Interpretable Models from Distributed Data via Merging of Decision Trees

Artur Andrzejak, Felix Langner, Silvestre Zabala
PVS, Heidelberg University, Germany
Emails: artur.andrzejak@informatik.uni-heidelberg.de,
felix.langner@uni-heidelberg.de, silvestre.zabala@urz.uni-heidelberg.de

***Abstract*—Learning in parallel or from distributed data becomes increasingly important. Factors contributing to this trend include emergence of data sets exceeding RAM sizes and inherently distributed scenarios such as mobile environments. Also in these cases *interpretable* models are favored: they facilitate identifying artifacts and understanding the impact of individual variables. Given the distributed environment, even if the individual learner on each site is interpretable, the *overall* model usually is not (as e.g. in case of voting schemes). To overcome this problem we propose an approach for efficient merging of decision trees (each learned independently) into a single decision tree. The method complements the existing parallel decision trees algorithms by providing interpretable intermediate models and tolerating constraints on bandwidth and RAM size. The latter properties are achieved by trading RAM and communication constraints for accuracy. Our method and the mentioned tradeoffs are validated in experiments on real-world data sets.**

I. Introduction

Learning from distributed data (or generally, in parallel) has received in the last decade a growing amount of attention and this trend accelerates in recent years [1], [2]. Several factors contribute to this development: data sets which exceed memory and computing capacities of single computing nodes, inherent data distribution in sensor networks and mobile environments [3], and ubiquity of multi-core and many-core systems which permit parallel processing of data sets.

There has been many excellent works on supervised learning in this domain [4], [5], [2]. Among them we are interested in classification approaches which yield *interpretable* models. In addition high accuracy, the ability to *understand* a model is still one the primary requirements in real-world applications [6]. This ability permits to detect model artifacts and to identify the key variables influencing the classification outcome. The latter is especially valuable in business domains, where models frequently serve as tools uncovering relationships and optimization.

The most popular interpretable classifiers are the decision trees [7], [8]. Many authors have designed decision tree algorithms working on distributed data [9], [4], [10], [11], [12]. Most of them mimic the classical tree-growing approaches but use distributed versions of the split-point selection methods. While (to our knowledge) they are the only methods which yield interpretable overall models learned on distributed data, they have several weaknesses. First, they assume a "tightly connected" processing environment (such as a cluster in a data center) and do not constrain the number and size of exchanged messages. This can preclude their deployment in truly distributed scenarios such as mobile environments or wide-area networks where the bandwidth can be limited and the latency can be high. Another disadvantage is lack of interpretable *intermediate* models (e.g. decision tree for each data site). Such intermediate models allow for checking early whether learned models contain artifacts and "make sense". They can also expose dominant features of each data site and the differences between them.

We propose an approach which does not mimic a classical (centralized) tree-growing schema. Its core operation is *merging* two decision trees (build over different data but with same attribute set) into a single tree (Sec. III). By cascading or sequential execution of this operation we are able to create a single interpretable model for all (distributed) data sets. Our method allows combining *various* tree-growing methods (including streaming approaches such as VFDT [13]) and yields interpretable models at each intermediate step of the process.

Our tree merging method exploits the geometry of the input (attribute) space. The central idea is to represent each tree as a set of decision regions (iso-parallel boxes), then intersect efficiently the boxes from each of the two trees and finally to induce a tree from the resulting set of boxes. We noticed that while deploying this operation repeatedly (e.g. in a hierarchical fashion) the set of boxes (or the corresponding tree) grows to an unmanageable size. This problem (discussed in Sec. IV-C) is obviously related to the *curse of dimensionality* [8] and it becomes more severe with the number of attributes (Sec. IV-C). Our solution consists in pruning the intermediate box sets by various criteria (e.g. the number of covered original samples, Sec. III-D). To obtain a corresponding (intermediate) decision tree we modify the traditional tree-growing algorithm by treating each "unpruned" box as a data sample (see Sec. III-E for details).

Merging two decision trees into a single one have been studied previously. Works [14], [15] do this by merging rule sets derived from trees. However, the discussion is not comprehensive and omits many special cases. Another method proposed in [3] transforms trees into Fourier spectra and merges them by vector addition in the dual space. While this approach is complete and well-described, it is difficult to extend and to implement (e.g. only binary attributes are considered by the authors); also a performance evaluation is missing. None of the above approaches even reports the problem of size explosion of intermediate trees. Consequently, in the presented form these methods are not feasible in case of many distributed data fragments.

Distributed learning approaches should also consider the practical limitations imposed by the computing environment, including constraints on the memory of individual nodes, size of exchanged messages, and the communication patterns influencing scalability. This can be done by reducing memory and data transfer requirements at the cost of accuracy. Some works on (non-distributed) classification have been studying this problem, notably by the forgetron approach [16] and works on classification of data streams [17]. Our approach allows to parametrize the size of memory devoted to a (partial) model in each step of the distributed model building. The drawback is that (as expected) the classification accuracy drops with smaller data size.

Summarizing, our contributions are the following:

- We propose in Sec. III an efficient approach to merge two decision trees into a single one. It exploits the geometry of the attribute space for a simple yet comprehensive method of combining such trees.
- We consider the problem of growing sizes of intermediate trees and propose a solution based on pruning of the corresponding box sets by various criteria (Sec. III-D). One variant (less accurate) requires one pass over data, the other variant requires multiple passes. The degree of pruning can be parametrized: this allows for trading communication and RAM requirements for accuracy.
- We evaluate the accuracy and running time of our algorithms as well as the requirements vs. accuracy trade-offs on real data sets (Sec. IV). We also illustrate empirically the problem of the growing size of intermediate trees.
- We introduce novel metrics for measuring efficiency of communication patterns of learning algorithms (Sec. II).

Furthermore, we discuss the related work in Sec. V and present our conclusions in Sec. VI.

