \langle leaf, attr \rangle$ is contained in the aggregated local statistics but not in the state, then a new pair with state $key : \langle leaf, attr \rangle$ is initialized in the state. When a $key : \langle leaf, attr \rangle$ is not contained in the aggregated local statistics during a user-defined time-interval, but the state includes a pair with $key : \langle leaf, attr \rangle$, the pair is deleted from the state. This mechanism deletes global statistics of obsolete leaf nodes or interior nodes. In case of obsolete leaf nodes, global statistics can be re-initialized when the aggregated local statistics contain information for it.

The data flow diagram for updating the statistics is illustrated in 4.1. In this figure, the red arrows represent the operations needed shuffling of data across the cluster, while the white arrows represent the operations that do not require shuffling. The blue arrows represent the operations that would avoid shuffling of data in another platform if they were hash-partitioned based either on $\langle leaf, attr_n \rangle$ or on $\langle leaf \rangle$. However, as described previously, shuffling of data could not be avoided in Spark.

4.3 Updating the Hoeffding Decision Tree

By storing all counters of each $\langle leaf, attr \rangle$ double as a sparse matrix, all information required for evaluating attribute splits is locally available. In order to evaluate attributes splits, $B - 1$ splits are examined for a numerical attribute, taking B values, in each leaf node. As described previously, a fixed-sized one-dimensional histogram is built for each numerical attribute based on B first observations. Therefore, the boundary values are these B first observations and all midpoints of two successive boundary values are considered as candidate splits. A split on a numerical attribute $attr_n$, taking val_{ni} value, is formed as $val_{ni} \leq mid$, where mid is the midpoint value of two successive boundary values of the histogram of attribute $attr_n$. On the other hand, $2^B - 2$ splits are examined for a categorical attribute, taking values from a set of B values, in each leaf node; because all possible subsets of this set, except for the empty set and the set containing all values, are considered as candidate splits. The candidate split with the highest information gain $info_{best}$ is considered as the best attribute split $split_{best}$ of $attr_n$. The number of instances N_{leaf} and the majority

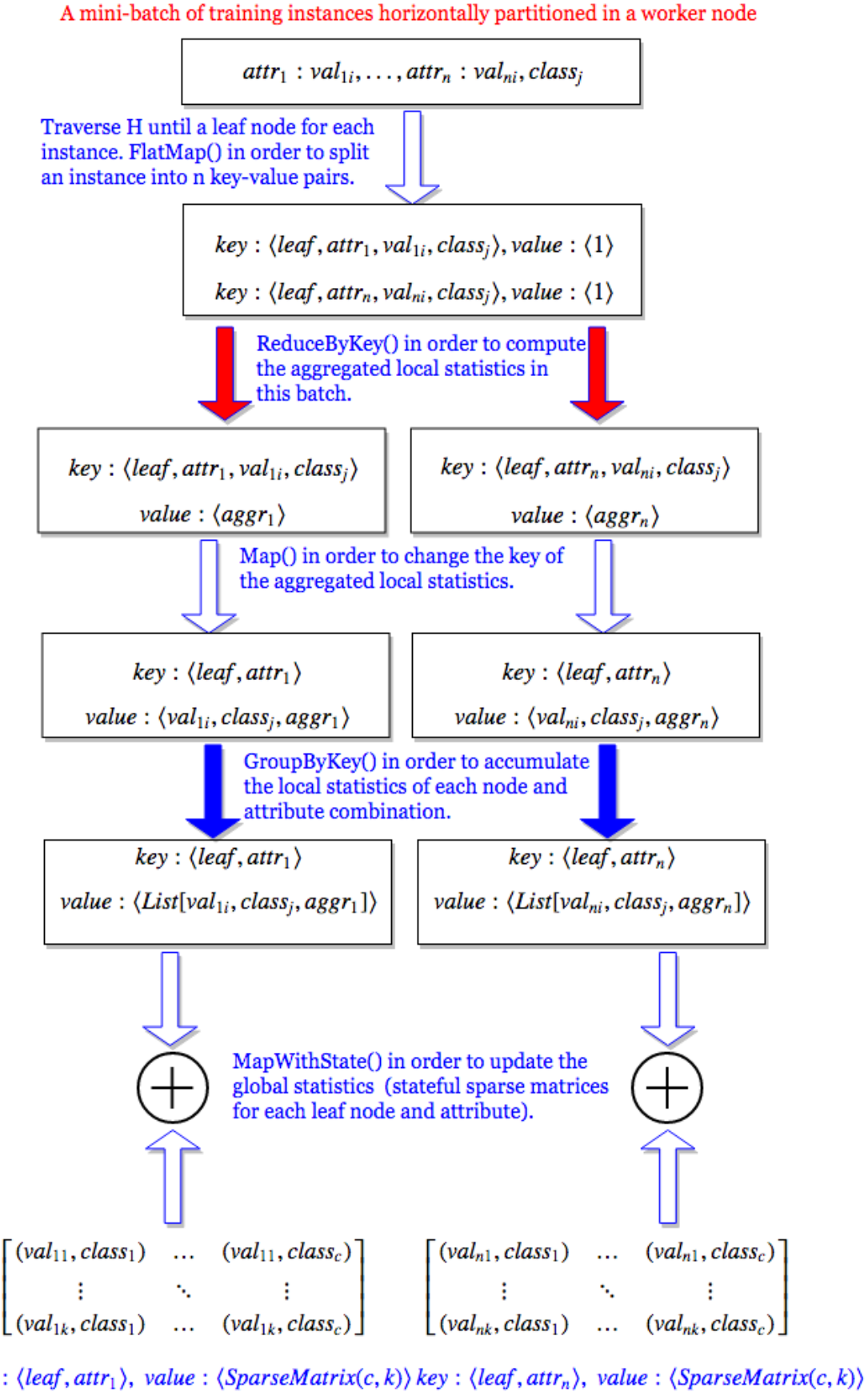


FIGURE 4.1: The data flow diagram for updating the global statistics

class $class_{leaf}$ in each leaf node are also computed from the matrices in parallel. This information is modelled as a Dstream key-value pair. The key holds all information regarding a leaf node and the value holds all information regarding the best attribute split of an attribute in this leaf node. More precisely, the key and the value derived from the global statistics of each leaf node and attribute is $key : \langle leaf, N_{leaf}, class_{leaf} \rangle$ and $value : \langle attr_n, split_n, info_n \rangle$. This output is generated by applying a *Map* operation to the snapshot of the state DStream.

Then, a *GroupByKey* operation with hash partitioning based on key is applied in order to accumulate the best attribute splits for each leaf nodes and find the two attributes with the highest $\overline{G}(X_a)$ and the second highest information gain $\overline{G}(X_b)$. Data shuffling could be avoided in this operation by hash-partitioning based on $\langle leaf \rangle$ in previous *ReduceByKey* and *GroupByKey* operations. However, partitioning information of RDDs, produced by *MapWithState*, has been forgotten since *Map* operation has changed the key of the snapshot of state. Moreover, the range R as well as δ are stored as broadcast variables and they are distributed only once across worker nodes for read-only usage. Therefore, all information required to decide whether to split a leaf node based on the Hoeffding bound is available in the same worker node. More precisely, the highest information gain $\overline{G}(X_a)$, the second highest information gain $\overline{G}(X_b)$, the number of instances N_{leaf} in a leaf node, the range R and δ are available in main memory of the worker nodes. So, a *Map* operation is used in order to derive a splitting decision for each leaf node in parallel. When $split = -1$, the inequality $\overline{G}(X_a) - \overline{G}(X_b) > \epsilon$ does not hold and as a consequence the corresponding leaf node does not split. Otherwise, the leaf node is split based on the $split_{best}$ of $attr_{split}$ attribute.

