Distributed Decision Tree Induction in Peer-to-Peer Systems

Kanishka Bhaduri[*], Ran Wolff[†], Chris Giannella[‡] Hillol Kargupta [§*]

[*]Department of Computer Science and Electrical Engineering,

University of Maryland, Baltimore County,

1000 Hilltop Circle,

Baltimore, Maryland, 21250, USA

Email:{kanishk1,hillol}@cs.umbc.edu

[†]Department of Management Information Systems,

Haifa University,

Haifa, 31905, Israel

Email:rwolff@mis.haifa.il

[‡]Department of Computer Science,

Loyola College, Baltimore Maryland, USA

Email:cgiannel@acm.org

[§]The author is also affiliated to Agnik, LLC., Columbia, MD, USA

Abstract

This paper offers a scalable and robust distributed algorithm for decision tree induction in large Peer-to-Peer (P2P) environments. Computing a decision tree in such large distributed systems using standard centralized algorithms can be very communication-expensive and impractical because of the synchronization requirements. The problem becomes even more challenging in the distributed stream monitoring scenario where the decision tree needs to be updated in response to changes in the data distribution. This paper presents an alternate solution that works in a completely asynchronous manner in distributed environments and offers low communication overhead, a necessity for scalability. It also seamlessly handles changes in data and peer failures. The paper presents extensive experimental results to corroborate the theoretical claims.

Index Terms

peer-to-peer, data mining, decision trees

I. Introduction

Decision tree induction [1][2] is a powerful statistical and machine learning technique widely used for data classification, predictive modeling and more. Given a set of learning examples (attribute values and corresponding class labels) at a single location, there exist several well-known methods to build a decision tree such as ID3 [1] and C4.5 [3]. However, there can be several situations in which the data is distributed over a large, dynamic network containing no special server or client nodes. A typical example is a Peer-to-Peer (P2P) network. Performing data mining tasks such as building decision trees is very challenging in a P2P network because of the large number of data sources, the asynchronous nature of the P2P networks, and dynamic nature of the data. A scheme which centralizes the network data is not scalable because any change must be reported to the central peer, since it might very well alter the result.

In this paper we propose a P2P decision tree induction algorithm in which every peer learns and maintains the correct decision tree compared to a centralized scenario. Our algorithm is completely decentralized, asynchronous, and adapts smoothly to changes in the data and the network. The algorithm is efficient in the sense that as long as the decision tree represents the data, the communication overhead is low compared to a broadcast-based algorithm. As a result, the algorithm is highly scalable. When the data distribution changes, the decision tree is updated automatically. Our work is the first of its kind in the sense that it induces decision trees in large P2P systems in a communication-efficient manner without the need for global synchronization and the tree is the same that would have been induced given all the data to all the peers.

The rest of the paper is organized as follows. In Section II, we present several scenarios in which decision tree induction in large P2P networks is important for decision making. In Section III, we discuss related work. In Section IV, we present assumptions about the distributed computing environments we make in this paper and some background material necessary to understand the P2P decision tree algorithm presented in Section V. In Section VI, we describe extensive experiments illustrating the low communication cost incurred by our algorithm. In Section VII, we conclude the paper.

II. Motivation

P2P networks are quickly emerging as large-scale systems for information sharing. Through networks such as Kazaa, e-Mule, BitTorrents, consumers can readily share vast amounts of information. While initial consumer interest in P2P networks was focused on the value of the data, more recent research such as P2P web community formation argues that the consumers will greatly benefit from the knowledge locked in the data [4] [5].

For instance, music recommendations and sharing systems are a thriving industry today [6][7] -- a sure sign of the value consumers have put on this application. However, all existing systems require that users submit their listening habits, either explicitly or implicitly, to centralized processing. Such centralized processing can be problematic because it can result in severe performance bottleneck. Wolff *et al.* [8] have shown that centralized processing may not be a necessity by describing distributed algorithms which compute association rules (and hence, recommendations) in-network in a robust and scalable manner. Later, Gilburd *et al.* [9] showed that it is relatively easy, given an in-network knowledge discovery algorithm, to produce a similar algorithm which preserves the privacy of users in a well defined sense. Moreover Mierswa *et al.* [10] have demonstrated the collaborative use of features for organizing music collections in a P2P setting. Users would prefer not to have the organization of their data change with every slight change in the data of the rest of the users. At the same time, users would also not like the quality of that organization to degrade a lot. As suggested in this paper, monitoring the change and updating the model seems to hit a good balance between being up-to-date and robust.

Another application which offers high value to the consumers is failure determination [11][12]. In failure determination, computer-log data which may have relation to the failure of software and this data is later analyzed in an effort to determine the reason for the failure. Data collection systems are today integral to both the Windows and Linux operating systems. Analysis is performed off-line on a central site and often uses knowledge discovery methods. Still, home users often choose not to cooperate with current data collection systems because they fear for privacy and currently there is no immediate benefit to the user for participating in the system. Collaborative data mining for failure determination can be very useful in such scenarios, and resolve concerns of both privacy and efficiency. However, failure determination models can be quite complex and developing such models require a lot of expert knowledge. The method we describe allows computing a model centrally, by whatever means deemed suitable, and then testing it against user's data without overloading the users or requiring them to submit their data to an untrusted party.

In the next section we present some work related to this area of research.

III. Related Work

Distributed data mining (DDM) deals with the problem of data analysis in environments with distributed data, computing nodes, and users. This area has seen considerable amount of research during the last decade. For an introduction to the area, interested readers are referred to [13] and [14]. P2P data mining has recently emerged as an area of DDM research, specifically focusing on algorithms which are asynchronous, communication-efficient and scalable. Datta *et al.* [15] presents an overview of this topic.

The work described in this paper relates to two main bodies of research: classification algorithms and computation in large distributed systems also referred to as P2P systems.

A. Distributed Classification Algorithms

Classification is one of the classic problems of the data mining and machine learning fields. Researchers have proposed several solutions to this problem — Bayesian models [16], ID3 and C4.5 decision trees [1][3], and SVMs [17] being just a tiny selection. In this section we present several algorithms which have been developed for distributed classification. The solutions proposed by these algorithms differ in three major aspects — (1) how the search domain is represented using an objective function, (2) which algorithm is chosen to optimize the objective function, and (3) how the work is distributed for efficient searching through the entire space. The latter item has two typical modes — in some algorithms the learning examples are only used during the search for a function (e.g., in decision trees and SVMs) while in other they are also used during the classification of new samples (notably, in Bayesian classifiers).

Meta classifiers are another interesting group of classification algorithms. In a meta classification algorithm such as bagging [18] or boosting [19], many classifiers (of any of the previous mentioned kinds) are first built on either samples or partitions of the training data. Then, those "weak" classifiers are combined using a second level algorithm which can be as simple as taking the majority of their outcomes for any new sample.

Some classification algorithms are better suited for a distributed set up. For instance, Stolfo et el. [20] learn a weak classifier on every partition of the data, and then centralize the classifiers and a sample of the data. This can be a lot cheaper than transferring the entire raw data. Then the meta-classifier is deduced centrally from these data. Another suggestion, by Bar-Or *et al.* [21] was to execute ID3 in a hierarchical network by centralizing, for every node of the tree and at each level, only statistics regarding the most promising attributes. These statistics can, as the authors show, provide a proof that the selected attribute is indeed the one having the highest gain — or otherwise trigger the algorithm to request further statistics.

Caragea *et al.* [22] presented a decision tree induction algorithm for both horizontally and vertically distributed data. Noting that the crux of any decision tree algorithm is the use of an effective splitting criteria, the authors propose a method by which this criteria can be evaluated in a distributed fashion. More specifically, the paper shows that by only centralizing the summary statistics from each site to one location, there can be huge savings in terms of communication when compared to brute force centralization. Moreover, the distributed decision tree induced is the same compared to a centralized scenario. Their system is available as part of the INDUS system.

A different approach was taken by Giannella *et al.* [23] and Olsen [24] for inducing decision tree in vertically partitioned data. They used Gini information gain as the impurity measure and showed that Gini between two attributes can be formulated as a dot product between two binary vectors. To reduce the communication cost, the authors evaluated the dot product after projecting the vectors in a random smaller subspace. Instead of sending either the raw data or the large binary vectors, the distributed sites communicate only these projected low-dimensional vectors. The paper shows that using only 20% of the communication cost necessary to centralize the data, they can build trees which are at least 80% accurate compared to the trees produced by centralization.

Distributed probabilistic classification on heterogenous data sites have also been discussed by Merugu and Ghosh [25]. Similarly, Park *et al*. have proposed a fourier spectrum-based approach for decision tree induction in vertically partitioned datasets [26].

A closely related topic is Multivariate Regression (MR) where the output is real-valued instead of categorical. Hershberger *et al*. [27] considered the problem of computing a global MR in a vertically partitioned data distribution scenario. The authors proposed a wavelet transform of the data such that, after the transformation, the effect of the cross terms can be dealt with easily. The local MR models are then transported to the central site and combined to form the global MR model. Several other techniques have been proposed for doing distributed MR using distributed kernel regression such as by Guestrin *et al*. [28] and Predd *et al.* [29].

When the scale of the system grows to millions of network peers — as in most modern P2P systems — the quality of such algorithms degrades. This is because for such large scale systems, no centralization of data, statistics, or models, is practical any longer. By the time such statistics can be gathered, it is reasonable that both the data and the system have changed to the point that the model needs to be calculated again. Thus, classification in P2P networks requires a different breed of algorithms — ones that are fully decentralized, asynchronous and can handle dynamically changing data and network.

B. Data Mining in Large Scale Distributed Systems

Previous work on data mining in P2P networks span three main types of algorithms: best effort heuristics, gossip and flooding based computations, and so called local algorithms. In a typical best effort heuristic [30][5], peers sample data (using some variations of graph random walk as proposed in [31]) from their own partition and that of several neighbors and then build a model assuming that this data is representative of that of the entire set of peers. All these algorithms can be grouped as probabilistic approximate algorithms. On the contrary, deterministic approximate algorithms for large scale networks return the same result every time they are run. Mukherjee *et al.* [32] have developed a communication efficient algorithm for inferencing in sensor networks using a variational approximation technique [33][34]. The paper considers vertically distributed data where each peer learns a probability distribution of the hidden variables given the visible variables. It aims to solve problems such as target tracking, target classification etc. in wireless sensor networks. Another deterministic approximate algorithm is the distributed k-means algorithm developed by Datta *et al*. [35]. It works on horizontally partitioned data whereby each node communicates with its immediate neighbors only.

Flooding algorithms, as their name hints, flood data (or sufficient statistics) through the entire network such that eventually every peer has all the data (or combined statistics) of the entire network. Since flooding is too costly in the common case, actual algorithms usually use gossip — randomized flooding. In gossip, every peer sends its statistics to a random peer. As demonstrated by Kempe *et al*. [36] and Jelasity *et al*. [37] a variety of aggregated statistics can be computed in this way. Gossip algorithms provide probabilistic guarantee for the accuracy of their outcome. However, they can still be quite costly — requiring hundreds of messages per peer for the computation of just one statistic.

Researchers have proposed several robust and efficient algorithms for P2P systems commonly termed as local algorithms in the literature such as association rule mining [8], facility location [38], outlier detection [39], and meta-classification [40] (described more thoroughly below). They are data dependent distributed algorithms. However, in a distributed setup data dependency means that at certain conditions peer can cease to communicate with one another and the algorithm terminates with an *exact* result (equal to that which would be computed given the entire data). These conditions can occur after a peer collects the statistics of just few other peers. In such cases, the overhead of every peer becomes independent of the size of the network (and generally very low). Furthermore, these data dependent conditions can be rechecked every time the data or the system changes. If the change is stationary (i.e., the result of the computation remains the same) then, very often, no communication is necessary. This feature makes local algorithms exceptionally suitable for P2P networks (as well as for wireless sensor networks). Lately, researchers have looked into the description of local algorithm complexity using veracity radius [41] and the description of generic local algorithms which can be implemented for a large family of functions [42].