II. Distributed Learning and its Efficiency

A. Preliminaries

We introduce first terms and symbols used throughout the paper. A *data set* or *training set* D is a collection of n *labeled samples* $(\mathbf{v}, c)$ where $\mathbf{v}$ is element of an *input space* $\mathscr{D}$ and c element of an *output space* $\mathscr{L}$. In classification D is typically sampled from an unknown distribution and $\mathscr{L}$ is a categorical domain: a set of nominal or discrete elements called *labels* or *class values*. During *training* or *learning* phase we search for a *model* $F : \mathscr{D} \rightarrow \mathscr{L}$ that best approximates the true label distribution in D. In the *prediction* phase we assign to an *unlabeled sample* or *input vector* $\mathbf{v} \in \mathscr{D}$ a label $F(\mathbf{v})$ according to the previously learned function F. We assume that $\mathscr{D}$ is spanned by d real-valued, continuous *attributes* (random variables) $X_1, \ldots, X_d$[1].

In the setting of a parallel or distributed system the data set D is split into k (usually disjoint) *fragments* $D_1, \ldots, D_k$. The split of D can be done *vertically*, where a fragment contains a subset of the attributes $X_1, \ldots, X_d$ (and possibly the labels) or *horizontally*, where a fragment comprises a subset of samples [10]. In typical applications n is much larger than d and so we assume the latter scenario. In this context a *global model* is a model created by using as input (possibly processed) information from all fragments $D_1, \ldots, D_k$. Such a global model can be obtained either by a *centralized* or a *distributed* (or equivalently, *parallel*) algorithm. The centralized algorithm can access the whole data set $D = D_1 \cup \ldots \cup D_k$ in a single memory space while the distributed one does not assume the latter condition.

B. Efficiency of distributed learning

There are several aspects which determine the efficiency of a distributed algorithm. One is the *communication efficiency*, that is the number and dependencies of the messages exchanged between the processors. Further aspect is the maximum required memory of a single processor (for simplicity, we assume that all processors have the same memory). Finally, also the total size of data to be exchanged between processors plays a role. We call the two latter aspects *memory* and *transfer efficiency*, respectively.

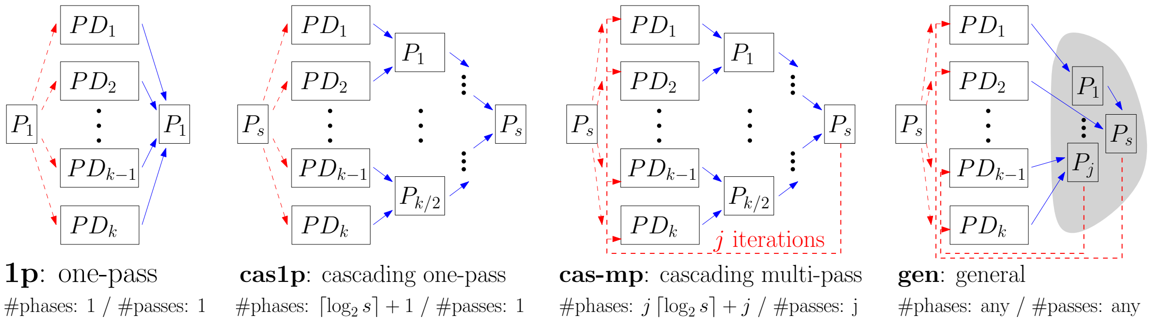
In practical terms, the communication and memory efficiencies have the biggest impact: the former influences greatly the running time and scalability, and the latter determines whether a data set can be processed under given memory size *at all*. In some scenarios, such as sensor nets or mobile devices, the transfer efficiency can be important [3].

The communication patterns of most distributed learning algorithms [5], [2] can be described in terms of *sufficient statistics* [10]. In brief, a part of the algorithm called *learner* L can repeatedly issue queries to each of the processors maintaining a data fragment D_i. Each query q is answered by a *statistic* which can be any function of D_i. Informally, these functions are called *sufficient statistics* if L can build a global model under this schema. While not discussed in [10], L itself can run on a single processor or on a set of them.

Fig. 1 illustrates some common patterns. Here $PD_1, \ldots, PD_k$ denote processors holding data fragments and $P_1, \ldots, P_s$ (other) processors implementing L. We can evaluate their efficiency in terms of two statistics: number of sync phases and number of data passes. A *(sync) phase* is a dependency of a set of processors on outputs of other processors to continue [18]. For example, in the 1p pattern P_1 waits only once for output of all other processors, and so we have only one phase. Number of *data passes* is the number of queries (for sufficient statistics) to $PD_1, \ldots, PD_k$ or a subset of them; if all queries are identical we count them as one as in cas1p pattern. For this pattern, we have one data pass but up to $\lceil \log_2 s \rceil + 1$ sync phases. Figure 1 shows the number of phases and passes for each pattern.

The number of phases and data passes allow for a system-independent evaluation of communication efficiency of a dis-

[1] Extension to discrete or nominal attributes can be trivially achieved via an ordering of their values.



1p: one-pass
#phases: 1 / #passes: 1

cas1p: cascading one-pass
#phases: $\lceil \log_2 s \rceil + 1$ / #passes: 1

cas-mp: cascading multi-pass
#phases: $j \lceil \log_2 s \rceil + j$ / #passes: j

gen: general
#phases: any / #passes: any

Figure 1. Communication patterns and the corresponding number of sync phases (#phases) and data passes (#passes); red/dashed = queries, blue/solid = sufficient statistics

tributed algorithm and are good indicators of its scalability. First note that number of data passes determines whether an algorithm is capable of handling data streams (one pass only) and how long (possibly large) data fragments must be kept in memory. The number of sync phases indicates whether some processors are idle waiting for others. It also translated exactly to the number of runs of the Map-Reduce schema needed to complete the algorithm. Map-Reduce is a very popular approach for massively parallel data processing [5], [18] which can be used to implement directly each of the patterns in Figure 1. However, each additional iteration can add significant overhead of start-up and tear-down times.

Note that the cascading patterns (and possibly the others) utilize repeatedly a `merge`-operation: two inputs (usually partial models) are joined into a single output which is processed further. This operation is used as the key element of the cascade SVM [19] and plays central role in the approach presented in Sec. III.

III. Merging decision Trees

A. From Trees to Sets of Iso-Parallel Boxes

A *decision tree* (or simply a tree) F predicts a label via a sequence of tests on attribute-values (each associated with an *inner node*) where depending on the last test result a next test is selected. This process terminates when a *leaf* of the tree is reached. A leaf has an associated label (result of the prediction) or as an extension a *class probability distribution* - probability of occurrence of each class. Most commonly, a test at inner node compares a values of single attribute against a constant; we assume this in the following. Fig. 2-2 illustrates the prediction process. The attribute values of an input vector $(x, y) \in R^2$ are compared (one at a time) against constants until a leaf is reached.