Moreover, the tie breaking mechanism, proposed in VFDT, has been implemented. When $\overline{G}(X_a) - \overline{G}(X_b) < \epsilon < \tau$ and $N_{leaf} \geq n_{ties}$ in a leaf node, there is a tie between X_a and X_b. This mechanism prevents the Hoeffding tree from not splitting when the attributes X_a and X_b have almost identical information gain measures after n_{ties} observations in a leaf node. In this case, the leaf node is split based on the attribute with the highest information gain X_a. The number of instance n_{ties}, assumed enough in order to decide that there is a tie, and τ, are user-defined variables. In the proposed implementation, τ and n_{ties} are stored as broadcast variables and they are distributed only once across worker nodes for read-only usage. Finally, a *ForeachRDD* output operation applies *Collect* RDD action to a new RDD of the DStream, containing splitting decision at each new batch interval. The *Collect* action collect to the master node (driver program) all splitting decisions and the new majority class of each leaf node. Then, the driver program splits the Hoeffding tree and updates the majority classes whenever needed. More precisely, splitting decisions are formed as $\langle leaf, attr_{split}, class_{leaf} \rangle$ triples. When $attr_{split} = -1$ then, the leaf node $leaf$ is not split. The leaf node $leaf$ is assigned to the class $class_{leaf}$. Otherwise, the leaf node $leaf$ is split based on $attr_{split}$ and two child leaf nodes are created. Both child leaf nodes are not assigned to a class, until a training instance be classified to them.

The tree-model is saved on secondary storage after the update phase of each batch. The Hoeffding tree can be also initialized by loading this information from secondary storage. In the prediction phase, described in the previous section, the unseen instances are horizontally partitioned and each worker node contains an up-to-date model of the Hoeffding tree. Therefore, each new

unseen instance in this worker node traverses the tree until it reaches to a leaf node. Then, the unseen instance is classified to the majority-class of this leaf node. When this leaf node is not assigned as no training instance has not been classified to it yet, the unseen instance is classified to the majority-class of its parent interior node. Therefore, the proposed implementation allows making predictions throughout the building phase.

The data flow diagram for updating the Hoeffding decision tree is illustrated in 4.3. In this figure, the red arrows represent the operations needed shuffling of data across the cluster, while the white arrows represent the operations that do not require shuffling. The green arrows represent the operations that would avoid shuffling of data if all operations were hash-partitioned based on $\langle leaf \rangle$. However,shuffling of data could not be avoided in Spark; because the previous operations have changed the key of future DStreams and as a consequence the previous partitioning information has been forgotten.

4.4 The proposed implementation in Spark Streaming

Generally, a Spark streaming application in the Spark Streaming platform consists of a sequence of jobs. At each fixed time interval, known as batch interval, a new job is created for each output operation of a Spark streaming application. As described in the previous chapter, every job in Spark Streaming contains numerous transformations and exactly one output operation. In order to execute a job, the scheduler builds a directed acyclic graph (DAG) of stages based on the lineage graphs of each RDD of the Dstream, used in this job. A new stage is started, when a transformation of the previous stage requires shuffling of data. Therefore, a stage can contain multiple transformations executed in pipeline, but at most one transformation requiring shuffling of data. Finally, each stage is divided into tasks. Each task contains a partition of an RDD and all operations of its stage. Then, the scheduler assigns a task to an executor of a worker node based on data locality.

The proposed implementation, presented in the previous sections, is a Spark streaming application. At each batch interval, a new job is created, corresponding to the *ForeachRDD* output operation. The *ForeachRDD* output operation applies the *Collect* RDD action to the new RDD, containing all splitting decision during a batch interval. In our implementation, each job consists of 4 stages. A job of the proposed streaming application is presented in figure 4.3.

The first stage is performed in order to read new training instances from Hadoop Distributed File System (HDFS) and aggregate all local statistics generated during a batch interval. It starts when the *textFileStream* transformation is executed and it ends when the *ReduceByKey* transformation is executed. When new files, containing new training instances, are created in a directory of HDFS during a batch interval, the *TextFileStream* includes the new training instances in the new RDD of the DStream. However, *TextFileStream* is not able to detect changes in a file and thus it is mandatory to store new training instances as a new file in a directory of HDFS.

The second stage is used for sending aggregated local statistics to the worker nodes containing global statistics. It starts after the execution of the *ReduceByKey* transformation and it ends when the *GroupByKey* transformation is executed. These jobs could be merged to one job by replacing the *ReduceByKey* and the *GroupByKey* transformations with a *ReduceByKey* transformation with