The work most related to the one described in this paper is the Distributed Plurality Algorithm (DPV) by Ping *et al.* [40]. In that work, a meta classification algorithm is described in which every peer computes a weak classifier on its own data. Then weak classifiers are merged into a meta classifier by computing — per new sample — the majority of the outcomes of the weak classifiers. The computation of weak classifiers requires no communication overhead at all, and the majority is computed using an efficient local algorithm.

Our work is different from DPV in several ways: firstly, we compute an ID3-like decision tree from the entire data (rather than many weak classifiers). Because the entire data is used, smaller sub-populations of the data stand a chance to gather statistical significance and contribute to the model — therefore, we argue that our algorithm can be, in general, more accurate. Secondly, as proposed in DPV, every peer needs to be aware of each new sample in order to classify it. This mode of operation, which is somewhat reminiscent of Bayesian classification, requires broadcasting the new samples to all peers or limits the algorithm to specific cases in which all peers cooperate in classification of new samples (given to all) based on their private past experience. In contrast, in our work, all peers jointly study the same decision tree. Then, when a peer is given a new sample, it can be classified with no communication overhead at all. When learning samples are few and new samples are in abundance, our algorithm can be far more efficient. This can happen, for example, when the data distribution does not change, but the test tuples arrive at a fast rate.

IV. Background

A. Distributed Computation Assumptions

Let S denote a collection of data tuples with class labels that are horizontally distributed over a large (undirected) network of machines (peers) wherein each peer communicates only with its immediate neighbors (one hop neighbors) in the network. The communication network can be thought of as a graph with vertices (peers) V. For any given peer $k \in V$, let N_k denote the immediate neighbors of k. Peer k will only communicate directly with peers in N_k.

Our goal is to develop a distributed algorithm under which each peer computes the decision tree over S (the same tree at each peer).[1] However, the network is dynamic in the sense that the network topology can change i.e. peers may enter or leave at any time or the data held by each peer can change. Hence S, the union of all peers' data, can be time-varying. Our distributed

[1]Throughout this paper, peers refer to the machines in the P2P network and nodes refer to the entries of the tree (in which the attribute is split).

algorithm is designed to seamlessly adapt to network and data changes in a communication-efficient manner.

We assume that communication among neighboring peers is reliable and ordered. Moreover, we assume that when a peer is disconnected or reconnected, all of its neighbors, k, are informed, *i.e.* N_k is known to k and is updated automatically. These assumptions can easily be enforced using standard numbering and retransmission in which messages are numbered, ordered and retransmitted if an acknowledgement does not arrive in time, ordering, and heart-beat mechanisms. Moreover, these assumptions are not uncommon and have been made elsewhere in the distributed algorithms literature [43]. Khilar and Mahapatra [44] discuss the use of heartbeat mechanisms for failure diagnosis in mobile ad-hoc networks.

Furthermore, it is assumed that data sent from P_i to P_j is never sent back to P_j. Since we are dealing with statistics of data (e.g. counts), one way of ensuring this is to make sure that a tree overlay structure is imposed on the original network such that there are no cycles. This allows us to use a relatively simple distributed algorithm as our basic building block: distributed majority voting (more details later). We could get around this assumption in one of two ways. (1) Liss *et al.* [45], have developed an extended version of the distributed majority voting algorithm which does not require the assumption that the network topology forms a tree. We could replace our use of simple majority voting as a basic building block with the extended version developed by Liss *et al.* (2) The underlying tree communication topology could be maintained independently of our algorithm using standard techniques such as [43] (for wired networks) or [46] (for wireless networks).

B. Distributed Majority Voting

Our algorithm utilizes, as a building block, a variation of the distributed algorithm for *majority voting* developed by Wolff and Schuster [8]. Each peer $k \in V$ contains a real number δ^k and the objective is to determine whether $\Delta = \sum_{k \in V} \delta^k \geq 0$.

The following algorithm meets this objective. For peers $k, \ell \in V$, let $\delta^{k\ell}$ denote the most recent message (a real number) peer k sent to ℓ. Similarly, $\delta^{\ell k}$ denotes the latest message received by k from ℓ. Peer k computes $\Delta^k = \delta^k + \sum_{\ell \in N_k} \delta^{\ell k}$, which can be thought of as $k's$ estimate of Δ based on all the information available. Peer k also computes $\Delta^{k\ell} = \delta^{k\ell} + \delta^{\ell k}$, for each neighbor $\ell \in N_k$ which is all the exchanged information between k and ℓ. When an event at peer k occurs, k will decide, for each neighbor ℓ, whether a message need be sent to ℓ. An event at k consists of one of the following three situations: (i) k is initialized (enters the network or otherwise begins computation of the algorithm); (ii) k experiences a data change δ^k or a change of its neighborhood, N_k; (iii) k receives a message from a neighbor ℓ.

The crux of the algorithm is in determining when k must send a message to a neighbor ℓ in response to k detecting an event. More precisely, the question is when can a message be avoided, despite the fact that the local knowledge has changed. Upon detecting an event, peer k would send a message to neighbor ℓ when either of the following two situations occurs: (i) k is initialized; (ii) $\left(\Delta^{k\ell} \geq 0 \wedge \Delta^{k\ell} > \Delta^k\right) \vee \left(\Delta^{k\ell} < 0 \wedge \Delta^{k\ell} < \Delta^k\right)$ evaluates true.

When k detects an event and the conditions above indicate that a message must be sent to neighbor ℓ, k sets $\delta^{k\ell}$ to $\alpha\Delta^k - \delta^{\ell k}$ (thereby making $\Delta^{k\ell} = \alpha\Delta^k$) where α is a fixed parameter between 0 and 1. k then sends $\delta^{k\ell}$ to ℓ. If α were set close to one, then small subsequent variations in Δ^k will trigger more messages from k increasing the communication overhead. On the other hand, if α were set close to zero, the convergence rate of the algorithm could be made unacceptably slow. In all our experiments, we set α to 0.5. This mechanism replicates the one used by Wolff *et al.* in [42]. Observe that the majority voting algorithm is asynchronous, highly decentralized, and adapts smoothly to data and network change.

To avoid a message explosion, we implement a *leaky bucket* mechanism such that the interval between messages sent by a peer does not become arbitrarily small. This mechanism was also used by Wolff *et al.* in [42]. Each peer logs the time when the last message was sent. When a peer decides that a message need to be sent (to any of its neighbors), it does the following. If L time units have passed since the time the last message was sent, it sends the new message right away. Otherwise, it buffers the message and sets a timer to L units after the registered time the last message was sent. Once the timer expires all the buffered messages are sent. For the remainder of the paper, we leave the leaky bucket mechanism implicit in our distributed algorithm descriptions.

V. P2P Decision Tree Induction Algorithm (PeDiT)

This section presents a distributed and asynchronous algorithm *PeDiT* which induces a decision tree over a P2P network in which every peer has a set of learning examples. *PeDiT*, which is inspired by ID3 and C4.5, aims to select at every node — starting from the root — the attribute which will maximize a gain function. Then, *PeDiT* aims to split the node, and the learning examples associated with it, into two new leaf nodes and this process continues recursively. A stopping rule directs *PeDiT* to stop this recursion. In this section a simple depth limitation is used. Other, more complex predicates are described in the appendix.

The main computational task of *PeDiT* is choosing the attribute having the highest gain among all attributes. Similar to other distributed data mining algorithms, *PeDiT* needs to coordinate this decision among the multiple peers. The main exceptions of *PeDiT* are that it underscores the efficiency of decision making and the lack of synchronization. These features make it exceptionally scalable and therefore suitable for networks spanning millions of peers.

PeDiT deviates from the typical decision tree induction algorithms (*e.g.* J48 implemented in Weka) in the use of a simpler gain function — the misclassification error — rather than the more popular (and, arguably, better) information-gain and gini-index functions. Misclassification error offers less distinction between attributes: a split can have the same misclassification error in these two seemingly different cases — (1) the erroneous examples are divided equally between the two leaves it creates or (2) if one of these leaves is 100% accurate. Comparatively, both information-gain and gini-index would prefer the latter case to the former. Still, the misclassification error can yield accurate decision trees (as discussed later) and its relative simplicity makes it far easier to compute in a distributed set-up.

For the sake of clarity, we divide the discussion of the algorithm into two subsections: Section V-A below describes an algorithm for the selection of the attribute offering the lowest misclassification error from amongst a large set of possibilities. Next, Section V-B describes how a collection of such decisions can be efficiently used to induce a decision tree.

A. Splitting attribute choosing using the misclassification gain function

1) Notations: Let S be a set of learning examples — each a vector in $\{0,1\}^d \times \{0,1\}$ — where the first d entries of each example denote the attributes, $A^1, \ldots, A^d$, and the additional one denotes the class $Class$. The cross table of attribute A^i and the class is $X^i = \begin{array}{cc} x^i_{00} & x^i_{01} \\ x^i_{10} & x^i_{11} \end{array}$ where x^i_{01} is the number of examples in the set S for which $A^i = 0$ and $Class = 1$. We also define the indicator variables $s^i_0 = sign\left(x^i_{00} - x^i_{01}\right)$ and $s^i_1 = sign\left(x^i_{10} - x^i_{11}\right)$, where

$$sign\left(x\right) = \begin{cases} 1 & x \geq 0 \\ -1 & x < 0 \end{cases}.$$

We have used the impurity measure based on misclassification error (instead of Gini or Entropy) as described in [47] page 158. The impurity of the $A^i = 0$ tuples is defined to be $1 - \max\{x^i_{00}, x^i_{01}\}/x^i_{0.}$ (where $x^i_{0.}$ is the number of tuples with $A^i = 0$). Similarly, impurity of the $A^i = 1$ tuples is defined to be $1 - \max\{x^i_{10}, x^i_{11}\}/x^i_{1.}$. Choosing the attribute with maximum gain (as described in [47] page 158) is equivalent to minimizing the weighted sum of impurities

$$\text{argmin}\left\{1 \leq i \leq d : \frac{x^i_{0.}}{|S|}\left[1 - \frac{\max\{x^i_{00}, x^i_{01}\}}{x^i_{0.}}\right] + \frac{x^i_{1.}}{|S|}\left[1 - \frac{\max\{x^i_{10}, x^i_{11}\}}{x^i_{1.}}\right]\right\},$$

which can be shown to equal

$$\text{argmax}\left\{1 \leq i \leq d : \left|x^i_{00} - x^i_{01}\right| + \left|x^i_{10}, x^i_{11}\right|\right\}.$$

Thus, if A^{best} is the attribute resulting from the above expression, then for any $A_j \neq A_{best}$,

$$C^{best,j} = \left|x^{best}_{00} - x^{best}_{01}\right| + \left|x^{best}_{10} - x^{best}_{11}\right| - \left|x^j_{00} - x^j_{01}\right| - \left|x^j_{10} - x^j_{11}\right| \geq 0.$$