A tree model F induces by the above process a division of the input space $\mathscr{D}$ into *decision regions*, each corresponding to a leaf of the tree (see Fig. 2-2). In case of assumed single-attribute-tests each region has a form of an *iso-parallel box* (or simply box). Each box is either $\mathbb{R}^d$ or an intersection of half-spaces, each bounded by a hyperplanes parallel to $d-1$ axes of $\mathbb{R}^d$ (thus the term *iso-parallel*). Fig. 2 shows the correspondence between the leaves of the tree F_1, the induced decision boundaries (Fig. 2-3) and the boxes (Fig. 2-4). Each circled number at a leaf of the tree corresponds to exactly one box shown in Fig. 2-4.

For a model tree F let $B(F)$ denote the set of boxes induced by all leaves of F. To obtain $B(F)$ for a tree model F, we only need to "follow" each path from the root to a leaf and update the boundaries of the corresponding box at each inner node (the initial box is $\mathbb{R}^d$). Consequently, the running time is bounded by the number of leaves in F times tree depth. It is worth noting that boxes in $B(F)$ are pairwise disjoint and that $B(F)$ partitions exhaustively the input space. This follows from the fact that a tree assigns an input vector (any element of $\mathscr{D} = R^d$) to exactly one leaf.

Another representation form of a box frequently encountered in the literature is a *decision rule* [14]. Such a rule corresponds to a single leaf K. If a prediction process reaching K passed tests $T_0, \ldots, T_i$, then the corresponding rule has the form

`if` T_0 `and` T_1 `and ... and` T_i `then <class of` K`>.`

In the tree from Figure 2-2, node ④ is associated with the rule `if` $y \geq 0.25$ `and` $x \geq 0.5$ `and` $y \geq 0.625$ `and` $x < 0.75$ `then class` A. This can be simplified to the rule `if` $0.5 \leq x < 0.75$ `and` $0.625 \leq y$ `then class` A which immediately translates to box ④ in Fig. 2-4.

B. Overview of the Merging Approach

The process of merging k decision trees $F_1, \ldots, F_k$ into a single one starts with creating for each tree F_i its box set $B(F_i)$ as explained in Sec. III-A. The collection $B(F_1), \ldots, B(F_k)$ is reduced into a final box set B' by repeated application of the operation `mergeBoxset` (see below) on pairs of box sets. Finally, B' is turned into a decision tree as described in Sec. III-E.

The operation `mergeBoxset` is the core of the approach: it merges two box sets B_1, B_2 into a single one. It consists of several phases. First, we *unify* B_1 and B_2 by computing pairwise intersections of all pairs (b_1, b_2) with $b_1 \in B_1$, $b_2 \in B_2$ and resolving potential label conflicts (`unify`-operation, Sec. III-C). The next phase reduces the size of the resulting box set merging adjacent boxes with same labels (Sec. III-D1) and further pruning of some boxes based on box ranking (Sec. III-D2). Finally, a decision tree F on the

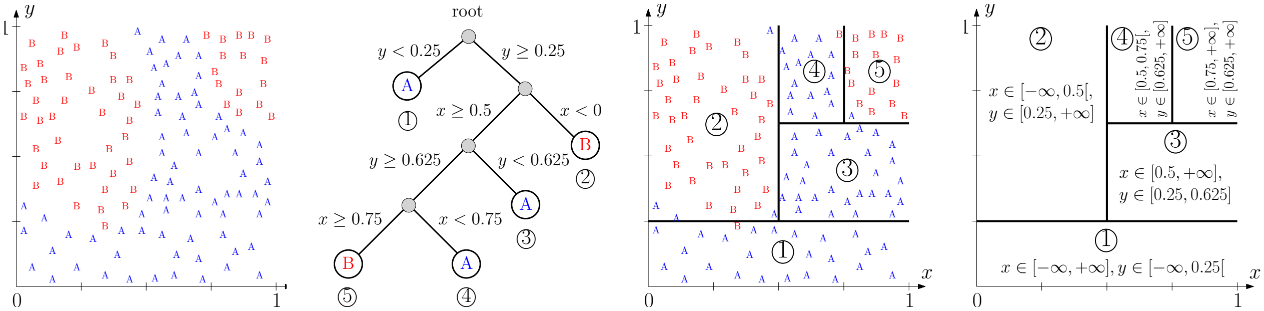


Figure 2. 1: A dataset with attributes x, y and classes A, B; 2: Corresponding decision tree F_1; 3: Decision boundaries of F_1; 4: Boxes derived from F_1

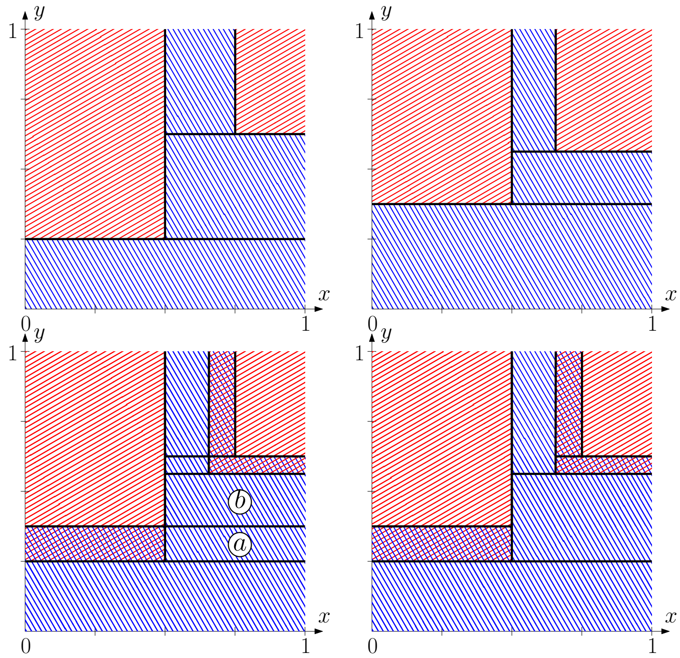


Figure 3. Upper: Box set $B(F_1)$ of the tree from Fig. 2 (left) and box set $B(F_2)$ of another tree F_2 (right); Lower: Unified boxes of $B(F_1)$ and $B(F_2)$ before (left) and after (right) joining adjacent boxes described in Sec. III-D1

remaining boxes is grown as discussed in Sec. III-E. For intermediate models, this tree F used to get a box set to be merged (by the procedure from Sec. III-A).