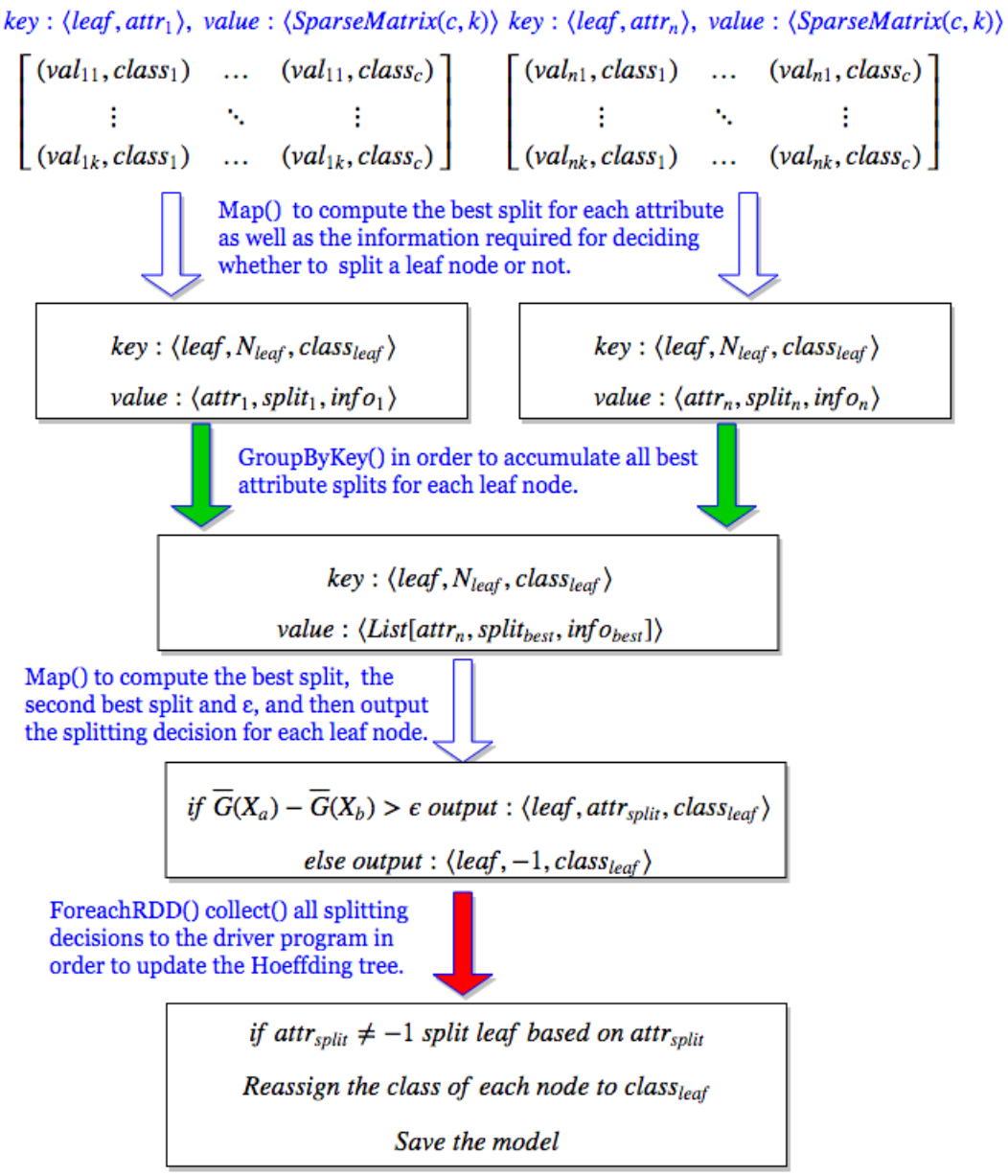


FIGURE 4.2: The data flow diagram for updating the Hoeffding decision tree

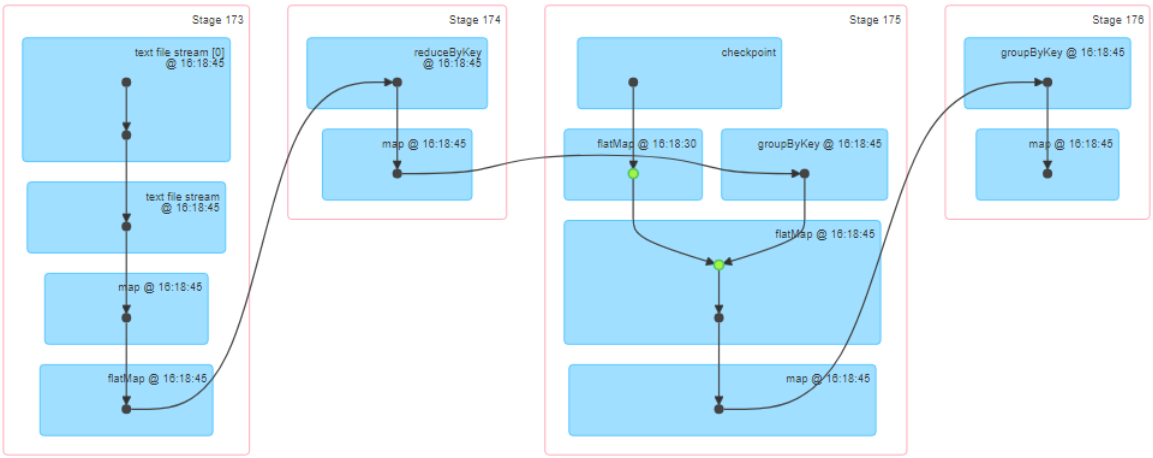


FIGURE 4.3: An execution plan of a Spark job in the proposed implementation

hash-partitioning based on $\langle leaf, attr \rangle$. However, in this case, large amounts of data would send to the worker nodes holding global statistics. Sequentially, worker nodes would use a large amount of main memory to store this information. Further, these worker nodes would perform a large number of computations. Therefore, we prefer to perform two jobs in order to accumulate aggregated local statistics and global statistics of each leaf node and attribute in the same worker nodes.

The third stage is used for updating global statistics, evaluating all attribute splits, and aggregating all best attribute splits of each leaf node. It starts after the execution of the *GroupByKey* transformation and it ends when another *GroupByKey* transformation is executed. The latter one is performed in order to accumulate all best attribute splits of each leaf node in the same worker node. This stage is by far the most time-consuming of our implementation. Further, the third stage in the current job is independent of the first and the second stage of future jobs. When the third stage has not finished, Spark Streaming executes the first and the second stages of the pending jobs. Therefore, the aggregated local statistics have already been shuffled in the worker nodes containing the global statistics.

Generally, this mechanism, called timestep pipelining, allows submitting tasks of stages from the pending jobs before the active one has finished. The only requirement is that the stages of the pending jobs should be independent of the stages of the active job. Therefore, this is another one reason for choosing the stage of aggregating local statistics not to perform in the same stage of updating global statistics. In order to leverage on Spark Streaming optimizations, the batch interval should be less than the averaged time required for completing a job (processing time). However, the aggregated local statistics of pending jobs have been computed with the version of Hoeffding tree during the batch interval of the active job. If a leaf node is split during the active job, then the aggregated local statistics of the pending jobs for this node are useless. As a consequence, all information, included in pending jobs, is lost for the leaf nodes of an internal node, which is split during the active job.

However, each leaf node of a Hoeffding tree requires a large number of training instances in order to split, and thus it can tolerate losing information of a number of training instances.

The forth stage is used for accumulating the best attribute splits of a leaf node, and making a splitting decision for the leaf node based on the Hoeffding bound. It starts after the execution of the *GroupByKey* transformation and it ends when the *ForeachRDD* output operation is executed. Then, all spitting decisions are sent to the driver program which updates the Hoeffding tree. Then, the next job, regarding training instances arrived at the next batch interval, is executed.

Chapter 5

Experimental Study

5.1 Hyper-spectral imaging

Hyper-spectral imaging is a non-destructive detection technology used for remote sensing. It combines spectroscopy with imaging in order to acquire both spatial and spectral information of objects. Hyper-spectral imaging has been used in a broad range of applications (e.g., agriculture, biomedicine, and satellite imaging) over the years. Hyper-spectral images represent observations of a scene at many different wavelengths, beyond the visible electromagnetic spectrum. In figure 5.1 (*source: [29]*), the whole electromagnetic spectrum is illustrated. From spectral and spatial information of these observations, a hyper-spectral data cube can be constructed. A hyper-spectral data cube can also be considered as a stack of images, each of them acquired at a different wavelength. The intensity of each pixel at a different wavelength is measured and thus a complete spectrum of each pixel is available in a data cube. A complete spectrum of a pixel is considered the light emerging from an object as a function of the wavelength. Figure 5.2a (*source: [4]*) represents a hyper-spectral data cube, comprising a set of images acquired at N different wavelengths. The corresponding complete spectrum of two pixel vectors of this cube is presented in figure 5.2b (*source: [4]*) as well. A more detailed description of the theoretical background of hyper-spectral imaging is available in this work [29].

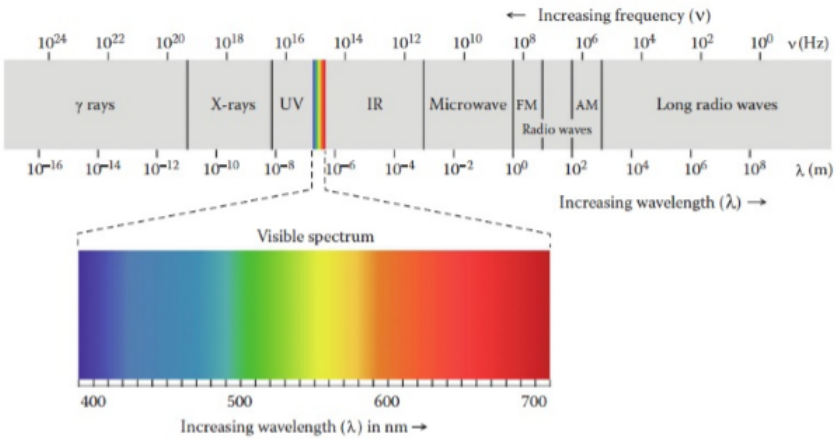
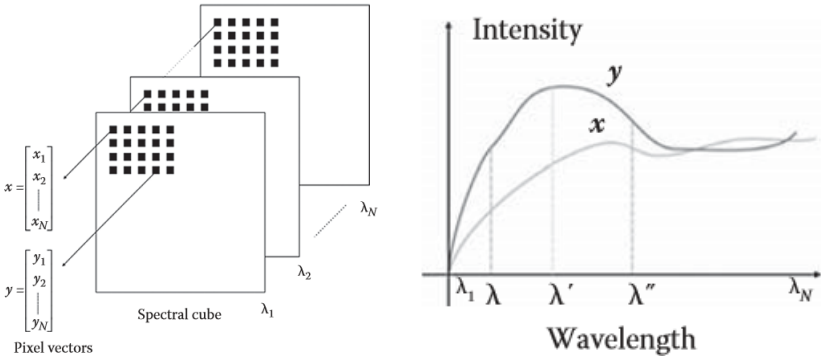