In a distributed setup, S is partitioned into n sets S_1 through S_n. $X^i_k = \begin{matrix} x^i_{k,00} & x^i_{k,01} \\ x^i_{k,10} & x^i_{k,11} \end{matrix}$ would therefore denote the cross table of attribute A^i and the class in the example set S_k. Note that $x^i_{00} = \sum_{k=1\ldots n} x^i_{k,00}$ and $C^{i,j} = \left|\sum_{k=1\ldots n}\left[x^i_{k,00} - x^i_{k,01}\right]\right| + \left|\sum_{k=1\ldots n}\left[x^i_{k,10} - x^i_{k,11}\right]\right| - \left|\sum_{k=1\ldots n}\left[x^j_{k,00} - x^j_{k,01}\right]\right| - \left|\sum_{k=1\ldots n}\left[x^j_{k,10} - x^j_{k,11}\right]\right|$. Also, notice that $C^{i,j}$ is not, in general, equal to $\sum_{k=1\ldots n}\left|x^i_{k,00} - x^i_{k,01}\right| + \sum_{k=1\ldots n}\left|x^i_{k,10} - x^i_{k,11}\right| - \sum_{k=1\ldots n}\left|x^j_{k,00} - x^j_{k,01}\right| - \sum_{k=1\ldots n}\left|x^j_{k,10} - x^j_{k,11}\right|$. Still, using the indicators s^i_0 and s^i_1 defined above we can write $C^{i,j} = s^i_0 \sum_{k=1\ldots n}\left[x^i_{k,00} - x^i_{k,01}\right] + s^i_1 \sum_{k=1\ldots n}\left[x^i_{k,10} - x^i_{k,11}\right] - s^j_0 \sum_{k=1\ldots n}\left[x^j_{k,00} - x^j_{k,01}\right] - s^j_1 \sum_{k=1\ldots n}\left[x^j_{k,10} - x^j_{k,11}\right]$ which can be rewritten as

$$C^{i,j} = \sum_{k=1\ldots n}\left(s^i_0\left[x^i_{k,00} - x^i_{k,01}\right] + s^i_1\left[x^i_{k,10} - x^i_{k,11}\right] - s^j_0\left[x^j_{k,00} - x^j_{k,01}\right] - s^j_1\left[x^j_{k,10} - x^j_{k,11}\right]\right).$$

This last expression, in turn, is simply a sum — across all peers — of a number $\delta^{i,j}_k = s^i_0\left[x^i_{k,00} - x^i_{k,01}\right] + s^i_1\left[x^i_{k,10} - x^i_{k,11}\right] - s^j_0\left[x^j_{k,00} - x^j_{k,01}\right] - s^j_1\left[x^j_{k,10} - x^j_{k,11}\right]$ which can be computed independently by each peer, assuming it knows the values of the indicators. Finally, denote $\delta_k^{i,j}|abcd$ the value of $\delta_k^{i,j}$ assuming $s_0^i = a$, $s_1^i = b$, $s_0^j = c$, and $s_1^j = d$. Notice that, unlike $\delta_k^{i,j}$, $\delta_k^{i,j}|abcd$ can be computed independently by every peer, regardless of the actual values of s_0^i, s_1^i, s_0^j, and s_1^j.

It is therefore possible to compute A^{best} by concurrently running the following set of majority votes: two per attribute A^i, with inputs $x_{00}^i - x_{01}^i$ and $x_{10}^i - x_{11}^i$, to compute the values of s_0^i and s_1^i; and one for every per of attributes and every possible combination of s_0^i, s_1^i, s_0^j, and s_1^j. Given the results for s_0^i and s_1^i, one could select the right combination and ignore the other. Then, given all of the selected votes, one could find the attribute whose gain is larger than that of any other attribute. Below, we describe an algorithm which performs the same computation far more efficiently.

2) P2P Misclassification Minimization: The P2P Misclassification Minimization P^2*MM* algorithm, Algorithm 1, aims to solve two problems concurrently: it will decide which of the attributes A^1 through A^d is A^{best} and at the same time compute the true value of s_0^i and s_1^i. The general framework of the P^2*MM* algorithm is that of pivoting: a certain A^i is selected as the potential A^{best}. Then the algorithm tries to validate this selection. If A^i turns out to be indeed the best attribute, then this is reported. On the other hand, the revocation of the validation proves that there must exist another attribute (or attributes) A^j which is (are) better than A^{best}. At this point, the algorithm again renames one of these better attributes A^j to be A^{best}, and tries to validate this by comparing it to the other attributes that turned out to be better than the previous selection i.e. previously suspected best. To provide the input to those comparisons, each peer computes $\delta_k^{i,j}|abcd$ relying on the current ad-hoc value of s_0^i, s_1^i, s_0^j, and s_1^j. The ad-hoc values peer k computes for s_0^i and s_1^i will be denoted by $s_{k,0}^i$ and $s_{k,1}^i$, respectively. To make sure those ad hoc results converge to the correct value, two additional majority votes are carried per attribute concurrent to those of the pivoting; in these, the inputs of peer k are $x_{00}^i - x_{01}^i$ and $x_{10}^i - x_{11}^i$, respectively.

The P^2*MM* algorithm works in streaming mode: Every peer k takes two inputs — the set of its neighbors N_k and a set S_k of learning examples. Those inputs may (and often do) change over time and the algorithm responds to every such change by adjusting its output and by possibly sending messages. Similarly, messages stream in to the peer and, can influence both the output and the outgoing messages. The output of peer k is the attribute with the highest misclassification gain. This output, by nature, is ad-hoc and may change in response to any of the events described above.

P^2*MM* is based on a large number of instances of the distributed majority voting algorithm described earlier in Section IV-B. On initialization, P^2*MM* invokes two instances of majority voting per attribute to determine the values of s_0^i and s_1^i — let M_0^i and M_1^i denote these majority votes. For each peer k, its inputs to these votes (instances) are $M_0^i.\delta_k = x_{k,00}^i - x_{k,01}^i$ and $M_1^i.\delta_k = x_{k,10}^i - x_{k,11}^i$. Additionally, for every pair $i < j \in [1 \ldots d]$ P^2*MM* initializes sixteen instances of majority voting — one for each possible combination of values for s_0^i, s_1^i, s_0^j, and s_1^j. Those instances are denoted by $M_{abcd}^{i,j}$ with $abcd$ referring to the combination of values for s_0^i, s_1^i, s_0^j, and s_1^j. For each peer k, its inputs to these instances are $M_{abcd}^{i,j}.\delta_k = a\left[x_{k,00}^i - x_{k,01}^i\right] + b\left[x_{k,10}^i - x_{k,11}^i\right] - c\left[x_{k,00}^j - x_{k,01}^j\right] - d\left[x_{k,10}^j - x_{k,11}^j\right]$. A related algorithm, distributed plurality voting (DPV) [40], could be used to describe our algorithm for selecting one of the sixteen comparisons $M_{abcd}^{i,j}|abcd$ when $abcd$ are static. But since in our case the values of s_0^i, s_1^i, s_0^j, and s_1^j are always changing, it seems difficult to follow the same description as given in [40].

Following initialization, the algorithm is event based. Each peer k is idle unless there is a change in N_k or S_k, or an incoming message changes the Δ_k of one of the instances of majority voting. On such event, the simple solution would be to check the conditions for sending messages in any of the majority votes. However, as shown by [48][40] pivoting can be used to reduce the number of conditions checked from $O\left(d^2\right)$ to an expected $O(d)$. In DPV, the following heuristic is suggested for pivot selection: the pivot would be the vote which has the largest agreement value with any of the neighbors. It is proved that this method is deadlock-free and it is shown to be effective in reducing the number of conditions checked. We implement the same heuristic, choosing the pivot as the attribute A^i which has the largest $M^{j,i}.\Delta_{k,\ell}$ for $j < i$ or the smallest $M^{i,m}.\Delta_{k,\ell}$ for $m > i$ and for any neighbor ℓ. If the test for any $M^{i,j}$ fails, $M^{i,j}.\Delta_{k,\ell}$ needs to be modified by sending a message to ℓ. P^2*MM* does this by sending a message which will set $M^{i,j}.\Delta_{k,\ell}$ to $\alpha M^{i,j}.\Delta_k$ (α is set to $\frac{1}{2}$ by default), which is in line with the findings of [42].

Notice $M_{abcd}^{i,j}.\delta_k = -M_{-a-b-c-d}^{i,j}.\delta_k$ and thus half of the comparisons actually replicate the other half and can be avoided. This optimization is avoided in the pseudocode in order to maintain conciseness. Also notice that while the peer accepts messages in the context of any of the majority votes $M_{abcd}^{i,j}$, it will only respond with a message for the majority vote $M^{i,j}$ — the instance with a, b, c, and d equal to $s_{k,0}^i$, $s_{k,1}^i$, $s_{k,0}^j$, and $s_{k,1}^j$, respectively. The rest of the instances are, in effect, ‘suspended’ and cause no communication overhead.

Algorithm 1 P2P Misclassification Minimization (P^2*MM*)

```
Input variables of peer $k$: the set of neighbors — $N_k$, the set of examples — $S_k$
Output variables of peer $k$: the attribute $A^{pivot}$
Initialization:
  • For every $A^i$ in $A^1 \dots A^d$ initialize two instances of LSD-Majority with inputs $x^i_{k,00} - x^i_{k,01}$
    and $x^i_{k,10} - x^i_{k,11}$. Denote these instances by $M^i_0$ and $M^i_1$ respectively and let $M^i_0.\Delta_k$ and
    $M^i_1.\Delta_k$ denote the knowledge of those two instances. Further, for every $\ell \in N_k$, let $M^i_0.\Delta_{k,\ell}$
    and $M^i_1\Delta_{k,\ell}$ be their agreement.
  • For every $a, b, c, d \in \{-1, 1\}$ and every $A^i, A^j \in [A^1 \dots A^d]$ initialize an instance of LSD-
    Majority with input $\delta^{i,j}_k|abcd$. Denote these instances by $M^{i,j}_{abcd}$. Let $M^{i,j}_{abcd}.\Delta_k$ and $M^{i,j}_{abcd}\Delta_{k,\ell}$
    ($\forall \ell \in N_k$) be the knowledge and agreement of the $M^{i,j}$ instance, respectively. Specifically
    denote $M^{i,j}.\Delta_k$ and $M^{i,j}\Delta_{k,\ell}$ the instance with $a$, $b$, $c$, and $d$ equal to $s^i_{k,0}$, $s^i_{k,1}$, $s^j_{k,0}$, and
    $s^j_{k,1}$, respectively.
On any event:
  • For $A^i \in \{A^1 \dots A^d\}$ and every $\ell \in N_k$
    – If not $M^i_0.\Delta_k \le M^i_0.\Delta_{k,\ell} < 0$ and not $M^i_0.\Delta_k \ge M^i_0.\Delta_{k,\ell} \ge 0$ call $Send(M^i_0, \ell)$
    – If not $M^i_1.\Delta_k \le M^i_1.\Delta_{k,\ell} < 0$ and not $M^i_1.\Delta_k \ge M^i_1.\Delta_{k,\ell} \ge 0$ call $Send(M^i_1, \ell)$
  • Do
    – Let $pivot = \arg\max_{i \in [1 \dots d]} \left\{ \max_{\ell \in N_k, j<i, m>i} \left\{ M^{j,i}.\Delta_{k,\ell}, -M^{i,m}.\Delta_{k,\ell} \right\} \right\}$
    – For $A^i \in \{A^1 \dots A^{pivot-1}\}$ and every $\ell \in N_k$
      * If not $M^{i,pivot}.\Delta_k \le M^{i,pivot}.\Delta_{k,\ell} < 0$ and not $M^{i,pivot}.\Delta_k \ge M^{i,pivot}.\Delta_{k,\ell} \ge 0$ call
        $Send(M^{i,pivot}, \ell)$
    – For $A^i \in \{A^{pivot+1} \dots A^d\}$ and every $\ell \in N_k$
      * If not $M^{pivot,i}.\Delta_k \le M^{pivot,i}.\Delta_{k,\ell} < 0$ and not $M^{pivot,i}.\Delta_k \ge M^{pivot,i}.\Delta_{k,\ell} \ge 0$ call
        $Send(M^{pivot,i}, \ell)$
  • While $pivot$ changes
On message $(id, \delta)$ from $\ell$:
  • Let $M$ be a majority voting instance with $M.id = id$
  • Set $M.\delta_{\ell,k}$ to $\delta$
Procedure Send$(M, \ell)$:
  • $M.\delta_{k,\ell} = \alpha M.\Delta_k + M.\delta_{\ell,k}$
  • Send to $\ell$ $(M.id, M.\delta_{k,\ell})$
```