C. Unifying Box Sets

The `unify`-operation of two box sets B_1, B_2 yields a new box set $B' = \texttt{unify}(B_1, B_2)$. It consists of two steps:

A. For each pair of boxes (b_1, b_2) with $b_1 \in B_1$, $b_2 \in B_2$ we compute the intersection $b = b_1 \cap b_2$ and add it to B' (Sec. III-C1). It is not hard to see that at the end of this process B' covers completely the input space $\mathscr{D} = \mathbb{R}^d$ and no two boxes in B' overlap. Fig. 3 illustrates this step: two sets of each five boxes (Fig. 3-1 and Fig. 3-2) are unified, yielding a set with 10 intersection boxes (Fig. 3-3).

B. For a box b in B' we compute a label and class probability distribution associated with b (Sec. III-C2). Observe that this information depends solely "parents" b_1, b_2 of b: Let $\mathbf{v} \in \mathscr{D}$ be an input vector with $\mathbf{v} \in b$. Obviously $\mathbf{v}$ is in exactly one box $b_1 \in B_1$ and in exactly one box $b_2 \in B_2$ and so the label assigned to $\mathbf{v}$ ("via" b) shall depend only on b_1 and b_2.

1) Efficient identification of intersecting boxes: Given box sets B_1 and B_2, a naive implementation of Step A would require $O(|B_1| \cdot |B_2|)$ tests whether two boxes intersect (each with $O(d)$ steps). We have implemented a more scalable solution resembling (per dimension) a line sweep algorithm. This approach requires $O(d \cdot (|B_1| + |B_2|) \cdot \log(|B_1| + |B_2|) + |S|)$ steps which is in practice much less that number of steps of the naive approach. We describe in the following the details.

In the main loop we iterate over all dimensions (i.e. attributes) indexed by $i = 1, \ldots, d$. In each iteration i we compute set S_i of box pairs (b_1, b_2) with $b_1 \in B_1$, $b_2 \in B_2$ whose *projections* on the dimension i intersect (described below). The intersection $S = S_1 \cap \ldots \cap S_d$ contains exactly the pairs of boxes from B_1 and B_2 which intersect. For each such pair (b_1, b_2) we compute in the last step the actual intersection $b_1 \cap b_2$. The set of all these box intersections constitutes B, the consolidated box set of B_1 and B_2.

Computation of the set S_i for dimension i is implemented as follows. We insert in a list E the starting and ending boundaries (for dimension i) of all boxes in B_1 and in B_2 and annotate each entry by the origin information (i.e. triple (B_1or B_2, starting or ending boundary, pointer to the box)). Subsequently E is sorted and the entries scanned ("line sweep") by growing values. At each entry of E we update an *active set* for boxes in B_1 and an *active set* for boxes in B_2. Such an active set is in essence the set of boxes whose starting boundary (at dimension i) have been encountered in the scan of E but whose ending boundaries are yet to come. By comparing both active sets for each entry of E we can easily identify box pairs whose projection at dimension i intersect.

The running time for computation of S_i is bounded by $O((|B_1| + |B_2|) \cdot \log(|B_1| + |B_2|))$ (for creating E, sorting and scan) plus number of found pairs (which can be in worst case $O(|B_1| \cdot |B_2|)$). Our implementation includes further optimizations, e.g. special treatment of boxes unbounded in dimension i. We observe in our evaluation that this number intersection pairs for dimension i is similar to the final output size $|S|$. Thus, the implemented algorithm requires $O(d \cdot (|B_1| + |B_2|) \cdot \log(|B_1| + |B_2|) + |S|)$ steps.

2) Computing labels and conflict resolution: Computing the label of a box $b = b_1 \cap b_2$ (step B) is easy when both "parent boxes" b_1, b_2 have the same label: it is also assigned to b. Otherwise we have a label conflict; we call then b a *conflict box*. Fig. 3 (3) shows 3 conflict boxes (falling and rising patterns overlap). We propose several strategies for computing label and class probability information for a conflict box b ("CR" is for conflict resolution):

CR1. This strategy assigns b the label of b_1, b_2 with higher confidence.

CR2. We average the two class probability distributions associated with b_1 and b_2 (see Sec. II-A) and select the class with highest probability in the result as associated label.

CR3. This (more expensive) method requires an additional pass through the input data. In a first step we filter all fragments D_1, ..., D_k retaining only those (labeled) samples $(\mathbf{v}, t)$ whose input vectors $\mathbf{v}$ are contained in b. The resulting set of labeled vectors $D(b)$ gives us a class probability distribution of all samples in b and specifies the associated label (in our implementation it is the most frequent class in $D(b)$).

D. Pruning Boxes

The reduction of size of a box set B obtained after unification and conflict resolution is necessary for the following reason. During experiments we experienced that $|B|$ was 2 to 8 times larger than the input size (i.e. $|B_1|+|B_2|$). Moreover, this factor was larger with higher d (see Sec. IV-C). This caused the size of the final box set to grow unfeasibly large, even for number of fragments $k \leq 8$. Sec. IV-C discusses likely reasons for this effect.

We first tried to conquer this problem joining adjacent boxes with same labels (Sec. III-D1 and Fig. 3). While this eliminated up to 50% of boxes, it was not sufficient. A better solution turned out to be reduction of box set size ranking of the boxes and their pruning (Sec. III-D2).

1) Joining adjacent boxes: The box set B after conflict resolution may contain boxes which can be merged at one dimension and which have the same label. They can be merged, thus reducing the overall size of B. An example is shown in Fig. 3 (bottom left), where boxes marked as a and b can be merged to a single new box (as done in Fig. 3 (bottom right)).

The algorithm for this purpose is a simplified version of the line sweep approach from Sec. III-C1. For each dimension $i = 1, \ldots, d$ we iterate over all boxes in B and insert their starting and ending boundaries (at dimension i) into a list. For each boundary value v_i in this list we check all pairs of boxes b and b' where b ends and b' starts at v_i (or vice versa) whether they can be merged. The latter is the case if their boundaries agree in all other dimensions but i and their class values are equal. The running time of this algorithm is dominated by creating and sorting the list for each dimension. Thus, the time complexity is $O(d \cdot |B| \cdot \log(|B|))$.

