Scalable Learning of Tree-Based Models on Sparsely Representable Data

Fares Hedayati, upwork, fares19@upwork.com
Arnauld Joly, University of Liege, a.joly@ulg.ac.be Panagiotis
Papadimitriou, upwork, papadimitriou@upwork.com

Abstract—Many machine learning tasks such as text annotation usually require training over very big datasets, e.g., millions of web documents, that can be represented in a sparse input space. State-of-the-art tree-based ensemble algorithms cannot scale to such datasets, since they include operations whose running time is a function of the input space size rather than a function of the non-zero input elements. In this paper, we propose an efficient splitting algorithm to leverage input sparsity within decision tree methods. Our algorithm improves training time over sparse datasets by more than two orders of magnitude and it has been incorporated in the current version of *scikit-learn*[1], the most popular open source Python machine learning library.

Keywords—achine learning, classification trees, regression trees, sparse data, scalabilityachine learning, classification trees, regression trees, sparse data, scalabilitym

I. Introduction

Decision trees provide simple predictive models that are widely used in machine learning and data mining for both classification and regression problems. Each decision tree leaf represents a particular label in a classification problem (or a particular scalar value in a regression problem) and the internal nodes of the tree represent conjunctions of features that lead into the tree leaves. Decision trees are popular not only because they are easy to interpret, but also because they provide the basic blocks of more sophisticated and powerful ensemble methods such as Adaboost [8], Random Forests [3] or Gradient Tree Boosting [9]. Such methods are usually the best performers in industry for both classification and regression tasks.

Learning a decision tree over a training set is a recursive top-down procedure. In each step the procedure finds a splitting rule that partitions the underline space into two segments so that it maximizes some notion of information gain, i.e., it reduces the variability of the labels in the underlying (two) child nodes compared to the parent node. The search for the optimal splitting rule makes the running time of each partition decision linear to the number of features and possible values in the training set.

Such a cost make the use of tree-based methods prohibitive for high dimensional supervised learning problems such as text or image annotation that are common in many research and industrial problems. Such problems require defining a mapping between the raw input space and a target vector space. For example, in text classification a document (raw input space) is usually mapped to a vector whose dimensions correspond to all of the possible words in a dictionary and the values of the vector elements are determined by the frequency of the words in the document. Although such vectors have many dimensions, they are often sparsely representable. For instance, the number of unique words associated to a text document is actually small compared to the number of words of a given language.

Many models, such as logistic regression or SVM, could directly benefit from the input sparsity by formulating the entire algorithm through a set of dot products that can be computed fast with appropriate sparse vector representations. However, this is not possible for tree based methods, since the tree learning procedure cannot be expressed with dot products. Consequently, most machine learning packages do not support sparse input for tree-based methods, are restricted to decision stumps (decision tree with only one internal node) or have a sub-optimal implementation through the simulation of a random access memory as in the dense case. The only solution is often to densify the input space which leads first to severe memory constraints and then to slow training time.

In this paper, we present an efficient splitting procedure tailored for numerical sparse input data in compressed sparse column format, a sparse matrix format. For a given subset of samples, we are able to efficiently extract non zero values for a given feature of this subset of samples. Knowing which elements are nonzero allows large speedup. It decreases sorting time of samples in the current node along features which is an essential component in all tree-based models. Moreover it reduces the set of possible splits to evaluate at each node. We also want to highlight that the contribution of this paper have been proposed for and merged in the *scikit-learn* [5], [12] open source package. This will benefit the machine learning community.

The rest of this paper is organized as follows: Section II introduces decision tree splitting algorithm and sparse matrix formats; Section III describes the proposed splitting algorithm for sparse input data; Section IV summarizes the experiments carried out with the proposed algorithm comparing them to the traditional decision tree training algorithm and Section VI concludes and describes further perspectives.