a) Proof Sketch: To see why P^2*MM* convergence is guaranteed, first notice that eventual correctness of each of the majority votes M^i_0 and M^i_1 is guaranteed because the condition for sending messages is checked for every one of them each time the data changes or a message is received. Next, consider two neighbor peers who choose different pivots, peer k which selects

i and peer ℓ which selects j. Since both k and ℓ will check the condition of $M^{i,j}$, and since $M^{i,j}.\Delta_{k,\ell} = M^{i,j}.\Delta_{\ell,k}$ at least one of them will have to change its pivot. Thus, peers will continue to change their pivots until they all agree on the same pivot. To see that peers will converge to the decision that the pivot is the attribute with the maximal gain (denote it A^m), assume they converge on another pivot. Since the condition of votes comparing the A^{pivot} to any other attribute is checked whenever the data changes, it is guarantee that if $m < pivot$ then $M^{m,pivot}.\Delta_{k,\ell}$ will eventually be larger than zero for all k and $\ell \in N_k$ and if $pivot < m$ then $M^{m,pivot}.\Delta_{k,\ell} > 0$ for all k and $\ell \in N_k$. Thus, the algorithm will replace the pivot with either m or another attribute, but would not be able to converge on the current pivot. This completes the proof of the correct convergence of the P^2*MM* algorithm. ∎

b) Complexity: The P^2*MM* algorithm compares the attributes in an asynchronous fashion and outputs the best attribute. Consider the case of comparing only two attributes. The worst case communication complexity of the P^2*MM* algorithm is O(*size of the network*). This can happen when the misclassification gains of the two attributes are very close. Since our algorithm is eventually correct, the data will need to be propagated through the entire network i.e. all the peers will need to communicate to find the correct answer. Thus the overall communication complexity of the P^2*MM* algorithm, in the worst case, is O(*size of the network*). Now if the misclassification gains of the two attributes are not very close (which is often the case for most datasets), the algorithm is not global; rather the P^2*MM* algorithm can prune many messages as shown by our extensive experimental results. Finally formulating the complexity of such data dependent algorithms in terms of the complexity of the data is a big open research issue even for simple primitives such as majority voting protocol [8], let alone the complex P^2*MM* algorithm presented in this paper.

Figure 1 shows the pivot selection process for three attributes A_h, A_i and A_j for peer P_k having two neighbors P_ℓ and P_m. Snapshot 1 shows the knowledge (Δ_k) and agreements of P_k ($\Delta_{k,\ell}$ and $\Delta_{k,m}$) for the three attributes. Since pivot is the highest agreement, A_h is selected as the pivot. Now there is a disagreement between Δ_k and $\Delta_{k,\ell}$ for A_h. This results in a message and subsequent reevaluation of $\Delta_{k,\ell}$ (to be set equal to Δ_k). In the next snapshot, A_i is selected as the pivot and since $\Delta_{k,\ell} < \Delta_k$, no message needs to be sent.

Comment: The P^2*MM* algorithm above utilizes the assumption that the data at all peers in boolean (attributes and class labels). The boolean attributes assumption can be relaxed to

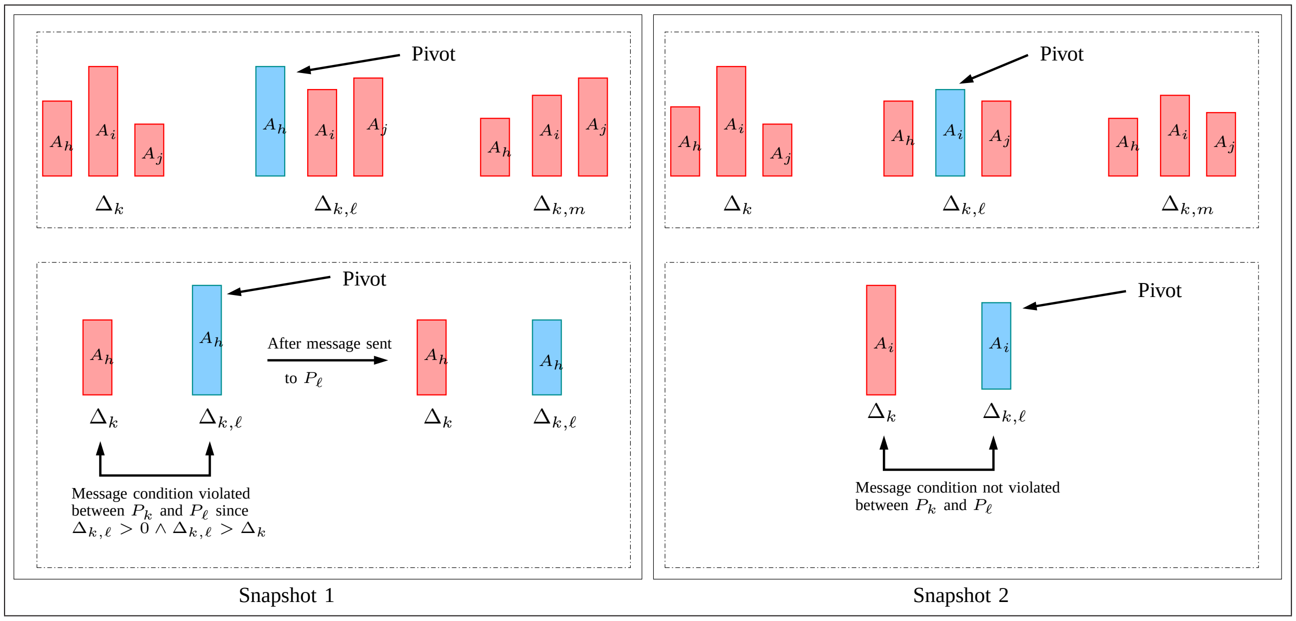

Pivot
A_h A_i A_j
Δ_k
$\Delta_{k,\ell}$
$\Delta_{k,m}$
After message sent to P_ℓ
Message condition violated between P_k and P_ℓ since $\Delta_{k,\ell} > 0 \wedge \Delta_{k,\ell} > \Delta_k$
Message condition not violated between P_k and P_ℓ
Snapshot 1
Snapshot 2

Fig. 1. The pivot selection process and how the best attribute is selected. The pivot is shown at each step.

arbitrary categorial data at the expense of increased majority voting instances per attribute pair (the number of instances increases exponentially with the number of distinct attribute values). Another approach to relaxing the boolean attributes assumption could be to treat each attribute distinct value as its own boolean attribute. As a result, each categorical attribute with v distinct values is treated as v boolean attributes. Here, the number of majority voting instances per pair of attributes increases only linearly with the number of distinct attribute values. However, the issue of deciding which attribute has the highest misclassification gain on the basis of the associated boolean attributes is not entirely clear and is the subject of future work.

B. *Speculative decision tree induction*

P^2MM can be used to decide which attribute would best divide a given set of learning examples. It is well known that decision trees can be induced by recursively and greedily dividing a set of learning examples — starting from the root and proceeding onwards with every node of the tree (e.g., ID3 and C4.5 algorithms [1][3]). In a P2P set-up the progression of the algorithm needs to be coordinated among all peers, or they might end up developing different trees. In smaller scale distributed systems, occasional synchronization usually addresses coordination. However, since a P2P system is too large to synchronize, we prefer speculation [49].

The starting point of tree development — the root — is known to all peers. Thus, they can all initialize P^2*MM* to find out which attribute best splits the example set of the root. However, the peers are only guaranteed to converge to the same attribute selection eventually and may well choose different attributes intermediately. Several questions thus arise: How and when should the algorithm commit resources to a specific split of a node, what should be done if such split appears to be wrong after resources were committed and what should be done about incoherence between neighboring peers?

The P2P Decision Tree Induction ($PeDiT$, see Alg. 2) algorithm has two main functionalities. Firstly, it manages the ad hoc solution which is a decision tree composed of *active* nodes. The root is always active and so is any node whose parent is active provided that the node corresponds with one of the values of the attribute which best splits its parent's examples — i.e., the ad hoc solution of P^2*MM* as computed by the parent. The rest of the nodes are *inactive*. A node (or a whole subtree) can become inactive because its parent (or fore-parent) have changed its preference for splitting attribute. Inactive nodes are not discarded; a peer may well accept messages intended for inactive nodes — either because a neighbor considers them active or because the message was delayed by the network. Such a message would still update the majority voting to which it is intended. However, peers never send messages resulting from an inactive node. Instead, they check, whenever a node becomes active, whether there are pending messages (i.e., majority votes whose test require sending messages) and if so they send those messages.

Another activity which occurs in active nodes is further development of the tree. Each time a leaf it is generated it is inserted into a queue. Once every τ time units, the peer takes the first active leaf in the queue and develops it according to the ad hoc result of P^2*MM* for that leaf. Inactive leaves which precede this leaf in the queue are re-inserted at the end of the queue.

Last, it may happen that a peer receives a message in the context of a node it had note yet developed. Such messages are stored in the *out-of-context* queue. Whenever a new leaf is created, the out-of-context queue is searched and messages pertaining to the new leaf are processed.

A high level overview of the speculative decision tree induction process is shown in Figure 2. Filled rectangles represent newly created nodes. The first snapshot shows the creation of the root with A_1 as the best attribute. The root is split in the next snapshot, followed by further development of the left path. The fourth snapshot shows how the root is changed to a new attribute A_2 and the entire tree rooted at A_1 is made inactive (yellow part). Finally as time

Algorithm 2 P2P Decision Tree Induction $PeDiT$

```
Input: $S$ — a set of learning examples, $\tau$ — mitigation delay
Initialization:
  Create a root leaf and let $root.S \leftarrow S$. Set $nodes \leftarrow \{root\}$. Push $root$ to $queue$
  Send BRANCH message to self with delay $\tau$
On BRANCH message:
  Send BRANCH message to self with delay $\tau$
  For ($i \leftarrow 0$, $\ell \leftarrow null$; $i < queue.length$ and not active($\ell$); $i$++)
    Pop head of queue into $\ell$
    If not active($\ell$)
      enqueue $\ell$
    If active($\ell$)
    Let $A^j$ be the ad-hoc solution of $P^2$MM for $\ell$
    call Branch($\ell, j$)
On data message $\langle n, data \rangle$:
  If $n \notin nodes$
    store $\langle n, data \rangle$ in $out - of - context$
  Else
    Transfer the $data$ to the $P^2$MM instance of $n$
  If active($n$) then
    Process($n$)
Procedure Active($n$):
  If $n = null$ or $n = root$
    return true
  Let $A^j$ be the ad-hoc solution for $P^2$MM for $n.parent$
  If $n \notin n.parent.sons\,[j]$
    return false
  Return Active($n.parent$)
Procedure Process($n$):
  Perform tests required by $P^2$MM for $n$ and send any resulting messages
  Let $A^j$ be the ad-hoc solution for $P^2$MM for $n$
  If $n.sons\,[j]$ is not empty
    for each $m \in n.sons\,[j]$
      call Process($m$)
  Else
    push $n$ to the tail of the queue
Procedure Branch($\ell, , j$):
  Create two new leaves $\ell_0$ and $\ell_1$
  Set $\ell_0.parent \leftarrow \ell$, $\ell_1.parent \leftarrow \ell$
  Set $\ell_0.S \leftarrow \{s \in \ell.S : s\,[j] = 0\}$ and $\ell_1.S \leftarrow \{s \in \ell.S : s\,[j] = 1\}$
  Remove from $out - of - context$ messages intended for $\ell_0$ and $\ell_1$ and deliver the data to the
respective instance of $P^2$MM
  Set $\ell.sons\,[j] = \{\ell_0, \ell_1\}$, add $\ell_0$, $\ell_1$ to nodes and push $\ell_0$ and $\ell_1$ to the tail of the queue
```

progresses, the tree rooted at A_2 is further developed. If it so happens that A_1 now becomes better than A_2, the old inactive tree will now become active and the tree rooted at A_2 will become inactive.