2) Pruning by box ranking: To further reduce the number of boxes in B we initially tried to prune the tree (created from B, Sec. III-E) by traditional methods like pre- and postpruning [8]. However, it turned out that direct box set pruning described below gives much more control over accuracy and the box set size.

In general, we first rank boxes in B by one of the criteria R1 or R2 (below) and retain at most max_{box} best-ranked of them (max_{box} is an input parameter). The ranking criteria are:

R1: The rank of a box b is its *relative volume*: volume of the box divided by the volume of the *bounding box* induced by a subset D' of training samples (i.e. smallest box covering all samples in D'). As some dimensions of b might not be bounded, we use b's intersection with the bounding box for volume computation. A bounding box can be computed easily for an original fragment D_i and is retained as a property of box set data structure. If two such box sets B_1, B_2 are unified, we associate with the result the smallest box enclosing the bounding boxes of B_1 and B_2. Thus, a bounding box for a box set can be maintained easily and so this criterion is computationally inexpensive.

R2: We compute for each box b in B the number of training samples inside b and use this as box rank. The rationale is to eliminate meaningless boxes (e.g. artifacts of intersections) which do not describe the original data. In this case we modify the pruning strategy such that boxes with rank 0 are not retained even if the total number of remaining boxes is below max_{box}.

While the set of boxes retained after pruning does not completely cover the input space $\mathbb{R}^d$, it is not hard to see that the tree-growing algorithm from Sec. III-E works without changes and yields a tree which assigns a unique label to each test sample (i.e. "covers" $\mathbb{R}^d$ completely).

Criterion R2 yields accurate (and small) trees but it is also computationally costly, as it requires a dedicated pass through all data fragments. Contrary to this, R1 does not need any data pass but the accuracy of the resulting trees is generally worse than for R2. By inspecting intermediate results we found out that for $d \geq 5$ boxes with high relative volume are not necessary those with high number of covered training instances (i.e. having high R2 rank). Thus, R1 can wrongly promote boxes which do not capture information from the original data set. We have also experimented with various other criteria like the relative length of longest box side (projection on i-th dimension) or the Information Gain of the class probability distribution, but none of them was better than R1 or R2.

E. Growing a Decision Tree

Our technique for tree creation from a set of boxes mimics a traditional recursive algorithm such as C4.5 for building a (binary) decision tree (see e.g. [7], [8] for details). The essential difference is that boxes are treated as training samples. This has two consequences for the recursive splitting of a box set B: (i) some of the boxes from B need to be bisected in two parts, which increases the total number of boxes; (ii) we use a proprietary method for selecting split dimension d_s and the split-point v_s.

The first point arises from the fact that contrary to the case of samples, a box from B can be on *both* sides of a split-hyperplane. In detail, given fixed d_s and v_s, a hyperplane h_s induced by them intersects $d_s$axis at v_s and so bisects $\mathbb{R}^d$. As in the case of samples, all boxes in B which are *completely* on one side of h_s are assigned to the "left" or "right" subbranch of the current inner tree node. However, some boxes in B can be intersected by h_s. We bisect each such box into two new boxes according to this intersection and assign them to the separate tree subbranches.

The choice of the split dimension d_s and the split-point v_s heavily influences the number of the boxes which are bisected. We have experimented first with the traditional "impurity" criteria such as Information Gain [8]. However, this method yielded (for some datasets) a very large number of dissections, nearly doubling the number of remaining boxes in a single recursive tree building step. A better criterion is simply selecting d_s and v_s which dissects the least number of boxes in B' (see Sec. IV-C and Fig. 5 (right)). We use it in our approach.

F. Communication Efficiency

The execution cost of the approach is largely determined by the number of passes through the input data. Note that the conflict resolution strategies CR1 and CR2 (Sec. III-C2) do not require any additional passes as well as the tree pruning with box ranking strategy R1. Contrary to this, the conflict resolution strategy CR3 and the box ranking strategy R2 need a dedicated data pass for a single mergeBoxset-operation. Consequently, we combine these cases together to the following classification approaches:

TMo (**t**ree-**m**erging **o**ne-pass) Here the mergeBoxset-operation uses CR1 or CR2 together with R1.

TMm (**t**ree-**m**erging **m**ultiple-passes) Here mergeBoxset-operation uses CR3 combined with R2. Note that this can be implemented by sending the box set returned by the unify-operation to each of the processors holding a data fragment. These return for each box the number of samples per label. By aggregating this data over all fragments the processor executing mergeBoxset-operation obtains sufficient data for conflict resolution and box ranking.

On most data sets the choice of tree-pruning criterion has much more impact on accuracy than choice of the conflict resolution strategy (which CR3 is consistently best). Due to large costs for R2 and CR3 other combinations are not practical. In both cases we exclude the joining of adjacent boxes (Sec. III-C1) as this operation did not reduce the number of boxes significantly (the reduction was at most 30%).

To obtain a final model on k fragments, we need $k-1$ mergeBoxset-operations, either via the cas1p or the 1p communication pattern (Sec. II-B) for TMo and gen-pattern for TMm (as each mergeBoxset requires a data pass). Consequently, TMo requires a single pass through data and TMm k dedicated passes (implemented as above).

ID	UCI Name / Link	#samples	#attributes	#classes
D1	Cardiotocography (NSP)	2126	23	3
D2	Wall-Following-Robot	5456	24	4
D3	Spambase	4601	57	2
D4	Magic-Gamma-Telescope	19020	11	2
D5	Letter-Recognition	20000	16	26
D6	MiniBooNE (balanced)	73001	50	2

Table I
DATA SETS USED FOR EMPIRICAL EVALUATION

The per-node memory requirement of the algorithm depends on the maximum number of boxes, max_{box}. For the unify-operation we keep both input box sets B_1, B_2, data structures of similar size and the results (non-pruned box sets). The latter can be (roughly) up to 8 times larger than $|B_1| + |B_2|$. As each box needs $2d$ scalars for geometry and a vector for class distribution (size $|\mathscr{L}|$), the total storage requirements is bound by $20max_{box}(2d + |\mathscr{L}|)$ scalars.