FIGURE 5.1: The electromagnetic spectrum



(A) Two pixel vectors of a hyper-spectral data cube

(B) The complete spectrum of two pixels

FIGURE 5.2: Hyper-spectral data cubes

Various machine learning methods [48, 7, 28] have been applied to hyper-spectral data cubes over the years in order to provide surveillance, reconnaissance and biomedical diagnosis. Decision tree learning algorithms [74] can effectively address classification problems based on hyper-spectral data cubes. The reason is that every substance reacts in a unique manner to light at different wavelengths; because it has a different chemical composition. Therefore, spectral information of each substance is unique and this principle is known as spectral signature of each substance. A hyper-spectral data cube is an instance of the training set. Each possible combination of pixels and wavelength is a numerical attribute taking a value from 0 to 255. This value corresponds to the intensity of a pixel at a specific wavelength, extracted from a grayscale image. In many applications, spatial information is not informative and is discarded.

Despite the predictive accuracy of decision trees, their building phase demands high computational power and large amounts of main memory; since a data cube is high-dimensional. In this regard, Apache Spark can satisfy the requirements of building a decision tree from a large amount of data cubes. Moreover, the proposed incremental decision tree learning algorithm can minimize the memory requirements as only the statistics needed to evaluated the splits are required to be stored. More precisely, a data cube, arriving at the system, is processed only once and then it is discarded from main memory. Thus, it is possible to efficiently build a decision tree from an unbounded stream of data cubes. As far as we concerned, the approach of handling hyper-spectral data cubes as data streams has not been proposed in any work in the literature.

5.2 Data sets

In this thesis, the Hoeffding decision tree is used in order to predict the color of an object based on its spectral signature. Each color has a different chemical composition, and as a consequence a different spectral signature. In order to evaluate our implementation, we use a real-time hyper-spectral data cube, generated from the Electronics Laboratory of the School of Electronic and

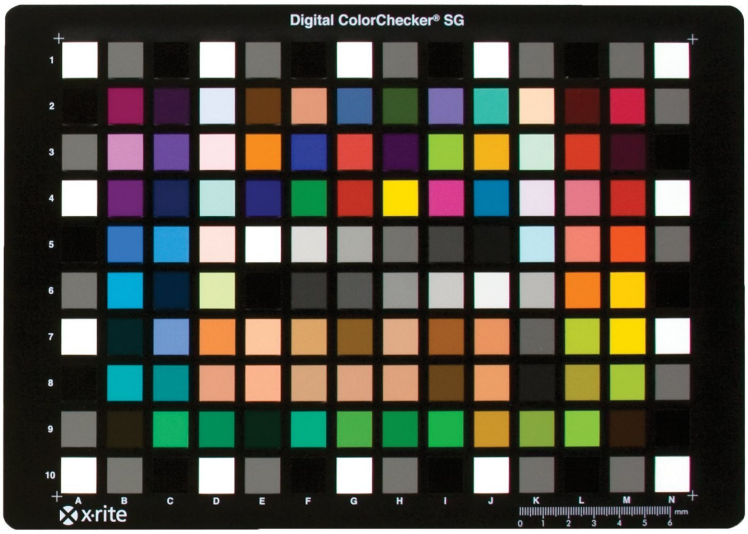


FIGURE 5.3: The X-Rite ColorChecker SG

Computer Engineering (ECE) of the Technical University of Crete (TUC). The data cube contains spectral and spatial information of the X-Rite ColorChecker SG. X-Rite ColorChecker SG, illustrated in figure 5.3, is a unique test pattern, designed to help determine the true color balance or optical density of any color rendition system. The X-Rite ColorChecker SG provides 140 different color and extends the X-Rite ColorChecker, offering 24 colors. The basic principle of our analysis is that each color has a different chemical composition, and as a consequence a different spectral signature. In X-Rite ColorChecker SG, 44 colors are identical and they are removed from the data cube. Therefore, there are 96 classes in our classification problem.

In our problem, spatial information of the hyper-spectral data cube is not useful, as the color of a target object is not related with its position. Our scope is to make color classification based on the spectral signature of each color. The number of classes is 96 since 96 colors are unique in ColorChecker SG. Generally, spectrometers measure the intensity of the light emerging from an object as a function of the wavelength. This information is captured by converting every image to a gray-scale representation. In our case, the intensity of each color (96 unique colors) is measured at 1,600 different positions (pixels) for 60 different bands of wavelength (400nm first band, 10nm band step, 990nm last band). The size of real-time data set, derived by the hyper-spectral data cube, is 153,600 (1600 pixels * 96 colors) instances.

A noise, drew from a zero-mean normal distribution with variance 5, is added to each measured intensity of the real-time data set. When the value (intensity) of an attribute (band) is out of $0 - 255$ range, the real value is used.

The procedure of creating a noisy data set form the real data set is repeated 10 times. Therefore, the noisy data set, used in our experiments, contains 1,536,000 (10 times * 1600 pixels * 96 colors) instances. The data set has 60 numerical attributes (60 different bands). Each attribute can take an integer value from 0 to 255. Finally, each instance has a class label from 96 colors. The data set contains 16,000 instances for each color and is stored in 16,000 batches. Each batch (96 instances) contains only one training instance for each color (96 colors). Moreover, all instances are stored in a specific format in order to be converted into the labeled points data type of Spark MLlib. From this noisy data set, various data sets are generated in order to evaluate our implementation based on the number of attributes and instances.

The procedure of creating the data sets for evaluating our implementation based on attributes is the following. Firstly, only the first 153,600 instances (1,600 batches) of the noisy data set are used. The data set, derived from holding only the first 153,600 instances of the noisy data set, is called the data set of full bands. The data set of full bands contains the intensity of 60 different bands of wavelength (400nm first band, 10nm band step, 990nm last band). The low spectral resolution data set is created by sampling the intensity of 15 bands from the data set of full bands (400nm first band, 40nm band step, 960nm last band). The data set of featureless bands (9_L) is created by sampling the intensity of 9 bands from the data set of full bands. These bands ($400nm, 500nm, 600nm, 700nm, 750nm, 800nm, 850nm, 900nm, 950nm$) are not as informative as the other bands, based on previous experience. The feature-based data set (9_H) is created by sampling from the data set of full bands, the intensity of 9 specific bands ($410nm, 470nm, 520nm, 550nm, 580nm, 610nm, 650nm, 700nm, 850nm$). Finally, a reduced feature-based data set is generated by sampling 7 bands ($410nm, 480nm, 530nm, 580nm, 640nm, 710nm, 900nm$).

The size of all these data sets is identical (153,600 instances). Each data set is stored in 1,600 batches, each containing 96 training instance classified to different classes. The 80 percent of these batches (1,280 batches = 122,880 training instances) is used as the training set and the other 20 percent (320 batches) is used as the test set. However, the number of numerical attributes of each data set is not identical. More precisely, the data set of full bands contains 60 attributes. The low spectral resolution data set has 15 attributes. The featureless and the feature-based data sets have both 9 attributes. However, the bands of the first data set holds less significant spectral information than the latter one. Finally, the reduced feature-based data set contains 7 attributes.