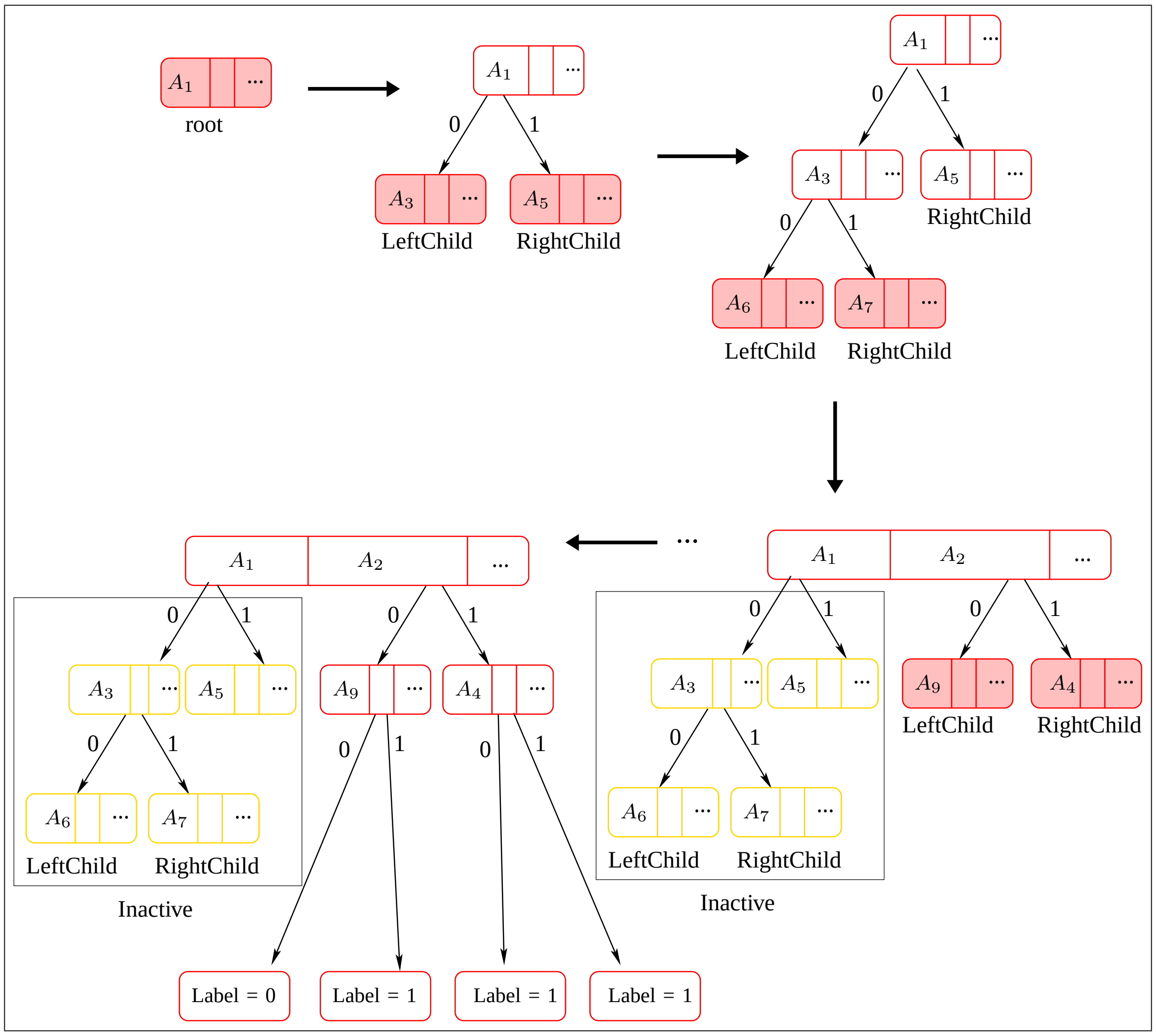
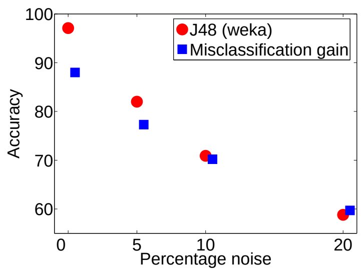


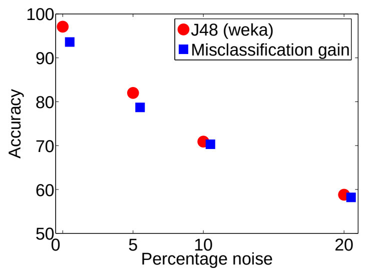
Fig. 2. Figure showing how the speculative decision tree is build by a peer. Filled rectangles represent the newly created nodes. In the first snapshot the root is just created with A_1 as the current best attribute. The root is split into two children in the second snapshot. The third snapshot shows further development of tree by splitting the left child. In the fourth snapshot, the peer gets convinced that A_2 is the best attribute corresponding to the root. Earlier tree is made inactive and a new tree is developed with split at A_2. Fifth snapshot shows the leaf label assignments.

In the next section we present a comparative study of the accuracy of a decision tree algorithm using misclassification gain as the impurity measure with fixed depth stopping and a standard entropy-based pruned decision tree J48 implemented in weka [50].

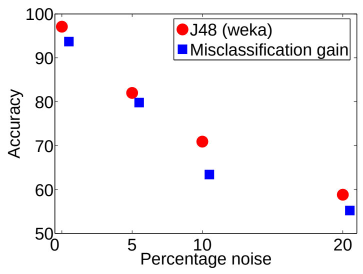
C. Accuracy on Centralized Dataset



(a) Depth of tree = 3.



(b) Depth of tree = 5.



(c) Depth of tree = 7.

Fig. 3. Comparison of accuracy (using 10-fold cross validation) of J48 weka tree and a tree induced using misclassification gain with fixed depth stopping rule on a centralized dataset. The three graphs correspond to depths of 3, 5 and 7 of the misclassification gain decision tree.

The proposed distributed decision tree algorithm *PeDiT* deviates from the standard decision tree induction algorithm in two ways: (1) instead of using entropy or gini-index as the splitting criteria, we have used misclassification gain, and (2) as a stopping rule, we have limited the depth of our trees (which effects the communication complexity of our distributed algorithm as we show later).

In this section we report the results of the comparison of a decision tree induced using misclassification gain as the impurity measure and fixed depth stopping rule to that of an off-the-shelf entropy-based pruned decision tree J48 implemented in Weka [50] on a centralized dataset. There are 500 tuples and 10 attributes in the centralized dataset generated using the scheme discussed in Section VI-B. We have varied the noise in the data ranging from 0% to 20%. For both the trees, the accuracy is measured using 10-fold cross validation. Figure 3 presents the comparative study. The circles correspond to the accuracy of J48 and the squares correspond to the accuracy of the misclassification gain decision tree. These set of results point out some important facts:

1) In most cases the misclassification gain decision tree results in a loss of accuracy over J48. This is not unexpected due to the restrictive nature of the decision tree induction algorithm employed. But the accuracy loss is modest. Moreover, due to the heavy communication and synchronization cost of centralizing and applying J48, this modest loss of accuracy seems quite reasonable.

2) The decrease in accuracy of the misclassification gain decision tree when going to depth 7 at high noise levels is likely due to over-fitting; since for a depth of 3 the average number of tuples per leaf is 56 compared to only 3 tuples per leaf for depth of 7.
3) The $PeDiT$ algorithm is guaranteed to converge to the misclassification gain decision tree with fixed stopping depth. Therefore, once convergence is reached, $PeDiT$ might loose some accuracy with respect to J48 and this is quantified by the accuracy of the misclassification gain decision tree with fixed stopping depth as shown here.

Since limiting the depth affects the communication complexity of our $PeDiT$ algorithm, we will use depths of 3, as it produces quite accurate trees for all noise levels.

VI. Experiments

To validate the performance of our decision tree algorithm, we conducted experiments on a simulated network of peers. In this section we discuss the experimental setup, measurement metric and the performance of the algorithm.

A. Experimental Setup

Our implementation makes use of the Distributed Data Mining Toolkit (DDMT) [51] which is a JADE-LEAP based event simulator developed by the DIADIC research lab at UMBC. The topology is generated using BRITE [52] — an open software for generating network topologies. We have used the *Barabasi Albert (BA)* model in BRITE since it is often considered a reasonable model for the Internet. We use the edge delays defined in BA as the basis for our time measurement[2]. On top of each network generated by BRITE, we overlayed a communication tree.

B. Data Generation

The input data of a peer is generated using the scheme proposed by Domingos and Hulten [53]. Each input data point is a vector in $\{0,1\}^d \times \{0,1\}$. The data generator is a random tree built as follows. At each level of the tree, an attribute is selected randomly and made an internal node of the tree with the only restriction that attributes are not repeated along the same path.

[2]Wall time is meaningless when simulating thousands of computers on a single PC.

After the tree is built up to a depth of 3, a node is randomly made a leaf with a probability of p along with a randomly generated label. We limit the depth of the tree to maximum 6, and make all the non-leaf nodes a leaf (with random labels) after it exceeds that depth. Whenever a peer needs an additional point, it generates a random vector in the d-dimensional space and then passes it through the tree. The label it gets assigned to forms the class label for that input vector. This forms noise-free input vectors. In order to add noise, the bits of the vectors (including the class label) are flipped with a certain probability. Therefore, $n\%$ noise means that each bit of the input vector is flipped with $n\%$ chance and the new value of that bit is chosen uniformly from all the possibilities (including the original value). The data generator is changed every 5×10^5 simulator ticks, thereby creating an epoch. We assume that over lengthy periods of time the data changes are only with respect to a stationary distribution. Therefore for a long time we generate new data points following same distribution. When the above number of simulator ticks elapse, the data distribution is changed. Hence our data is piecewise stationary. Other models of data change exist such as picking random points in time and changing the data or use a more complicated arrival model as done in queueing theory. We have not done it in this paper and is left as part of the future work.

A typical experiment consists of 10 equal length epochs In addition, throughout the experiment we change 10% data of each peer after every 1000 clock ticks. Therefore, in all our experiments there are two levels of data change — (1) stationary change when we sample from the same data distribution every 1000 simulator ticks, and (2) dynamic change when the data distribution changes after every 5×10^5 simulator ticks. Our analysis shows that changing the data more than once in the leaky bucket period does not affect the results, but puts much stress on the simulator.

C. Measurement Metric

The two measurements of our algorithm are the *quality* of the result and the *cost* incurred.

Given a test dataset to each peer, generated from the same distribution as the local dataset, quality is measured in terms of the percentage of correctly classified tuples of this test set. We report two quantities — (1) *stationary accuracy* which refers to the accuracy measured during the last 80% of the epochs and hence correspond to stationary changes in the data when the distribution becomes stable and (2) *overall accuracy* which is measured for the entire length of the experiment and hence corresponds to both the stationary and non-stationary changes. The *stationary accuracy* reflects the accuracy achieved by our algorithm when the distribution does not change, but new samples are drawn from the same distribution while the *overall accuracy* reflects the accuracy when both the distribution and the data changes. Moreover, for each quality graph in Figures 4, 5, 6, 7, 8 and 9 we report two quantities — (1) the average quality over all peers, all epochs and 10 independent trials (the center markers) and (2) the standard deviation over 10 independent trials (error bars).