II. Decision Trees With Dense Input Data

A. Notation

We denote by $\mathcal{X}$ an input space and by $\mathcal{Y}$ the output space. Without loss of generality, we suppose that $\mathcal{X} = \mathcal{R}^m$ where m denotes the number of features and $\mathcal{Y} = \mathcal{R}$ in case of regression or $\mathcal{Y} = \{0, 1, \cdots, K-1\}$ in case of a K-$class$ classification. Learning samples are represented by a pair of matrices $(X, Y) \subseteq (\mathcal{X}, \mathcal{Y})_{i=0}^{n-1}$ with n rows, where each row corresponds to a sample and each column to a feature or an output variable. In the process of decision tree induction we have to move samples around, instead of moving the actual rows of (X, Y) we move their indices. For this purpose we introduce $\mathcal{L} = [0, 1, \cdots, n-1]$ which is the list of row indices of (X, Y). As we will see in the next section, impurity measure plays a crucial role in induction of decision trees. An impurity measure of a set of samples $\mathcal{L}$ measures the heterogeneity of the target variable of the samples. The variance of the target variable is used in regression, and Gini and cross-entropy are usually used in tree-based classification algorithms. Given $\mathcal{L}$, we let $\hat{p}_k$ be the empirical probability of class k in the dataset. The Gini impurity measure is defined as:

$$I(\mathcal{L}) = \sum_{k=0}^{K-1} \hat{p}_k \left(1 - \hat{p}_k\right), \tag{1}$$

[1] http://scikit-learn.org

and the cross-entropy impurity measure is defined as:

$$I(\mathcal{L}) = \sum_{k=0}^{K-1} \hat{p}_k \log\left(\hat{p}_k\right) \tag{2}$$

Note that the impurity measure is maximized in case of uniform distribution of the target variable which makes $\mathcal{L}$ most heterogenous, i.e. $\hat{p}_k = \frac{1}{K}$. Moreover the impurity measure is minimized when the empirical probability is concentrated on one class while other classes have $\hat{p}_k = 0$.

B. Training

A decision tree [4] is built by recursively splitting the sample set into two nodes by maximizing the average reduction of an impurity measure:

$$\Delta I(s, \mathcal{L}) = \left(I(\mathcal{L}) - \left(\frac{|\mathcal{L}_r|}{|\mathcal{L}|} I(\mathcal{L}_r) + \frac{|\mathcal{L}_l|}{|\mathcal{L}|} I(\mathcal{L}_l)\right)\right) \tag{3}$$

where $s = (j,\, \theta)$ is a splitting rule composed of a feature index j and a threshold value θ which divides the sample set into $\mathcal{L}_l$ and $\mathcal{L}_r$ where

$$\mathcal{L}_l = \{i \in \mathcal{L} \,|\, X_{i,j} < \theta\}, \tag{4}$$

and

$$\mathcal{L}_r = \{i \in \mathcal{L} \,|\, X_{i,j} \geq \theta\} \tag{5}$$

This recursive procedure is repeated until a stopping condition is met, e.g. a maximal depth is reached or there are too few samples to split. Those stopping criteria act as regularization parameters. Leaves are labeled by the output mean in regression or by the class frequencies in classification with reaching training samples. The recursive induction of the decision decision is described by Algorithm 1 and the search for the best split is described by Algorithm 2. Note that sorting samples (Line 5) in a node along different features is at the core of Algorithm 2; it speeds up computation of the impurity measure for all possible splitting rules in an incremental manner. In Algorithm 1 below a stack is initially setup with a single element namely a pair of a root t_0 and the entire set of samples $\mathcal{L}$ (Line 5). Line 7 pops the next set of samples from the stack ($\mathcal{L}_p$) and split it into two nodes, the left node $\mathcal{L}_l$ and and the right node $\mathcal{L}_r$, based on the best splitting rule obtained in Line 11. Note that this splitting is carried out recursively until the stopping criterion are met (Line 8,9) in which case the current node becomes a leaf.

In the context of ensemble, trees are further randomized by searching for the best split among k features at each node and also might be induced on a bootstrap copy of the samples. The tree can be grown alternatively in a best-first search manner by replacing the stack of Algorithm 1 by a priority queue where priority is defined by expected impurity reduction.