The procedure of creating the data sets for evaluating our implementation based on the size of the training set is the following. The noisy data set, generated by the real-time data set, contains 1,536,000 instances. The noisy data set is stored in 16,000 batches. The first 1,6000, 3,200, 8,000 batches and all batches (16,000) are used in order to create four data sets with different size. The first 1,6000 batches, combining 153,600 instances, are used in order to evaluate the performance of our model for almost 125K training instances. The first 3,200 batches, combining 307,200 instances, are used in order to evaluate the performance of our model for almost 250K training instances. The first 8,000 batches, combining 768,000 instances, are used in order to evaluate the performance of our model for almost 625K training instances. Finally, all batches, combining 15,360,000 instances, are used in order to evaluate the performance of our model for almost 1.25M training instances. The 80 percent of the instances of each data set is used as the training set and the other 20 percent is used as the

test set. More precisely, the training sets contain 1,280 (122,880 $\approx$ 125K training instances), 2,560 (245,760 $\approx$ 250K training instances), 6,400 (614,400 $\approx$ 625K training instances) and 12,800 (1,228,800 $\approx$ 1.25M training instances) batches and the test data sets contain 320, 640, 1,600 and 3,200 batches.

5.3 Performance Evaluation

In order to evaluate the performance of the proposed implementation, we conducted three experiments. All experiments run on the cluster of the Software Technology and Network Applications Laboratory. In our experiments, the Spark applications are launched on YARN cluster manager in client-mode. In client mode, the driver runs in the client process, and the application master is only used for requesting resources from YARN. The driver node has 8 cores and 12 GB of RAM. The number of executors are changes based on the experiment, but the number of cores is 8 and the size of RAM is 8 GB for each executor. The batch interval of the Spark Streaming programs, at which a new RDD is created, is also changed. The accuracy of the Hoeffding tree for different number of attributes is measured. Further, the scalability of the proposed implementation is evaluated by measuring the throughput for training instances increased proportionally to the number of executors. The parameters of Hoeffding trees, used in all experiments, are $\delta = 10^{-3}$, $R = \log_2 96 \approx 6.584$, $\tau = 0.05$, $n_{ties} = 200$. For each numerical attribute, a histogram with 256 bins is built. Finally, global statistics of interior nodes and obsolete leaf nodes are deleted after 20 minutes.

5.3.1 Accuracy

The accuracy of the Hoeffdinig tree for different number of attributes and training instances is compared to a batch learning one. In these experiments, the batch-learning classification tree, implemented in Spark MLlib, is used. Batch-learning classification trees in Spark MLlib, presented in 3.3, allow users to define an evaluation metric a stopping criterion. In our experiments the information gain is used as the evaluation metric. Moreover, the depth of tree is used as the stopping criterion. The maximum depth of tree is denoted 15.

In the first experiment, the accuracy of the Hoeffding tree is measured for different number of attributes. As described previously, five different data sets, each containing 1,600 batches of 96 instances, are used. For each data set, the 80 percent of these batches is used as the training data set and the other 20 percent is used as the test data set. Each instance of the data set of full bands has 60 attributes. The low spectral resolution data set and the reduced feature-based data set have 15 and 7 attributes, respectively. Finally, the featureless data set and the feature-based data set have both 9 attributes, and they are denoted as 9_L and 9_H, respectively. Therefore, the performance of the Hoeffding tree, compared to the batch decision tree, can be evaluated for different number of attributes. The number of executors is 4 for both decision trees. The batch interval in the Spark Streaming program of the Hoeffdinng tree is 20 seconds. The results of our experiment are presented in figure 5.4.

In the second experiment, conducted in order to evaluate our implementation, we compare the accuracy of the batch-learning decision tree and the Hoeffding tree for different number of training instances. In this case, the

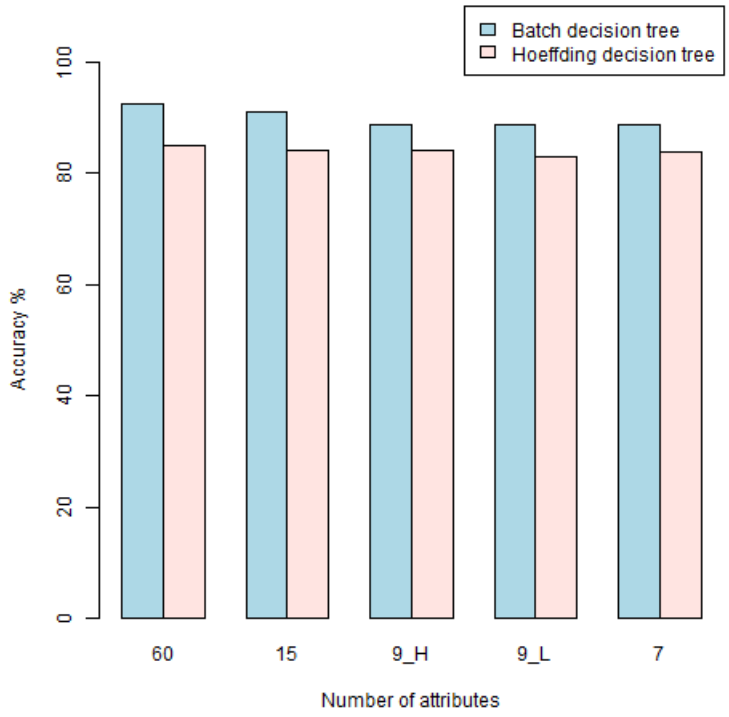


FIGURE 5.4: Comparison between the batch and the Hoeffding decision tree for different number of attributes in terms of accuracy

Model	60	15	9_H	9_L	7
Batch-learning tree	92.44 %	90.84 %	88.65 %	88.55 %	88.63 %
Hoeffding tree	85.02 %	84.02 %	84.16 %	83.01 %	83.88 %

TABLE 5.1: The accuracy of the batch-learning decision tree and the Hoeffding tree for different number of attributes

number of attributes is 60 (full-band) and the number of training instances are 125K, 250K, 625K and 1.25M. The batch intervals in the Spark Streaming programs of the Hoeffdinng tree are 20, 30, 40 and 60 seconds for 125K, 250K, 625K and 1.25M training instances, respectively. The results of our experiment are presented in figure 5.5.

In table 5.1, the percentage of accuracy for different number of attributes is presented. Both decision trees are more accurate as the number of attributes increases in almost all cases. The only exception is the trees, trained with the reduced feature-based data set, which are more effective than the trees, trained with the featureless data set. Moreover, the number of attributes has more impact on the batch learning decision tree than on the Hoeffding tree. In table 5.2, the percentage of accuracy for different number of training instances is presented. Both decision tree models are more accurate as the number of training instances increases. The number of training instances has more impact on the Hoeffding tree than on the batch-learning decision tree.