For measuring the cost of the algorithm we report two quantities — *normalized messages* sent and *normalized bytes* transferred. Our measurement metric for the normalized messages is the number of messages sent by each peer per unit of leaky bucket L. For an algorithm whose communication model is broadcast, its normalized messages is 2, considering 2 neighbors on an average per peer. We report both the overall messages and the monitoring messages; the latter refers to the "wasted effort" of the algorithm. For a given time instance, if a peer needs to send k separate messages corresponding to different majority votes to one particular peer, it is counted as one message to that neighbor.

Similarly, to understand the actual communication overhead of our algorithm in terms of the number of bytes sent, we report both the *overall* and *monitoring* bytes transferred, per unit of L. In every raw message the distributed algorithm sends 5 numbers — the data of the vote, the id of the vote, the id of the attributes which this vote corresponds to, the path of the tree and the maximum pivot. Considering the above example, the number of bytes sent is $5 * k$ per neighbor. As before, for a broadcast based algorithm, having an attribute cross-table with four entries (2 values and 2 classes), its bytes sent would be *no of attributes* $\times 4 \times 2$. The factor of two is assuming 2 neighbors per peer. For example, with 10 attributes, the number of bytes sent per L is 80. Similar to what we did for quality, we have plotted both the average cost and the standard deviation of the result over 10 independent trials.

There are three parameters of the *PeDiT* algorithm that we have explored — (1) the number of local tuples or the size of the local dataset $|S_i|$, (2) the depth of the induced tree, and (3) the size of the leaky bucket L. The measurement points for the local data points per peer are 250, 500, 1000, 2000 and 4000. For the depth of tree, we used values of 2, 3, 4, 5 and 7 while we varied L among 500, 1000, 2000, 3000 and 4000. The values of L are in simulator ticks where the average edge delay is about 1100 time units.

The data generator had two parameters — (1) noise in the data varied between 0%, 5%, 10% and 20%, and (2) number of attributes (10, 15, and 20).

Finally, as a system parameter we varied the number of peers from 50 to 1500.

Unless otherwise stated, we have used the following default values: $|S_i| = 500$, depth of the tree = 3, noise = 10%, number of attributes = 10, number of peers = 1000, and $L = 1000$ (where the average edge delay is about 1100 time units). Under these values, for a broadcast algorithm, the number of normalized messages is 2 while the number of normalized bytes is $10 \times 4 \times 2 = 80$.

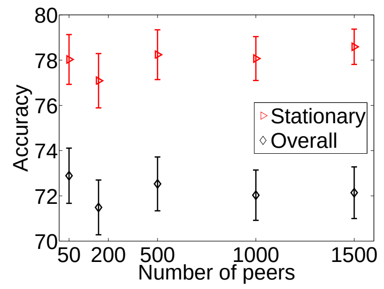
In all our experiments we have observed the following phenomenon. As soon as the epoch changes, the accuracy of the algorithm goes down and the communication increases since the algorithm adapts to the new distribution. As soon as the distribution becomes stable, the message overhead reduces and the accuracy improves. To take note of this, we have plotted both the overall and stationary behavior of the algorithm with respect to the different parameters.

In the next section we first present the accuracy of the *PeDiT* compared to a standard decision tree algorithm. Following, we present the performance of the *PeDiT* algorithm on the different parameters.

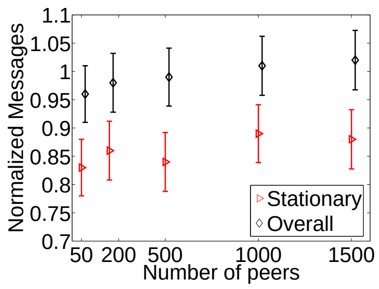
D. Performance Evaluation

1) Scalability: Our first set of results demonstrate the scalability of the *PeDiT* algorithm as the number of peers is varied from 50 to 1500. The number of peers has little effect on the performance as we see in Figure 4. In Figure 4(a), both the overall and stationary accuracy converges to a constant as the number of peers is increased. Normalized messages and normalized bytes transferred, as shown in Figures 4(b) and 4(c), changes very slowly and almost remains a constant as the number of peers is increased. Since our algorithm relies on some data dependent rules to prune messages, the total number of peers has little effect on the quality or the cost. Hence the algorithm is highly scalable. Note that for an algorithm which broadcasts sufficient statistics to maintain the trees the normalized messages and normalized bytes transferred would be 2 and 80 respectively. Our results show a significant improvement.

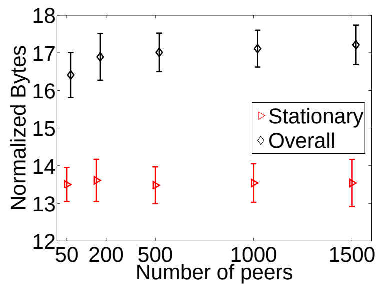
2) Data Tuples per Peer: In this section we have experimented with the first algorithm parameter — the number of tuples in the local dataset $|S_i|$. Figure 5 summarizes the results. Stationary accuracy increases from 72% to 85%, stationary messages decrease from 1.21 to 0.32



(a) Quality vs. number of peers.



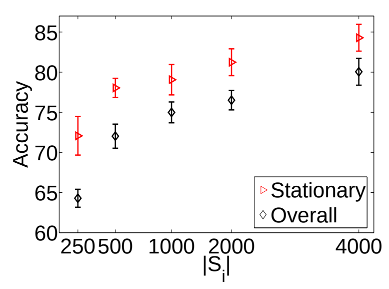
(b) Normalized messages vs. number of peers.



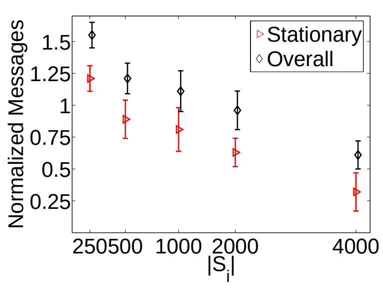
(c) Normalized bytes transferred vs. number of peers.

Fig. 4. Dependence of the quality and cost of the decision tree algorithm on the number of peers.

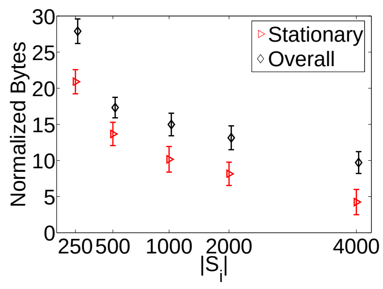
and stationary bytes reduce from 20.9 to 4.24 as the size of the local dataset is increased from 250 tuples per peer to 4000 tuples per peer. This is true since with increasing $|S_i|$, the global tree is induced on a larger dataset, leading to better accuracy. Moreover, with increasing $|S_i|$, the algorithm can capture more variability in the distribution (since the majority votes are run on a larger dataset) leading to lower communication. Even for the smallest dataset size of 250, the normalized messages is 1.21 and the stationary bytes is 20.9, both are far less than 2 and 80 respectively considering the broadcast algorithm.



(a) Quality vs. $|S_i|$.



(b) Normalized messages vs. $|S_i|$.



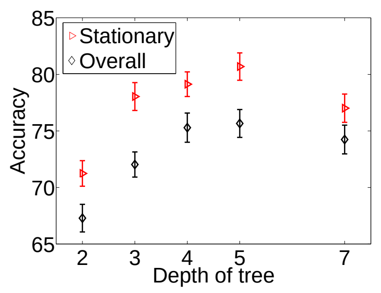
(c) Normalized bytes transferred vs. $|S_i|$.

Fig. 5. Dependence of the quality and cost of the decision tree algorithm on $|S_i|$.

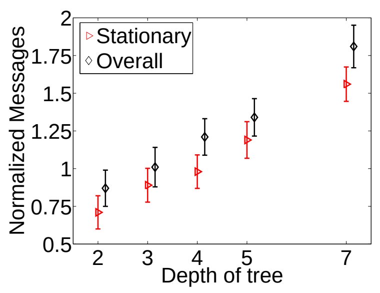
3) Depth of Tree: As pointed out in Section V-C, depth of the decision tree induced by the *PeDiT* algorithm affects the cost of the algorithm. In this section, we validate this result. The experimental results are shown in Figure 6. As shown, the effect of the depth is more pronounced on the communication than on the quality of the result. Accuracy increases from 72% to 81%

as the depth is increased from 2 to 5. However for a depth of 7, the accuracy of the *PeDiT* decreases by 2% compared to a depth of 5. The reason for this is overfitting of the domain. As the depth is increased, there is potentially more ties for every majority vote leading to a message explosion. As the depth is increased from 2 to 7, the stationary messages increase from 0.71 to 1.56. The stationary bytes goes up to 58.71, for a depth of 7.

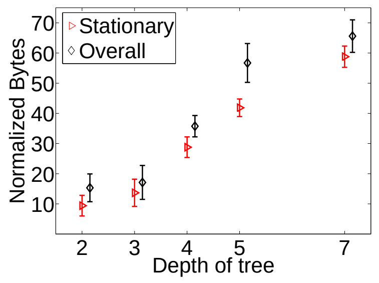
Although the induced tree of depth 5 is around 3% more accurate than the tree of depth 3, we have used trees of depth 3 in all as a baseline. This is because a tree of depth 3 has far lower communication overhead than a tree of depth 5 (0.89 normalized messages for depth of 3 compared to 1.19 normalized messages for depth of 5).



(a) Quality vs. depth of the tree.



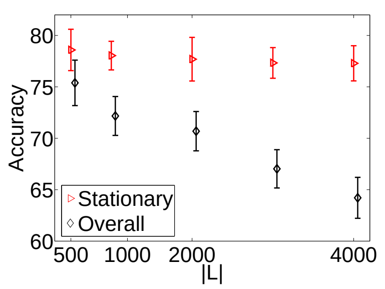
(b) Normalized messages vs. depth of the tree.



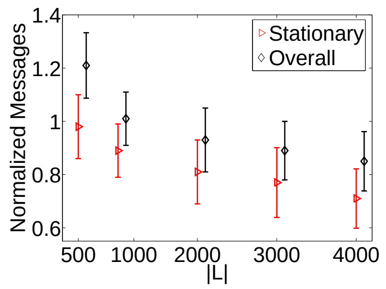
(c) Normalized bytes transferred vs. depth of the tree.

Fig. 6. Dependence of the quality and cost of the decision tree algorithm on the depth of the induced tree.

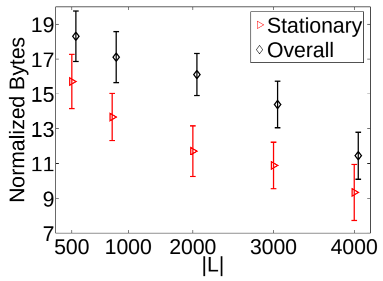
4) Size of Leaky Bucket: The last algorithm parameter that we have experimented with is the leaky bucket mechanism. In this section we present the effect of the size of the leaky bucket $|L|$ on the accuracy and the cost of the *PeDiT* algorithm. Figure 7 summarizes the effect. As shown in Figure 7(a), the stationary accuracy remains constant even as the size of the leaky bucket is made twice or thrice of the edge delay (which is roughly 1100 time units). The overall quality degrades. This is exactly what we expect. As $|L|$ is increased, for every epoch change, the algorithm takes more time to converge, thereby carrying inaccurate results for a longer time. For this reason the overall accuracy degrades. However, once the algorithm adapts to the new distribution, a small number of messages is sufficient to maintain correctness. This is why the leaky bucket has no effect on the stationary accuracy. Contrary to the quality, the cost reduces drastically, from 0.98 stationary messages per peer for $|L|$=500 to 0.71 for $|L|$=4000. Similar is the trend for the bytes transferred.



(a) Quality vs. size of $|L|$.