IV. EVALUATION

In this section we evaluate empirically the accuracy and running time of the introduced algorithms. We also analyze their communication and memory efficiencies. We used several real-world data sets with growing number of instances from the UCI Machine Learning Repository [20] shown in Table I (data set D6 has been changed to have equal number of instances from both classes).

We compare the accuracy of own approaches against two standard methods which serve as a baseline. The notations are the following:

TMo tree-merging one-pass approach (Sec. III-F)

TMm tree-merging multiple-pass (Sec. III-F)

VC "voting" classifier which is obtained by splitting a training set into k fragments, building a C4.5-tree on each fragment separately, and using a majority vote of the trees as the output. This approach serves as a baseline for a distributed (yet not interpretable) classification approach.

J48 WEKA implementation of the C4.5 decision tree (with default WEKA-settings) [21]. Shows the accuracy of a centralized decision tree.

For each accuracy result we have used a 10-fold cross-validation. This implies that each classifier was built on 90% of the number samples stated in Table I and tested on 10% of this number. In case of J48, the training set was used in a single piece, otherwise it was into k equal fragments (in order of original data). We have used the percentage of correctly classified samples as an accuracy metric as our data sets have different numbers of classes.

There are few parameters used in the experiments. Most important is k, the number of data fragments (Sec. II-A). We

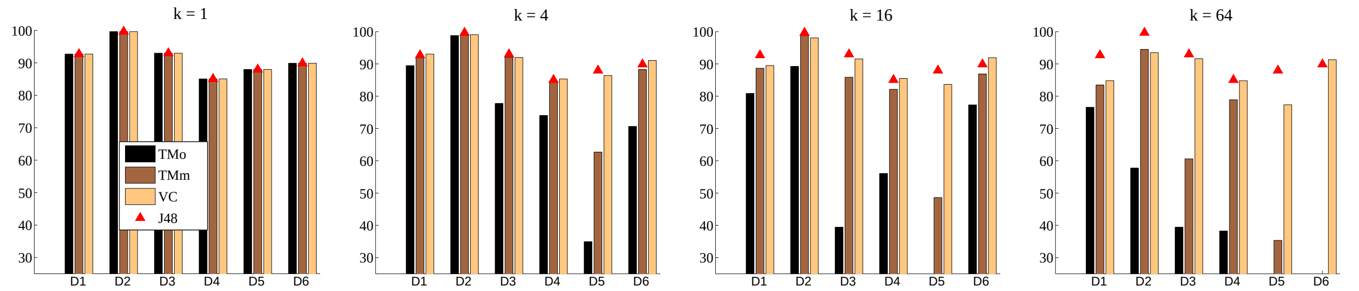


Figure 4. Accuracy (y-axis) of the approaches (bars within group) for all data sets (bar groups) and $k = 1, 4, 16, 64$ ($frac_{|D|} = 0.5$ and $max_{box} = 1000$)

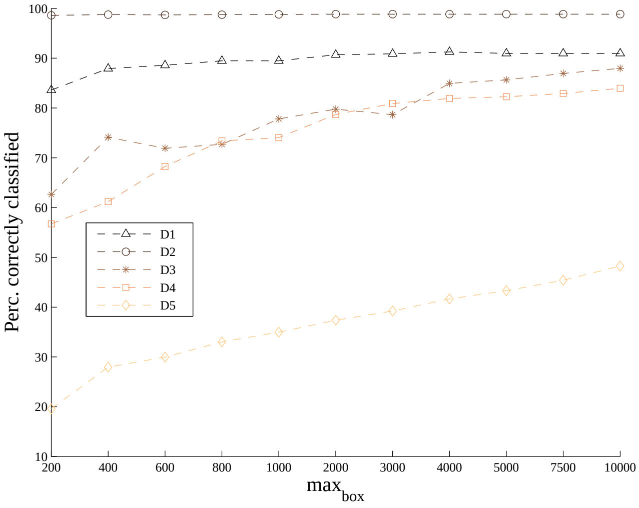


Figure 5. Accuracy (y-axis) of TMo for various data sets depending on the parameter max_{box}

fragment	D_0	D_1	D_2	D_3	D_4	D_5	D_6	D_7
tree size	58	64	45	61	43	46	50	41
unification	1033		952		619		606	
# conflicts	500		465		300		291	
pruning	642		587		389		395	
unification	19593				13905			
# conflicts	9092				6467			
pruning	1010				819			
unification	39026							
# conflicts	15907							
pruning	1005							

Table II
NUMBER OF BOXES AFTER UNIFICATION, AMONG THESE: WITH CONFLICTS (“# CONFLICTS”) AND AFTER PRUNING (TMM ON D4, $k = 8$)

also change as a parameter the maximum number of boxes max_{box} after pruning (Sec. III-D). This parameter determines the trade-off between “required-node-memory” (and also indirectly message sizes) vs. accuracy.

A. Overall Accuracy

We first compare “best possible” accuracy of the new approaches, i.e. with the trade-off parameters set to very high values favoring accuracy, namely $frac_{|D|} = 0.5$ and $max_{box} = 1000$. The result is shown in Fig. 4. For $k = 1$ (i.e. “centralized” processing) all algorithms perform similarly to the centralized J48, but for higher values of k the situation changes. The TMo algorithm performs weakly already for $k = 4$ for some data sets and for all of them for higher k’s. The TMm classifier faces some accuracy problems at $k = 16$ and even more of them for $k = 64$. Our conclusion (also confirmed by other experiments) is that the TMo / TMm approaches can be used only for small number of fragments (up to 16), with TMm being significantly more accurate than TMo.

B. Impact of Parameters

Fig. 5 shows dependency of TMo on max_{box} for various data sets (for $k = 4$). There is a clear trade-off between memory requirements per node (proportional to max_{box}) and accuracy. However, the number of boxes needed for reasonable results is on the order of thousands, which is much more than the typical size of trees built on individual data fragments by VC (few dozens, not shown). This shows that there is a very high price to pay for merging decision trees in one-pass fashion with the given box ranking scheme. The TMm classifier does not show the dependency of accuracy on max_{box} as boxes without any training data samples are always eliminated.