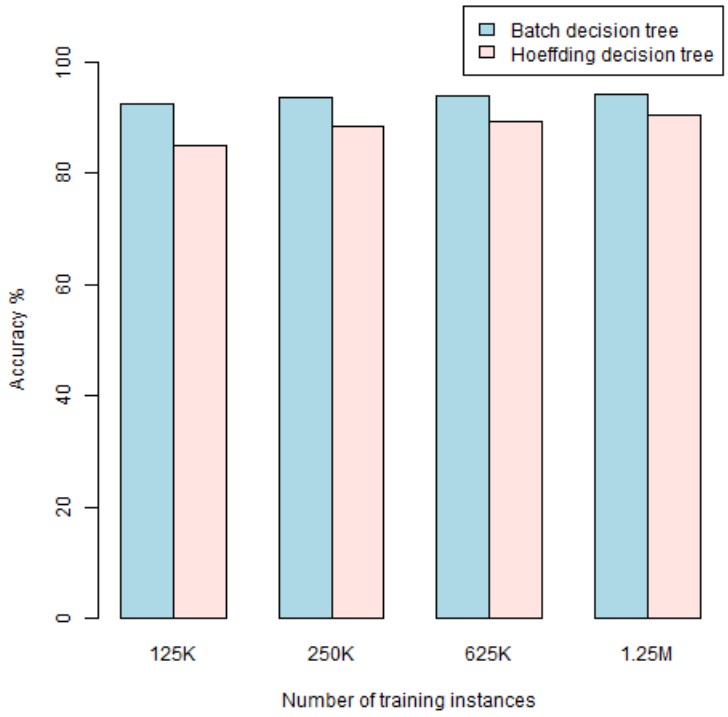


FIGURE 5.5: Comparison between the batch and the Hoeffding decision tree for different number of training instances in terms of accuracy

Model	125K	250K	625K	1.25M
Batch-learning tree	92.44 %	93.62 %	93.81 %	94.18 %
Hoeffding tree	85.02 %	88.42 %	89.28 %	90.24 %

TABLE 5.2: The accuracy of the batch-learning decision tree and the Hoeffding tree for different number of training instances

5.3.2 Scalability

In order to evaluate the scalability in our implementation, we compute the processing time of each training instance for different number of executors. The procedure of this experiment is the following. Firstly, our decision tree model is initialized by the tree-structure of a Hoeffding decision tree, trained with 1,228,800 training instances with 60 attributes, loaded from secondary storage. The global statistics of each node are not loaded and they are initialized when a training instance is classified to this node. Global statistics consist of 60 sparse matrices of 256 rows and 96 columns for each leaf node. In this experiment, as the number of executors increases, the number of training instances increases proportionally. More precisely, the number of executors are 1, 4, 8 and 16 and the number of training instances arrived every second are 10, 40, 80 and 160, respectively. In order to generate identical Hoeffding trees for all different executors, the number of training instances at each batch interval equals to 9,600. This is achieved by denoting the batch interval 16, 8, 4 and 1 minutes for the programs with 1, 4, 8 and 16 executors, respectively. The generated Hoeffding

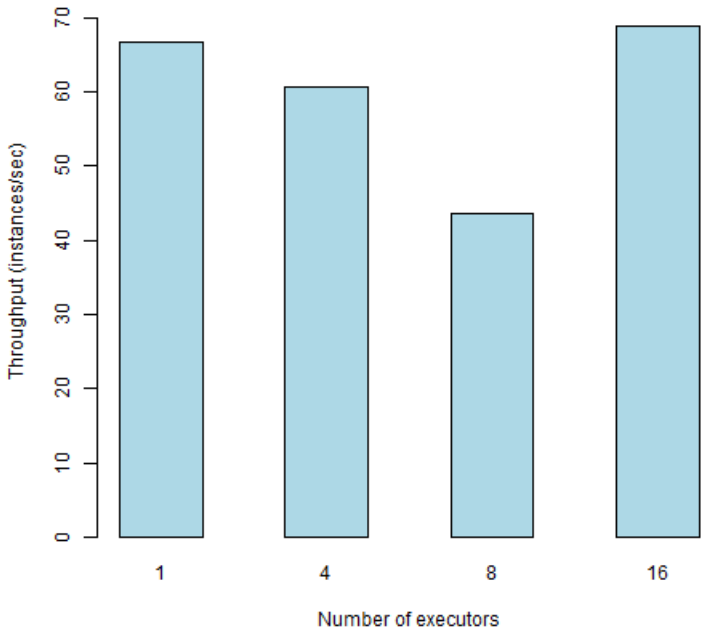


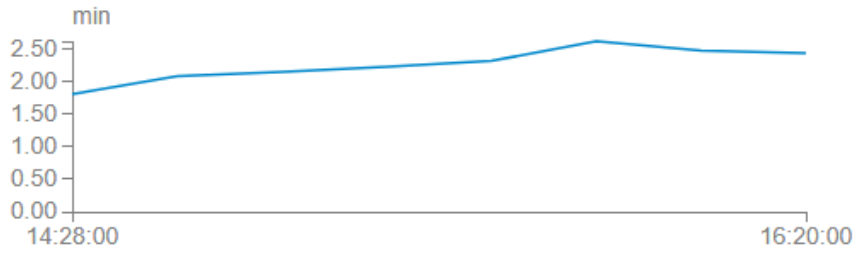
FIGURE 5.6: The throughput of the proposed implementation for training instances increased proportionally to the number of executors

trees are identical as they are updated for each new RDD, created at the batch interval in order to capture all new information. Therefore, Spark Streaming programs with 1, 4, 8 and 16 executors have 16, 4, 2 and 1 minutes batch intervals and last 128, 64, 32, and 8 minutes, respectively. Finally, the number of partitions of each RDD for Spark Streaming programs with 1, 4, 8 and 16 are 8, 32, 64 and 128, respectively.

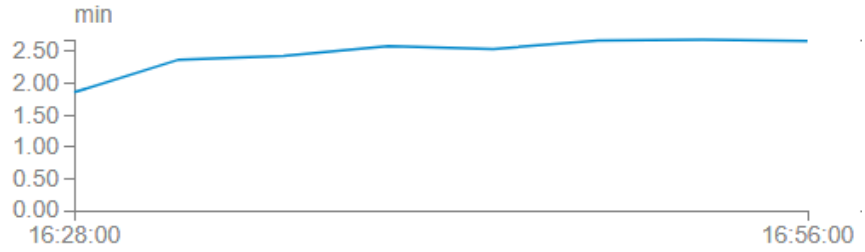
The processing time of 8 Spark batches (RDDs), each containing 9,600 training instances, is measured. The measurements for all different number of executors are illustrated in figure 5.7, extracted from the Spark UI. The averaged processing time of the last five measurements is used in order to compute the average processing time of each Spark batch. Then, the throughput (training instances/second) for each number of executors is measured by dividing the number of instances in each Spark batch (9,600 training instances) with the averaged processing time of the last five Spark batches. The throughput metric is the number of processed training instances per second. In figure 5.6, the throughput of the proposed implementation for different number of executors is presented.

It is observed from figure 5.6 that the throughput of our implementation significantly increases for 16 executors. As the number of executors increases, aggregated local statistics and global statistics are partitioned proportionally. However, a major factor for the improvement in performance of our implementation is that the batch interval is less than the averaged time required for completing a job (processing time). Therefore, our implementation leverages on Spark Streaming optimizations, described in 4.4. More precise, the

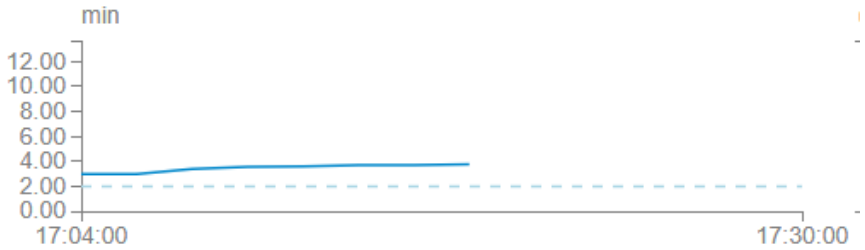
timestep pipelining allows executing stages of pending jobs before the active job is finishes. As a consequence, the aggregated local statistics of pending jobs have already been shuffled to the worker nodes containing global statistics. Therefore, the proposed implementation scales well, capitalizing on the optimizations of Spark Streaming.