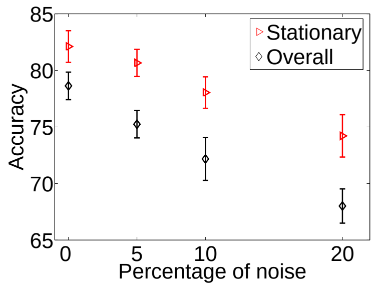
(b) Normalized messages vs. size of $|L|$.



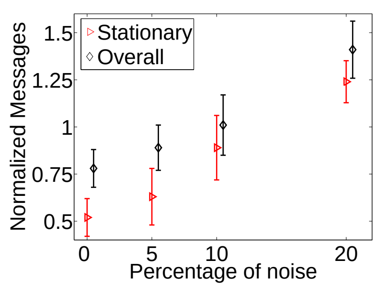
(c) Normalized bytes transferred vs. size of $|L|$.

Fig. 7. Dependence of the quality and cost of the decision tree algorithm on the size of the leaky bucket.

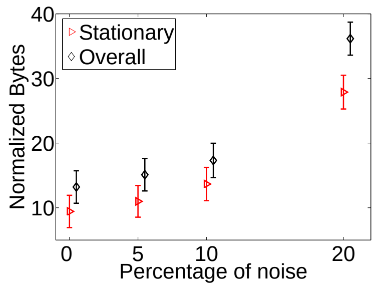
5) Noise in Data: In this section we vary one of the data parameters — noise. As the noise in the data is increased from 0% to 20%, quality degrades and cost increases. This is demonstrated in Figures 8. In Figure 8(a), the stationary accuracy decreases from 83% to approximately 75%. The stationary messages increase from 0.52 to 1.24 and stationary bytes increase from 9.43 to 17.89 as demonstrated in Figures 8(b) and 8(c) respectively. This happens because with increasing noise, every comparison consumes more resources to decide the better one and this decision can often get flipped every time the data changes. As a result, the quality degrades and the cost increases. The important observation here is that even for the highest noise, the number of bytes transferred is 17.89, far less than the maximal allowable rate of 80.



(a) Quality vs. noise.



(b) Normalized messages vs. noise.



(c) Normalized bytes transferred vs. noise.

Fig. 8. Dependence of the quality and cost of the decision tree algorithm on noise in the data.

6) Number of Attributes: The last parameter we varied is the number of attributes. We have measured the effect in three different ways — (1) increasing the number of attributes while keeping the #tuples constant (Figures 9(a), 9(b) and 9(c)), (2) increasing the number of attributes while increasing the #tuples linearly with the #attributes (Figures 9(d), 9(e) and 9(f)), and (3) increasing the number of attributes while increasing the #tuples linearly with the size of the domain (Figures 9(g), 9(h) and 9(i)).

As the number of the attributes is increased keeping the number of tuples constant (at 500 tuples per peer), the stationary accuracy decreases from 73% to 63% (Figure 9(a)). Similarly Figures 9(b) and 9(c) demonstrate that the normalized messages increase from 0.9 to 1.25 and the normalized bytes increase from approximately 17 to 92. Note that for the number of attributes=10, 15 and 20, the maximal bytes transferred for a broadcast-based algorithm is 80, 120 and 160 respectively.

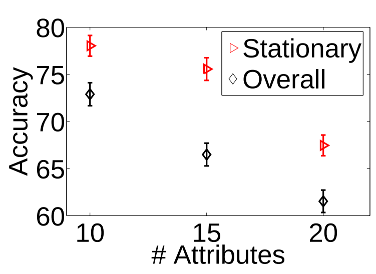
The second set of Figures 9(d), 9(e) and 9(f) show the effect on the number of attributes as the number of tuples is increased linearly with the number of attributes (such that, #tuples=$50\times$ #attributes. For 10 attributes we have used 500 data tuples per peer. We have increased it to 750 tuples for 15 attributes and further increased it to 1000 tuples for 20 attributes. The accuracy degrades from 73% to 63%. The more interesting is the effect on the communication. The stationary messages increase very slowly (from 0.89 to 0.92), demonstrating the fact that the algorithm is scalable.

One last variation is the relationship of number of attributes when the number of tuples is increased in proportion to the size of the domain (#tuples=1%$\times 2^{\#attributes}$). For 10, 15 and 20 attributes, the number of tuples per peer we used are 10, 330 and 10000 respectively. The accuracy improves and the normalized messages decrease as the number of attributes is increased. The number of bytes transferred increases, though for number of attributes=20, it is still well below what would have been used for a broadcast-based algorithm.

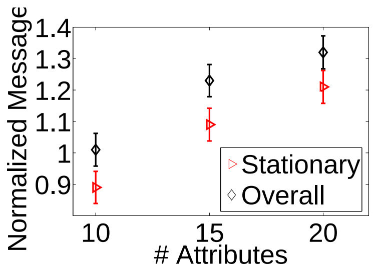
E. Discussion

In the previous section we have presented the quality and the cost of the *PeDiT* algorithm on the different algorithm parameters. Our findings can be summarized as follows.

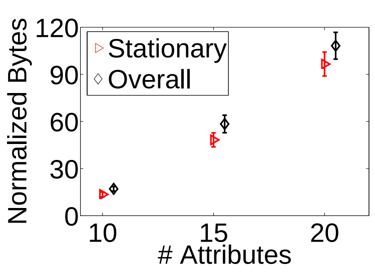
- In most cases *PeDiT* results in a loss of accuracy over J48. This is because of the simpler gain function that we have chosen. However, due to the heavy communication and synchronization cost of centralizing and applying J48, the observed modest loss of accuracy seems quite reasonable.



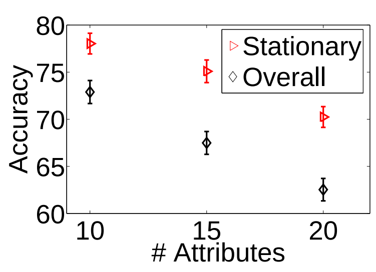
(a) Quality for #tuples=const.



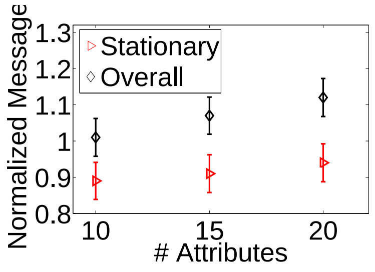
(b) Normalized msg. for #tuples=const.



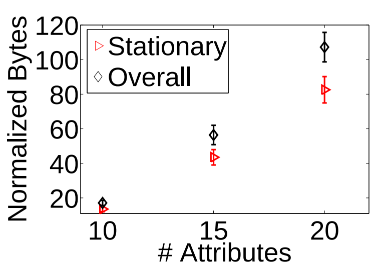
(c) Normalized bytes for #tuples=const.



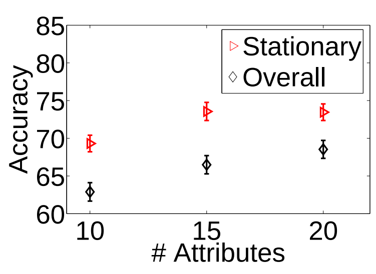
(d) Quality when #tuples increases linearly with #attr.



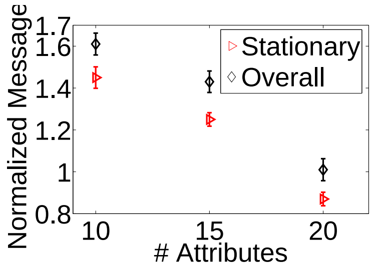
(e) Normalized msg. when #tuples increases linearly with #attr.



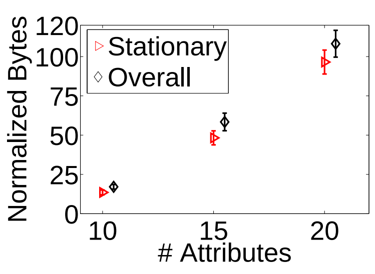
(f) Normalized bytes when #tuples increases linearly with #attr.



(g) Quality when #tuples increases linearly with |domain|.



(h) Normalized msg. #tuples increases linearly with |domain|.



(i) Normalized bytes #tuples increases linearly with |domain|.

Fig. 9. Dependence of the quality and cost of the decision tree algorithm on the number of attributes.

- The *PeDiT* algorithm is highly scalable with respect to the number of peers. As shown by our scalability experiments, increasing the number of peers has little effect on the quality or cost.
- Increasing the number of tuples increases the accuracy of the tree and decreases the cost. Even for tuples per peer = a quarter of the size of the domain (250 tuples per peer for 10 attributes), the monitoring cost is 1.25 — less than the maximal allowable cost of 2.0.
- Note that every increment in the number of attributes doubles the search space. The quality and cost of our algorithm remains moderate even of the number of attributes is doubled.

- Noise in the data degrades the quality and cost — it can be compensated either by increasing the number of tuples or increasing the depth of the tree. Depth of three seems to a be moderate choice since the accuracy is good and the monitoring cost is low as well. Increasing the depth improves the accuracy with a heavy penalty on the cost.

VII. Conclusion

In this paper we presented an asynchronous scalable algorithm for inducing a decision tree in a large P2P network. With sufficient time, the algorithm converges to the same tree given all the data of all the peers. To the best of the authors' knowledge this is one of the first attempts on developing such an algorithm. The algorithm is suitable for scenarios in which the data is distributed across a large P2P network as it seamlessly handles data changes and peer failures. We have conducted extensive experiments with synthetic dataset to analyze the different parameters of the algorithms. The results point out that the algorithm is accurate and suffers moderate communication overhead compared to a broadcast-based algorithm. The algorithm is also highly scalable both with respect to the number of peers and number of attributes.

This paper relies on the majority voting algorithm as a building block. The majority voting protocol is a highly efficient and scalable protocol, mainly due to the existence of local pruning rules. In the literature such algorithms are commonly referred to as local algorithms. Previous work on exact local algorithms mainly focused on developing efficient building blocks such as majority voting [8], L2-thresholding [42] and more. In this paper, similar to [38], we leverage these powerful building blocks to show how more complex data mining algorithms can be developed for large-scale distributed systems. In the process we have also shown how complex functions such as entropy need to be simplified to misclassification gain in order to aid in the algorithm development process.

Most of these algorithms use the term 'local' in an intuitive sense; they rely on experimental results to claim the efficiency and scalability of the algorithms. Very recently, researchers have proposed more formal definitions of local algorithms as well [5]. We do not prove the locality of our algorithm since it is not the central focus of our paper. We plan to explore it in the future along with developing other techniques of inductive learning such as Naive Bayes, k-nearest neighbor, SVM's and more.

ACKNOWLEDGMENTS

This research is supported by the NASA Grant NNX07AV70G.

REFERENCES

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

[2] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Wadsworth, Belmont, 1984.

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

[4] K. Liu, K. Bhaduri, K. Das, P. Nguyen, and H. Kargupta, "Client-side Web Mining for Community Formation in Peer-to-Peer Environments," *SIGKDD Explorations*, vol. 8, no. 2, pp. 11–20, 2006.

[5] K. Das, K. Bhaduri, K. Liu, and H. Kargupta, "Distributed Identification of Top-l Inner Product Elements and its Application in a Peer-to-Peer Network," *IEEE Transactions on Knowledge and Data Engineering (TKDE)*, vol. 20, no. 4, pp. 475–488, 2008.

[6] "Amazon Music Recommendation," http://www.amazon.com/.

[7] "PodPlaylist: Sharing iTune Playlists," http://http://www.podplaylist.com/.

[8] R. Wolff and A. Schuster, "Association Rule Mining in Peer-to-Peer Systems," *IEEE Transactions on Systems, Man and Cybernetics - Part B*, vol. 34, no. 6, pp. 2426 – 2438, December 2004.