C. Number of Boxes

Table II shows the phenomenon of large number of boxes after the conflict resolution (Sec. III-B) for TMm ($max_{box} = 1000$). Here the training set of D4 is split into $k = 8$ equal fragments, each with 1426 samples. The J48-trees built on these fragments have between 41 and 64 leaves. For all 3 phases of merging (pattern cas1p) we show the number of boxes after `unify`-operation (“unification”), among these the number of conflicts (“# conflicts”) and the final size after pruning (“pruning”). For example, in the last phase two box sets with sizes 1010 and 819 are unified, yielding 39026 boxes with 15907 conflicts; pruning reduces this number to 1005 boxes.

We explain the large size of the unified box set by several causes. First, the attributes and split-points at top-level inner nodes of a tree F influence the boundaries of a large share of the boxes in $B(F)$. Thus, if trees F_1, F_2 built on fragments D_1 and D_2 have only slight differences in the top-level nodes, the box sets $B(F_1)$ and $B(F_2)$ can contain largely different boxes.

These differences propagate to the further merging phases, as box boundaries are not changed.

The second reason is related to the to the curse of dimensionality [8]: two overlapping but not identical boxes $b_1 \in B_1$, $b_2 \in B_2$ can produce (in addition to $b_1 \cap b_2$) nearly up to 2^{2d} further boxes in the unified box set. For example, b_1 can bisect b_2 by hyperplanes spanned by each of the $2d$ faces of b_1. This explains why this effect is much larger with growing number of attributes d.

D. Runtime Efficiency

The algorithms have been implemented as a (single-threaded) simulation of a distributed environment in Java (JVM 1.6.0_26) and Groovy++ (0.9.0_1.8.2) with WEKA 3.6.2. We used as operating system Debian 6.0.5 (64 bit, kernel 2.6.32-5-amd64) on a KVM virtual machine with 8 GB RAM (max. 2.5 GB Java heap space) and I7-2600 host processor.

In Figure 6 we evaluate for TMm the running times of all essential operations listed in Sec. III-B as a function of total input (i.e. size of an input box set or sum of these sizes for the `unify`-operation). The shown operations are mandatory to obtain a tree given two input box sets. The operation of joining of adjacent boxes from Sec. III-D1 is not shown as it was not used in experiments. The shown relations facilitated the estimation of the cumulative running time until final tree is created (in either the cascading or sequential execution schema). Of course, in a distributed environment time for data transfer needs to be added as well.

The `unify`-operation (Sec. III-C) obviously profits from the efficient identification of intersection boxes (Sec. III-C1), yielding low running times. Also the tree growing operation (Sec. III-E) executes very efficiently. The conflict resolution and pruning operations (show for TMm) need much longer. This can be attributed to our rather inefficient implementation of determining *which* (for conflict resolution CR3) or *how many* (for pruning with R2 criterium) training samples are inside of each box (we just use a nested loop here). For the TMo version, these operations run significantly faster (similarly as the `unify`-operation).

We also evaluate the efficiency of algorithms in terms of the “abstract” metrics shown in Table IV-D (see also Sec. II-B). Obviously TMo is efficient in terms of memory and communication pattern; however, its low accuracy limits its applicability to $k \leq 4$. For TMm the limiting scalability factor can be (depending on the system and data set) either the accuracy ($k \leq 16$) or the costly communication pattern (gen) which requires in this case k passes over data.

V. Related Work

Parallel and distributed data mining. Interest in parallel and distributed machine learning exists since more than two decades, with a variety of contributions in unsupervised and supervised learning; see [1]. While these solutions can be considered as independent and sometimes platform-specific, some recent works attempt to unify the approaches either via theory such as sufficient statistics [10] or via common programming paradigms such as Map-Reduce. For example, [5] outlines how 10 popular algorithms (from linear regression to SVMs) can be implemented on Map-Reduce; the project Apache Mahout provides real implementations of several algorithms under this paradigm. A recent trend are algorithms tuned to new type of hardware such as GPUs and FPGAs, see [2].

Learning decision trees on large data sets. Contributions to large-scale tree learning include centralized algorithms which avoid RAM limitations by using efficiently disc storage, and parallel/distributed algorithms. The former include approaches such as BOAT, CLOUDS, RAINFOREST, SLIQ (see [4] for their discussion and references). Most parallel or distributed algorithms mimic a centralized tree-building process, frequently with variations of the split-point selection methods. Works [11], [10], [4] follow this schema, with the last one utilizing the Map-Reduce paradigm. See the excellent survey [12] for many other approaches in this family.

Such distributed/parallel approaches assume their deployment in a cluster environment which does not constrain the number and size of exchanged messages (and has low latency). As a consequence, they are inefficient in truly distributed scenarios such as mobile environments or wide-area networks where the bandwidth can be limited and the latency can be high. Another disadvantage is lack of interpretable intermediate models (e.g. decision tree for each data site). Contrary to this, our approach uses trees as intermediate models and is more suitable for “loosely coupled” distributed environments with constraints on bandwidth and node capacity.

Learning on data streams and under memory budget. Streaming algorithms work under a fixed memory budget and thus consider memory limitations as discussed in Sec. II-B. Among centralized approaches for tree construction VFDT is probably the most popular method [13]. [9] propose a parallel and streaming algorithm; this approach mimics a standard tree-growing process and is shown to scale up to 8 processors. Learning with a fixed memory budget has been also studied in context of SVMs [16]; also SVM-approaches which trade memory for accuracy exist [17]. These methods are either not distributed or yield no interpretable models, while our approach offers both properties.

Merging of classification trees. Merging rule sets derived from trees is closely related to our geometry-based tree-merging approach in Sec. III. While several contributions on this topic exist [14], they do not describe the method completely (e.g. omit many cases) or are limited to trees obtained from the same data set [15]. Another interesting way to tree merging transforms trees into Fourier spectra and merges them by vector addition in this dual space [3]. While all cases are considered, the authors analyze in-depth only the case of binary attributes and do not report empirical results on the performance. Moreover, the method is highly complex, making it hard to implement and extend. None of the above approaches considers the merging efficiency and the size explosion problem (Sec. III-B) when trees over many fragments are merged. As such, they are not feasible for distributed learning over many sites.