(A) 1 executor



(B) 2 executors



(C) 8 executor

(D) 16 executors

FIGURE 5.7: The processing time of each Spark batch for different number of executors

Chapter 6

Conclusion

In this thesis, we propose an implementation of Hoeffding decision trees in the Spark Streaming platform. The implementation performs horizontal data parallelism in the shared-nothing architecture of Spark. The high dimensional global statistics, required to evaluate the splits, are stored as sparse matrices across the cluster. These statistics are instantly updated, when new training instances are available. Furthermore, distributed computations are performed in order to identify the optimal split and assess whether the splitting criterion is satisfied. The generated model is used to predict the color of an object based on its spectral signature. We evaluated our implementation by comparing it with a batch learning classification tree in terms of accuracy for different number of attributes and training instances. The results show that a Hoeffding tree is always less effective than a batch learning decision tree. However, the differences between the accuracy of these models are not significant. Furthermore, as the training instances and the number of attributes increase, we perceive increased level of accuracy for both models. The number of attributes has more impact in terms of accuracy on a batch learning decision tree than on a Hoeffding tree. On the other hand, the size of training data set is more significant in terms of accuracy on a Hoeffding tree than on a batch-learning decision tree. Finally, the proposed implementation scales well, capitalizing on the optimizations of the Spark Streaming platform.

Bibliography

[1] Hussein Almuallim. An efficient algorithm for optimal pruning of decision trees. *Artificial Intelligence*, 83(2):347–362, 1996.

[2] Nuno Amado, Joao Gama, and Fernando Silva. Parallel implementation of decision tree learning algorithms. In *Portuguese Conference on Artificial Intelligence*, pages 6–13. Springer, 2001.

[3] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, et al. Spark sql: relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, pages 1383–1394. ACM, 2015.

[4] Costas Balas. Lecture notes in advanced topics in electronic imaging, 2018. Technical University of Crete.

[5] Yael Ben-Haim and Elad Tom-Tov. A streaming parallel decision tree algorithm. *Journal of Machine Learning Research*, 11(Feb):849–872, 2010.

[6] Gérard Biau and Erwan Scornet. A random forest guided tour. *Test*, 25(2):197–227, 2016.

[7] José M Bioucas-Dias, Antonio Plaza, Nicolas Dobigeon, Mario Parente, Qian Du, Paul Gader, and Jocelyn Chanussot. Hyperspectral unmixing overview: geometrical, statistical, and sparse regression-based approaches. *IEEE journal of selected topics in applied earth observations and remote sensing*, 5(2):354–379, 2012.

[8] Marko Bohanec and Ivan Bratko. Trading accuracy for simplicity in decision trees. *Machine Learning*, 15(3):223–250, 1994.

[9] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[10] Leo Breiman. *Classification and regression trees*. Routledge, 2017.

[11] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[12] Wray Buntine. Learning classification trees. *Statistics and computing*, 2(2):63–73, 1992.

[13] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[14] Thomas G Dietterich. Ensemble methods in machine learning. In *International workshop on multiple classifier systems*, pages 1–15. Springer, 2000.

[15] Pedro Domingos and Geoff Hulten. Mining high-speed data streams. In *Proceedings of the sixth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 71–80. ACM, 2000.

[16] Floriana Esposito, Donato Malerba, Giovanni Semeraro, and J Kay. A comparative analysis of methods for pruning decision trees. *IEEE transactions on pattern analysis and machine intelligence*, 19(5):476–491, 1997.

[17] Yoav Freund. An adaptive version of the boost by majority algorithm. *Machine learning*, 43(3):293–318, 2001.

[18] Yoav Freund. Boosting a weak learning algorithm by majority. *Information and computation*, 121(2):256–285, 1995.

[19] Yoav Freund and Robert E. Schapire. A decision-theoretic generalization of on-line learning and an application to boosting. *Journal of computer and system sciences*, 55(1):119–139, 1997.

[20] Jerome Friedman, Trevor Hastie, Robert Tibshirani, et al. Additive logistic regression: a statistical view of boosting (with discussion and a rejoinder by the authors). *The annals of statistics*, 28(2):337–407, 2000.

[21] Joao Gama. *Knowledge discovery from data streams*. CRC Press, 2010.

[22] Joao Gama, Indre Zliobaite, Albert Bifet, Mykola Pechenizkiy, and Abdelhamid Bouchachia. A survey on concept drift adaptation. *ACM computing surveys (CSUR)*, 46(4):44, 2014.

[23] Joao Gama, Ricardo Rocha, and Pedro Medas. Accurate decision trees for mining high-speed data streams. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 523–528. ACM, 2003.

[24] Joao Gama, Pedro Medas, and Ricardo Rocha. Forest trees for on-line data. In *Proceedings of the 2004 ACM symposium on Applied computing*, pages 632–636. ACM, 2004.

[25] Johannes Gehrke, Venkatesh Ganti, Raghu Ramakrishnan, and Wei-Yin Loh. Boat—optimistic decision tree construction. In *ACM SIGMOD Record*, volume 28 of number 2, pages 169–180. ACM, 1999.

[26] Johannes Gehrke, Raghu Ramakrishnan, and Venkatesh Ganti. Rainforest-a framework for fast decision tree construction of large datasets. In *VLDB*, volume 98, pages 416–427, 1998.

[27] Pierre Geurts, Damien Ernst, and Louis Wehenkel. Extremely randomized trees. *Machine learning*, 63(1):3–42, 2006.

[28] Utsav B Gewali, Sildomar T Monteiro, and Eli Saber. Machine learning based hyperspectral image analysis: a survey. *arXiv preprint arXiv:1802.08701*, 2018.

[29] Ioannis Gkouzionis. *Spectral Cube Reconstruction from Multiplexed Spatial and Spectral Data*. Master's thesis, Technical University of Crete, Chania ,Greece, 2017.

[30] Tin Kam Ho. The random subspace method for constructing decision forests. *IEEE transactions on pattern analysis and machine intelligence*, 20(8):832–844, 1998.

[31] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American statistical association*, 58(301):13–30, 1963.

[32] Geoff Hulten and Pedro Domingos. Vfml–a toolkit for mining high-speed time-changing data streams. *Software toolkit*:51, 2003.

[33] Geoff Hulten, Laurie Spencer, and Pedro Domingos. Mining time-changing data streams. In *Proceedings of the seventh ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 97–106. ACM, 2001.

[34] Laurent Hyafil and Ronald L Rivest. Constructing optimal binary decision trees is np-complete. *Information processing letters*, 5(1):15–17, 1976.

[35] Ruoming Jin and Gagan Agrawal. Communication and memory efficient parallel decision tree construction. In *Proceedings of the 2003 SIAM International Conference on Data Mining*, pages 119–129. SIAM, 2003.

[36] Ruoming Jin and Gagan Agrawal. Efficient decision tree construction on streaming data. In *Proceedings of the ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 571–576. ACM, 2003.

[37] Mahesh V Joshi, George Karypis, and Vipin Kumar. Scalparc: a new scalable and efficient parallel classification algorithm for mining large datasets. In *Parallel processing symposium, 1998. IPPS/SPDP 1998. proceedings of the first merged international... and symposium on parallel and distributed processing 1998*, pages 573–579. IEEE, 1998.

[38] Hyunjoong Kim and Wei-Yin Loh. Classification trees with unbiased multiway splits. *Journal of the American Statistical Association*, 96(454):589–604, 2001.

[39] Richard Brendon Kirkby. *Improving hoeffding trees*. PhD thesis, The University of Waikato, 2007.