[9] B. Gilburd, A. Schuster, and R. Wolff, "k-TTP: A New Privacy Model for Large-Scale Distributed Environments," in *Proceedings of SIGKDD'04*, New York, NY, USA, 2004, pp. 563–568.

[10] I. Mierswa, K. Morik, and M. Wurst, "Collaborative Use of Features in a Distributed System for the Organization of Music Collections," in *Intelligent Music Information Systems: Tools and Methodologies*, Shen, Shepherd, Cui, and Liu, Eds. Information Science Reference, 2007, pp. 147–175.

[11] N. Palatin, A. Leizarowitz, A. Schuster, and R. Wolff, "Mining for Misconfigured Machines in Grid Systems," in *Proceedings of SIGKDD'06*, New York, NY, USA, 2006, pp. 687–692.

[12] A. X. Zheng, J. Lloyd, and E. Brewer, "Failure Diagnosis Using Decision Trees," in *Proceedings of ICAC '04*, Washington, DC, USA, 2004, pp. 36–43.

[13] H. Kargupta and K. Sivakumar, *Existential Pleasures of Distributed Data Mining. Data Mining: Next Generation Challenges and Future Directions*. AAAI/MIT press, 2004.

[14] H. Kargupta and P. Chan, Eds., *Advances in Distributed and Parallel Knowledge Discovery*. MIT Press, 2000.

[15] S. Datta, K. Bhaduri, C. Giannella, R. Wolff, and H. Kargupta, "Distributed Data Mining in Peer-to-Peer Networks," *IEEE Internet Computing Special Issue on Distributed Data Mining*, vol. 10, no. 4, pp. 18–26, 2006.

[16] F. V. Jensen, *Introduction to Bayesian Networks*. Springer, 1997.

[17] V. Vapnik, *Statistical Learning Theory*. Wiley-Interscience, 1998.

[18] L. Breiman, "Bagging Predictors," *Machine Learning*, vol. 2, pp. 123–140, 1996.

[19] J. Friedman, T. Hastie, and R. Tibshirani, "Additive Logistic Regression: A Statistical View of Boosting," Dept. of Statistics, Stanford University, Tech. Rep., 1998.

[20] S. J. Stolfo, A. L. Prodromidis, S. Tselepis, W. Lee, D. W. Fan, and P. K. Chan, "JAM: Java Agents for Meta-Learning over Distributed Databases," in *Proceedings of SIGKDD'97*, 1997, pp. 74–81.

[21] A. Bar-Or, D. Keren, A. Schuster, and R. Wolff, “Hierarchical Decision Tree Induction in Distributed Genomic Databases,” *IEEE Transactions on Knowledge and Data Engineering – Special Issue on Mining Biological Data*, vol. 17, no. 8, pp. 1138 – 1151, August 2005.

[22] D. Caragea, A. Silvescu, and V. Honavar, “A Framework for Learning from Distributed Data Using Sufficient Statistics and Its Application to Learning Decision Trees,” *International Journal of Hybrid Intelligent Systems*, vol. 1, no. 1-2, pp. 80–89, 2004.

[23] C. Giannella, K. Liu, T. Olsen, and H. Kargupta, “Communication Efficient Construction of Deicision Trees Over Heterogeneously Distributed Data,” in *Proceedings of ICDM’04*, Brighton, UK, 2004, pp. 67–74.

[24] T. Olsen, “Distributed Decision Tree Learning From Multiple Heterogeneous Data Sources,” Master’s thesis, University of Maryland, Baltimore County, Baltimore. Maryland, October 2006.

[25] S. Merugu and J. Ghosh, “A Distributed Learning Framework for Heterogeneous Data Sources,” in *Proceedings of KDD’05*, 2005, pp. 208–217.

[26] B. Park, R. Ayyagari, and H. Kargupta, “A Fourier Analysis-Based Approach to Learn Classifier from Distributed Heterogeneous Data,” in *Proceedings of SDM’01*, Chicago, IL, April 2001.

[27] D. E. Hershberger and H. Kargupta, “Distributed Multivariate Regression Using Wavelet-based Collective Data Mining,” *Journal of Parallel and Distributed Computing*, vol. 61, no. 3, pp. 372–400, 2001.

[28] C. Guestrin, P. Bodi, R. Thibau, M. Paski, and S. Madden, “Distributed Regression: an Efficient Framework for Modeling Sensor Network Data,” in *Proceedings of IPSN’04*, Berkeley, California, 2004, pp. 1–10.

[29] J. Predd, S. Kulkarni, and H. Poor, “Distributed Kernel Regression: An Algorithm for Training Collaboratively,” *arXiv.cs.LG archive*, 2006.

[30] S. Banyopadhyay, C. Giannella, U. Maulik, H. Kargupta, K. Liu, and S. Datta, “Clustering Distributed Data Streams in Peer-to-Peer Environments,” *Information Science*, vol. 176, no. 14, pp. 1952–1985, 2006.

[31] S. Datta and H. Kargupta, “Uniform Data Sampling from a Peer-to-Peer Network,” in *Proceedings of ICDCS’07*, Toronto, Ontario, 2007, p. 50.

[32] S. Mukherjee and H. Kargupta, “Distributed Probabilistic Inferencing in Sensor Networks using Variational Approximation,” *Journal of Parallel and Distributed Computing*, vol. 68, no. 1, pp. 78–92, 2008.

[33] M. I. Jordan, Z. Ghahramani, T. S. Jaakkola, and L. K. Saul, “An Introduction to Variational Methods for Graphical Models,” *Machine Learning*, vol. 37, no. 2, pp. 183–233, November 1999.

[34] T. Jaakkola, “Tutorial on Variational Approximation Methods,” *In Advanced Mean Field Methods: Theory and Practice*, 2000.

[35] S. Datta, C. Giannella, and H. Kargupta, “K-Means Clustering over Large, Dynamic Networks,” in *Proceedings of SDM’06*, Maryland, 2006, pp. 153–164.

[36] D. Kempe, A. Dobra, and J. Gehrke, “Computing Aggregate Information using Gossip,” in *Proceedings of FOCS’03*, Cambridge, 2003.

[37] M. Jelasity, A. Montresor, and O. Babaoglu, “Gossip-based Aggregation in Large Dynamic Networks,” *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 3, pp. 219 – 252, August 2005.

[38] D. Krivitski, A. Schuster, and R. Wolff, “A Local Facility Location Algorithm for Large-Scale Distributed Systems,” *Journal of Grid Computing*, vol. 5, no. 4, pp. 361–378, 2007.

[39] J. Branch, B. Szymanski, C. Gionnella, R. Wolff, and H. Kargupta, “In-Network Outlier Detection in Wireless Sensor Networks,” in *Proceedings of ICDCS’06*, Lisbon, Portugal, July 2006.

[40] P. Luo, H. Xiong, K. Lü, and Z. Shi, “Distributed Classification in Peer-to-Peer Networks,” in *Proceedings of SIGKDD’07*, 2007, pp. 968–976.

[41] Y. Birk, I. Keidar, L. Liss, A. Schuster, and R. Wolff, “Veracity Radius - Capturing the Locality of Distributed Computations,” in *Proceedings of PODC’06*, 2006, pp. 102–111.

[42] R. Wolff, K. Bhaduri, and H. Kargupta, “Local L2 Thresholding Based Data Mining in Peer-to-Peer Systems,” in *Proceedings of SDM’06*, 2006, pp. 428–439.

[43] J. Garcia-Luna-Aceves and S. Murthy, “A Path-Finding Algorithm for Loop-Free Routing,” *IEEE Transactions on Networking*, vol. 5, no. 1, pp. 148–160, 1997.

[44] P. M. Khilar and S. Mahapatra, “Heartbeat Based Fault Diagnosis for Mobile Ad-Hoc Network,” in *Proceedings of IASTED’07*, Phuket, Thailand, 2007, pp. 194–199.

[45] Y. Birk, L. Liss, A. Schuster, and R. Wolff, “A Local Algorithm for Ad Hoc Majority Voting via Charge Fusion,” in *Proceedings of ICDCS’04*, 2004.

[46] N. Li, J. Hou, and L. Sha, “Design and Analysis of an MST-Based Topology Control Algorithm,” *IEEE Transactions on Wireless Communications*, vol. 4, no. 3, pp. 1195–1205, 2005.

[47] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*. Addison-Wesley, 2006.

[48] D. Krivitski, A. Schuster, and R. Wolff, “A Local Facility Location Algorithm for Large-Scale Distributed Systems,” *Journal of Grid Computing (in press)*, 2006.

[49] I. Keidar and A. Schuster, “Want scalable computing?: speculate!” *SIGACT News*, vol. 37, no. 3, pp. 59–66, 2006.

[50] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[51] “DDMT,” http://www.umbc.edu/ddm/Software/DDMT/.

[52] “BRITE,” http://www.cs.bu.edu/brite/.

[53] P. Domingos and G. Hulten, “Mining High-Speed Data Streams,” in *Proceedings of KDD’00*, Boston, MA, 2000, pp. 71–80.

APPENDIX

A decision tree induction algorithm cannot be considered complete without a pruning technique stating which branches over-fit the data. Pruning techniques can be divided between *post*-pruning — removing nodes of whole subtrees after the tree is induced — and *pre*-pruning — instructing the algorithm which nodes not to develop in the first place. Since the *PeDiT* algorithm works on streaming data the idea of first inducing the tree and afterwards pruning it seems less suitable (because there is no specific point in time in which pruning should begin). We therefore focus on pre-pruning heuristics. Several common pre-pruning techniques can easily be adopted by *PeDiT*, because they use simple statistics. We will describe three such techniques.

The simplest pre-pruning technique is to terminate new leave development which the tree reaches a pre-specified depth. The biggest benefit of this technique is that it limits the resources used by the algorithm. This technique is trivial to implement as part of *PeDiT* and performs quite favorably in experiments. It's greatest disadvantage is that the depth limit has to be found in trial and error, and is the same for all leaves.

A second popular pre-pruning technique is to require a minimal degree of gain from every split in the tree, or abort the split if the gain is below the required number. The gain from splitting any given node according to an attribute A^i is the reduction in the number of misclassified examples $|x^i_{00} + x^i_{10} - x^i_{01} - x^i_{11}| - |x^i_{00} - x^i_{01}| - |x^i_{10} - x^i_{11}|$. If indicators are calculated for each of the subexpressions: $s^i = sign\,(x^i_{00} + x^i_{10} - x^i_{01} - x^i_{11})$, $s^i_0 = sign\,(x^i_{00} - x^i_{01})$, and $s^i_1 = sign\,(x^i_{10} - x^i_{11})$ then the previous formula can be rewritten $s^i\,(x^i_{00} + x^i_{10} - x^i_{01} - x^i_{11}) - s^i_0\,(x^i_{00} - x^i_{01}) - s^i_1\,(x^i_{10} - x^i_{11})$. Now, to prove the number of errors decreases in more than a constant ϵ we can hold a majority vote for this expression with the threshold set to $-\epsilon$ rather than to 0 (as shown elsewhere in [48]). Just as in Section V-A.2, this would require holding eight different majority votes per attribute, one for every possible value of s^i, s^i_0, and s^i_1, and then selecting one of them according to the ad hoc result of separate votes on the values of s^i, s^i_0, and s^i_1. Notice that votes for s^i_0 and s^i_1 are held anyhow. Also, the input for the vote for s^i is independent of i so just one extra vote is needed for all A^i. Furthermore, for all nodes but the root, a vote on the value of s^i is actually held in the context of the parent of the node.

A third pruning technique is to require non-negative gain from every split when this gain is measured on a test set rather than on the learning set. This method introduces little complexity beyond the described above: the difference is that each node, starting with the root, should be associated with an additional set of examples and that the input to the votes regarding pruning be taken from that set. Notice that the input to votes of the s^i type should still be taken from the learning set.