Algorithm 1: Build a decision tree

1: **function** INDUCEDECISIONTREE($\mathcal{L}$, X, Y)
2: Initialize a tree structure τ with root node t_0
3: Initialize an empty stack $stack$
4: Initialize a sample set $\mathcal{L} = \{0, \ldots, n-1\}$
5: $stack$.PUSH((t_0, $\mathcal{L}$))
6: **while** $stack$ is not empty **do**
7: t_p, $\mathcal{L}_p$ = $stack$.POP()
8: **if** t_p satisfies stopping criterion **then**
9: Make t_p a leaf node using $\mathcal{L}_p$ and Y.
10: **else**
11: Find a splitting rule s^* which maximizes impurity reduction among possible splitting rules:

$$s^* = FindBestSplit(\mathcal{L}_p)$$

12: Partition $\mathcal{L}_p$ into $\mathcal{L}_r$ and $\mathcal{L}_l$ given s^*.
13: Create two empty nodes t_l and t_r
14: Set t_l and t_r the left and right children of t_p.
15: $stack$.PUSH((t_r, $\mathcal{L}_r$))
16: $stack$.PUSH((t_l, $\mathcal{L}_l$))
17: **end if**
18: **end while**
19: **return** τ
20: **end function**

C. The Optimal Split

Given a set of samples $\mathcal{L}_p$ the best splitting rule is computed by finding for each feature a threshold that can reduce the average impurity measure (Equation 3) the most and at the end picking the feature with its corresponding optimal threshold that has the best reduction among all features. For a given splitting rule $s = (j, \theta)$, the whole sample set needs to be traversed once for determining where each sample row belongs to, to the left (Equation 4) or to the right node (Equation 5). To avoid multiple traversals, for a given feature j, the samples are sorted along the feature in an ascending order, and the impurity reduction is computed in an incremental way from the lowest value of feature j to the highest value, treating each feature value as a threshold. As an example consider computing the impurity measure Gini (Equation 1) for all values of feature j. We can easily update Equation 3 by keeping track of left node $\mathcal{L}_l$ and right node $\mathcal{L}_r$ class frequencies and updating them as we move from one threshold to the next, e.g. from $\theta = X_{i,j}$ to $\theta = X_{i+1,j}$. In other words $\Delta I\left(s = (j, X_{i+1,j}), \mathcal{L}\right)$ can be easily computed by updating left node and right node class frequencies for $\theta = X_{i,j}$, and on top of them the empirical probabilities $\hat{p}_k$ and $|\mathcal{L}_l|$ and $|\mathcal{L}_r|$ of Equation 3.

In Algorithm 2 below, Line 3 iterates over all features ranging from 0 to m creating a dataset $\mathcal{F}_j$ for each one of them. $\mathcal{F}_j$ is basically a collection of all values in column j of the current node $\mathcal{L}_p$. The algorithm computes the impurity for each of the possible values in $\mathcal{F}_j$ in Lines 6 to 8. Line 10 keeps track of the best impurity reduction, and the corresponding rule (referred to s^* in Line 11) which is a combination of a feature index j and a threshold θ. s^* is the returned value of the algorithm and the best splitting rule.

Algorithm 2: Search for the best split

1: **function** FINDBESTSPLIT(X, $\mathcal{L}_p$)
2: best $= -\infty$
3: **for** $j \in \{0, \ldots, m-1\}$ **do**
4: Extract feature values reaching the node

$$\mathcal{F}_j = \{X_{i,j}, \forall i \in \mathcal{L}_p\}.$$

5: Sort $\mathcal{L}_p$ and $\mathcal{F}_j$ by increasing values of $\mathcal{F}_j$.
6: **for** θ in F_j **do**
7: s = (j, θ)
8: Evaluate impurity reduction s

$$\text{score} = \Delta I(s, \mathcal{L}_p).$$

9: **if** score $>$ best **then**
10: best $=$ score
11: $s^* = s$
12: **end if**
13: **end for**
14: **end for**
15: **return** s^*
16: **end function**