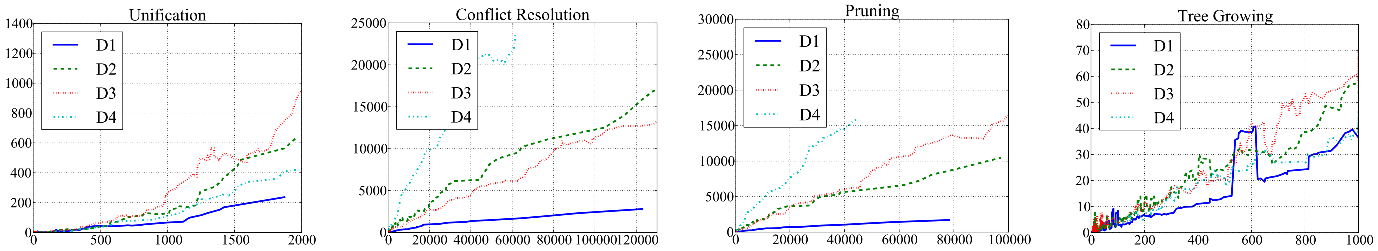


Figure 6. Running times (y-axis, in ms) of selected operations depending on the total input size (x-axis, in boxes) for TMm and datasets D1 to D4

Algorithm	**Node mem.**	**# passes**	**# sync phases**	**Pattern**
TMo	$20max_{box}(2d + \|\mathscr{L}\|)$	1	$\lceil\log_2 k\rceil + 1$ or k	cas1p / 1p
TMm	$20max_{box}(2d + \|\mathscr{L}\|)$	k	$2(\lceil\log_2 k\rceil + 1)$	gen

Table III
Communication efficiency of the algorithms

VI. Conclusions

We have presented algorithms for interpretable classification on distributed data which are based on merging decision trees represented as sets of decision regions. There are several advantages of this method: it produces interpretable intermediate models; works in resource-constrained environments; and it can be used with any tree-building algorithm applied to each data fragment. The consequence of the latter is that in combination with VFDT [13] we obtain a k-parallel "pseudo-streaming" classifier which outputs a single decision tree.

We discovered that consecutive application of the merging operation yields large number of new decision regions (which are intersections of original regions, see Table II). This problem has not been reported in prior work on tree merging [3], [14]. We solve it by ranking and pruning boxes according to various criteria which trade accuracy for communication efficiency. This leads to two variants of the overall approach: a single-pass (TMo) and a multi-pass (TMm) variant. TMm provides sufficient accuracy up to about $k = 16$ data fragments, but it requires k passes through data. The accuracy of TMo is in general worse.

Our further contribution is the analysis of communication pattern of distributed learning algorithms in context of their communication efficiency (Sec. II-B). This refines the discussion of communication efficiency in context of sufficient statistics [10].

References

[1] K. Liu and H. Kargupta, "Distributed data mining bibliography," 2008. [Online]. Available: http://www.csee.umbc.edu/~hillol/DDMBIB/

[2] R. Bekkerman, M. Bilenko, and J. Langford, Eds., *Scaling up Machine Learning: Parallel and Distributed Approaches*. Cambridge University Press, 2012.

[3] H. Kargupta and B.-H. Park, "A fourier spectrum-based approach to represent decision trees for mining data streams in mobile environments," *IEEE Trans. Knowl. Data Eng.*, vol. 16, no. 2, pp. 216–229, 2004.

[4] B. Panda, J. Herbach, S. Basu, and R. J. Bayardo, "PLANET: Massively parallel learning of tree ensembles with mapreduce," *PVLDB*, vol. 2, no. 2, pp. 1426–1437, 2009.

[5] C.-T. Chu, S. K. Kim, Y.-A. Lin, Y. Yu, G. R. Bradski, A. Y. Ng, and K. Olukotun, "Map-reduce for machine learning on multicore," in *NIPS*, 2006, pp. 281–288.

[6] D. Pyle, *Data Preparation for Data Mining*. Morgan Kaufmann, 1999.

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

[8] T. Hastie, R. Tibshirani, and J. Friedman, *Elements of Statistical Learning: Data Mining, Inference, and Prediction (2nd Ed.)*. New York: Springer Verlag, 2009.

[9] Y. Ben-Haim and E. Tom-Tov, "A streaming parallel decision tree algorithm," *Journal of Machine Learning Research*, vol. 11, pp. 849–872, 2010.

[10] D. Caragea, A. Silvescu, and V. Honavar, "A framework for learning from distributed data using sufficient statistics and its application to learning decision trees," *Int. J. Hybrid Intell. Syst.*, vol. 1, no. 2, pp. 80–89, 2004.

[11] J. C. Shafer, R. Agrawal, and M. Mehta, "SPRINT: A scalable parallel classifier for data mining," in *VLDB'96*, 3–6 Sep. 1996, pp. 544–555.

[12] F. J. Provost and V. Kolluri, "A survey of methods for scaling up inductive algorithms," *Data Min. Knowl. Discov.*, vol. 3, no. 2, pp. 131–169, 1999.

[13] P. Domingos and G. Hulten, "Mining high-speed data streams," in *Proc. 6th ACM SIGKDD*, 2000, pp. 71–80.

[14] L. O. Hall, N. Chawla, and K. W. Bowyer, "Decision tree learning on very large data sets," in *Proc. IEEE Int Systems, Man, and Cybernetics Conf*, vol. 3, 1998, pp. 2579–2584.

[15] G. J. Williams, "Inducing and combining decision structures for expert systems," Ph.D. dissertation, Australian National University, Canberra, Australia, 1991.

[16] O. Dekel, S. Shalev-Shwartz, and Y. Singer, "The forgetron: A kernel-based perceptron on a budget," *SIAM J. Comput.*, vol. 37, no. 5, pp. 1342–1372, 2008.

[17] P. Rai, H. D. III, and S. Venkatasubramanian, "Streamed learning: One-pass SVMs," in *IJCAI 2009*, 2009, pp. 1211–1216.

[18] J.-H. Böse, A. Andrzejak, and M. Högqvist, "Beyond online aggregation: Parallel and incremental data mining with online mapreduce," in *ACM MDAC*, Raleigh, NC, USA, 2010.

[19] H. P. Graf, E. Cosatto, L. Bottou, I. Durdanovic, and V. Vapnik, "Parallel support vector machines: The cascade SVM," in *NIPS*, 2004.

[20] A. Frank and A. Asuncion, "UCI machine learning repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[21] I. H. Witten and E. Frank, *Data Mining: Practical machine learning tools and techniques*, 2nd ed. Morgan Kaufmann, San Francisco, 2005.