[40] Josef Kittler. Combining classifiers: a theoretical framework. *Pattern analysis and Applications*, 1(1):18–27, 1998.

[41] Sotiris B Kotsiantis. Decision trees: a recent overview. *Artificial Intelligence Review*, 39(4):261–283, 2013.

[42] Nicolas Kourtellis, Gianmarco De Francisci Morales, and Albert Bifet. Large-scale learning from data streams with apache samoa. *arXiv preprint arXiv:1805.11477*, 2018.

[43] Nicolas Kourtellis, Gianmarco De Francisci Morales, Albert Bifet, and Arinto Murdopo. Vht: vertical hoeffding tree. In *Big Data (Big Data), 2016 IEEE International Conference on*, pages 915–922. IEEE, 2016.

[44] Bartosz Krawczyk, Leandro L Minku, Joao Gama, Jerzy Stefanowski, and Michał Woźniak. Ensemble learning for data stream analysis: a survey. *Information Fusion*, 37:132–156, 2017.

[45] Tjen-Sien Lim, Wei-Yin Loh, and Yu-Shan Shih. A comparison of prediction accuracy, complexity, and training time of thirty-three old and new classification algorithms. *Machine learning*, 40(3):203–228, 2000.

[46] Wei-Yin Loh and Yu-Shan Shih. Split selection methods for classification trees. *Statistica sinica*:815–840, 1997.

[47] Wei-Yin Loh and Nunta Vanichsetakul. Tree-structured classification via generalized discriminant analysis. *Journal of the American Statistical Association*, 83(403):715–725, 1988.

[48] Dimitris Manolakis and Gary Shaw. Detection algorithms for hyperspectral imaging applications. *IEEE signal processing magazine*, 19(1):29–43, 2002.

[49] Manish Mehta, Rakesh Agrawal, and Jorma Rissanen. Sliq: a fast scalable classifier for data mining. In *International Conference on Extending Database Technology*, pages 18–32. Springer, 1996.

[50] Xiangrui Meng, Joseph Bradley, Burak Yavuz, Evan Sparks, Shivaram Venkataraman, Davies Liu, Jeremy Freeman, DB Tsai, Manish Amde, Sean Owen, et al. Mllib: machine learning in apache spark. *The Journal of Machine Learning Research*, 17(1):1235–1241, 2016.

[51] Gianmarco De Francisci Morales and Albert Bifet. Samoa: scalable advanced massive online analysis. *Journal of Machine Learning Research*, 16(1):149–153, 2015.

[52] James N. Morgan and Robert C Messenger. Thaid, a sequential analysis program for the analysis of nominal scale dependent variables, 1973.

[53] James N. Morgan and John A Sonquist. Problems in the analysis of survey data, and a proposal. *Journal of the American statistical association*, 58(302):415–434, 1963.

[54] Biswanath Panda, Joshua S Herbach, Sugato Basu, and Roberto J Bayardo. Planet: massively parallel learning of tree ensembles with mapreduce. *Proceedings of the VLDB Endowment*, 2(2):1426–1437, 2009.

[55] Bernhard Pfahringer, Geoffrey Holmes, and Richard Kirkby. Handling numeric attributes in hoeffding trees. In *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pages 296–307. Springer, 2008.

[56] Foster Provost and Venkateswarlu Kolluri. A survey of methods for scaling up inductive algorithms. *Data mining and knowledge discovery*, 3(2):131–169, 1999.

[57] J. Ross Quinlan. *C4. 5: programs for machine learning*. Elsevier, 2014.

[58] J. Ross Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[59] J. Ross Quinlan. Simplifying decision trees. *International journal of man-machine studies*, 27(3):221–234, 1987.

[60] J. Ross Quinlan and Ronald L Rivest. Inferring decision trees using the minimum description lenght principle. *Information and computation*, 80(3):227–248, 1989.

[61] Sanjay Ranka and V Singh. Clouds: a decision tree classifier for large datasets. In *Proceedings of the 4th Knowledge Discovery and Data Mining Conference*, volume 2 of number 8, 1998.

[62] Juan José Rodriguez, Ludmila I Kuncheva, and Carlos J Alonso. Rotation forest: a new classifier ensemble method. *IEEE transactions on pattern analysis and machine intelligence*, 28(10):1619–1630, 2006.

[63] S. Rasoul Safavian and David Landgrebe. A survey of decision tree classifier methodology. *IEEE transactions on systems, man, and cybernetics*, 21(3):660–674, 1991.

[64] Robert E. Schapire. The strength of weak learnability. *Machine learning*, 5(2):197–227, 1990.

[65] Jeffrey C. Schlimmer and Douglas Fisher. A case study of incremental concept induction. In *AAAI*, volume 86, pages 496–501, 1986.

[66] John Shafer, Rakesh Agrawal, and Manish Mehta. Sprint: a scalable parallel classi er for data mining. In *Proc. 1996 Int. Conf. Very Large Data Bases*, pages 544–555. Citeseer, 1996.

[67] Claude Elwood Shannon. A mathematical theory of communication. *ACM SIGMOBILE Mobile Computing and Communications Review*, 5(1):3–55, 2001.

[68] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *Mass storage systems and technologies (MSST), 2010 IEEE 26th symposium on*, pages 1–10. Ieee, 2010.

[69] Mahesh K Sreenivas, Khaled Alsabti, and Sanjay Ranka. Parallel out-of-core divide-and-conquer techniques with application to classification trees. In *Parallel Processing, 1999. 13th International and 10th Symposium on Parallel and Distributed Processing, 1999. 1999 IPPS/SPDP. Proceedings*, pages 555–562. IEEE, 1999.

[70] Paul E Utgoff. Incremental induction of decision trees. *Machine learning*, 4(2):161–186, 1989.

[71] Paul E. Utgoff, Neil C. Berkman, and Jeffery A Clouse. Decision tree induction based on efficient tree restructuring. *Machine Learning*, 29(1):5–44, 1997.

[72] Shivaram Venkataraman, Zongheng Yang, Davies Liu, Eric Liang, Hossein Falaki, Xiangrui Meng, Reynold Xin, Ali Ghodsi, Michael Franklin, Ion Stoica, et al. Sparkr: scaling r programs with spark. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1099–1104. ACM, 2016.

[73] Tom White. *Hadoop: The definitive guide*. " O'Reilly Media, Inc.", 2012.

[74] Junshi Xia, Peijun Du, Xiyan He, and Jocelyn Chanussot. Hyperspectral remote sensing image classification based on rotation forest. *IEEE Geoscience and Remote Sensing Letters*, 11(1):239–243, 2014.

[75] Reynold S. Xin, Joseph E. Gonzalez, Michael J. Franklin, and Ion Stoica. Graphx: a resilient distributed graph system on spark. In *First International Workshop on Graph Data Management Experiences and Systems*, page 2. ACM, 2013.

[76] Matei Zaharia, Tathagata Das, Haoyuan Li, Timothy Hunter, Scott Shenker, and Ion Stoica. Discretized streams: fault-tolerant streaming computation at scale. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, pages 423–438. ACM, 2013.

[77] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.

[78] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, and Ion Stoica. Spark: cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[79] Mohammed Javeed Zaki, Ching-Tien Ho, and Rakesh Agrawal. Parallel classification for data mining on shared-memory multiprocessors. In *Data Engineering, 1999. Proceedings., 15th International Conference on*, pages 198–205. IEEE, 1999.

[80] Zhi-Hua Zhou. *Ensemble methods: foundations and algorithms*. CRC press, 2012.

[81] Ruoqing Zhu, Donglin Zeng, and Michael R Kosorok. Reinforcement learning trees. *Journal of the American Statistical Association*, 110(512):1770–1784, 2015.