III. Decision Trees With Sparse Input Data

A. Sparse matrix format

For memory efficiency and taking advantage of sparsity we use a data structure called compressed sparse column (csc) matrix. It is a general format to represent compactly sparse matrices using three arrays: a $data$ array that stores the value of each nonzero element for each column, an $indices$ array that stores the row indices of

nonzero elements for each column and an $indptr$ array which stores the beginning and end of each column in the $data$ and the $indices$ arrays. The row indices of non-zero data points of column c are stored in $indices[indptr[c]{:}indptr[c+1]]$ with their corresponding non-zero values stored in $data[indptr[c]{:}indptr[c+1]]$. Note that $indptr[c]{:}indptr[c+1]$ corresponds to numbers from $indptr[c]$ (inclusive) to $indptr[c+1]$ (exclusive).
For instance, this 3×5 matrix

$$\begin{bmatrix} 1 & 0 & 0 & 4 & 0 \\ 0 & 0 & 0 & 5 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

is represented by the following csc matrix with arrays

$$\begin{aligned} data &= \begin{bmatrix} 1 & 4 & 5 \end{bmatrix}, \\ indices &= \begin{bmatrix} 0 & 0 & 1 \end{bmatrix}, \\ inptr &= \begin{bmatrix} 0 & 1 & 1 & 1 & 3 & 3 \end{bmatrix}. \end{aligned}$$

In the above example the fourth column corresponds to the row indices $indices[indptr[3]{:}indptr[3+1]] = indices[1{:}3] = [0, 1]$ with data points $indices[indptr[3]{:}indptr[3+1]] = data[1{:}3] = [4, 5]$, meaning only row 0 and row 1 in column 3 have non-values with values 4 and 5 respectively. The main advantages of csc matrices are to allow fast column indexing, efficient arithmetic and matrix operations. However, row indexing is slow. Note that a similar row-based sparse matrix called compressed sparse row format also exists and works under similar principles. Note that both of these sparse matrices are available in *scikit-learn*[2].

In order to grow decision trees on sparse input matrix, we have to require a sparse matrix format with efficient row indexing as the tree works with subset of the samples, and also efficient column indexing as features are randomly sampled at each node. Furthermore, we hope to speed up the overall algorithm by taking into account the input space sparsity. Compressed sparse column matrix already satisfies the fast column indexing requirement. We are going to show how to efficiently exploit the data structure as to have a fast row indexing and use the proposed approach to speed up the overall algorithm on sparse data.

At the core of our proposed method is a fast sorting algorithm (a substitute for Line 5 in Algorithm 2) that works with non-zero values of a feature, sorts the positive and negative parts separately, and rearranges the sample set accordingly.

B. Nonzero Values of $\mathcal{L}_p$

Given the sparse matrix format, the main issue is to efficiently perform the extraction of the sample values reaching the node (the line 4 of Algorithm 2). Note that this is the only operation which requires interaction with the input matrix data. Otherwise said for a given feature j, one has to be able to perform the intersection between the sample set $\mathcal{L}_p$ which has reached the node and the row indices in $\mathcal{L}$ where feature j is non-zero or mathematically with the set $indices[indptr[j]{:}indptr[j+1]]$. In short, we need to compute the intersection of the set of row indices of the training data in the current node and the set of row indices of the whole training data where feature j is non-zero. This intersection is needed to generate a set of possible splitting rules for feature j.

We let m_j be the number of data points where their feature j is non-zero:

$$m_j = indptr[j+1] - indptr[j] \tag{6}$$

If we assume that the $indices$ of the input csc matrix array are sorted per column, then standard intersection algorithms have the following time complexity:

1) in $O(|\mathcal{L}_p| \log m_j)$ by performing $|\mathcal{L}_p|$ binary search on the sorted m_j nonzero elements;
2) in $O(|\mathcal{L}_p| \log |\mathcal{L}_p| + m_j \log |\mathcal{L}_p|)$ by sorting the sample set $\mathcal{L}_p$ and performing m_j binary search on $\mathcal{L}_p$;
3) in $O(|\mathcal{L}_p| \log |\mathcal{L}_p| + m_j + |\mathcal{L}_p|)$ by sorting the sample set $\mathcal{L}_p$ and retrieving the intersection by iterating over both arrays;
4) in $O(m_j)$ by maintaining a hash table of $\mathcal{L}_p$ and then checking if elements of $indices[indptr[j]{:}indptr[j+1]]$ are contained in the hash table.

As explained below, we will be using a hybrid solution composed of a variation of approach (4) and approach (1). In the context of decision tree induction, the intersection operation will be repeated for each feature and for various sample sets $\mathcal{L}_p$. Taking this into account, it's possible to improve approach (4).

Before going into details of the hybrid approach, understanding the manipulation of data in $\mathcal{L}$ is of essential importance. $\mathcal{L}$ is the list of row indices; if there are 10 samples in the set then $\mathcal{L} = [0, 1, \cdots, 9]$. During the tree growth in Algorithm 1 when $\mathcal{L}$ is split into two sets $\mathcal{L}_l = [9, 1, 5, 3]$ and $\mathcal{L}_r = [2, 7, 6, 4, 8, 0]$, the row indices will be shuffled in $\mathcal{L}$ in such a way that the indices in $\mathcal{L}_l$ are pushed to the left part of $\mathcal{L}$ and the indices in $\mathcal{L}_r$ are pushed to the right side of $\mathcal{L}$. This means that $\mathcal{L}$ will be shuffled to [9, 1, 5, 3, 2, 7, 6, 4, 8, 0]. Each node is then represented by a slice $[start : end]$, a chunk in the original list $\mathcal{L}$. In the above example, $\mathcal{L}_l$ will represented by the slice $[0 : 4]$ and $\mathcal{L}_r$ will be represented by the slice $[4 : 10]$. Now if the right node $\mathcal{L}_r$ is further split into a left node $\mathcal{L}_{rl} = [6, 0]$ and a right node $\mathcal{L}_{rr} = [2, 7, 4, 8]$ then $\mathcal{L}[4 : 10]$ is further rearranged to reflect this change. $\mathcal{L}$ becomes [9, 1, 5, 3, 2, 7, 4, 8, 6, 0], $\mathcal{L}_{rl}$ is represented by $[4 : 9]$, and $\mathcal{L}_{rr}$ is represented by $[9 : 10]$.

The main idea of approach (4) is to maintain during the tree growth a *mapping*, as shown in Figure 1, between the $indices$ array of the csc matrix, and the position of the related samples in the sample set array $\mathcal{L}$. Row indices gets shuffled in $\mathcal{L}$ all the time. For a row index i *mapping* keeps track of its current position. In other words *mapping* keeps track of where sample i is in $\mathcal{L}$. For example row index 6 (or the 7th sample in the original dataset) was initially at position 6 in $\mathcal{L}$ in the example above (note that $\mathcal{L}$ was initially [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]) which translates to $mapping[6] = 6$. But after a few splits $\mathcal{L}$ was rearranged to [9, 1, 5, 3, 2, 7, 4, 8, 6, 0] which in turn have changed $mapping[6]$ to be 8. More generally during training we always keep the following invariance in place for each **pos** in $\mathcal{L}$:

$$mapping[\mathcal{L}[\mathbf{pos}]] = \mathbf{pos}. \tag{7}$$

The main usage of $mapping$ is for efficient computation of the intersection of the current node $\mathcal{L}_p$ and all row indices that have non-zero values for a given feature j, i.e. the set $indices[indptr[j]{:}ndptr[j+1]]$. This intersection is basically the subset of the current node $\mathcal{L}_p$ that are non-zero at their feature j. Let's assume that the current node $\mathcal{L}_p$ is represented by $[start : end]$ as shown in Figure 1. Then for a row index i in $indices[indptr[j]{:}ndptr[j+1]]$, if $mapping[i]$ is between $start$ and end then the sample should be in $\mathcal{L}_p$. Thus, it's possible to check in $O(1)$ if a sample with non-zero value at a given feature is in the current node $\mathcal{L}_p$ or not. Maintaining the $mapping$ for a given position **pos** is done in $O(1)$ by setting $mapping[\mathcal{L}[\mathbf{pos}]]$ to **pos**. Thus we deduce that performing the intersection between the $indices[indptr[j]{:}ndptr[j+1]]$ array and $\mathcal{L}_p$ can be done in $O(m_j)$ ($m_j = indptr[j+1] - indptr[j]$ is the number of non-zero values of feature j in $\mathcal{L}$).

With the application of the mapping intersection algorithm, we can speed up the sorting operation and splitting rule evaluation of Algorithm 2 by working separately on positive and negative values. Furthermore, it's also possible to partition a sample set $\mathcal{L}_p$ into two partition $\mathcal{L}_r$ and $\mathcal{L}_r$ (line 12 of Algorithm 1) given a split on feature j in $O(m_j)$ instead of $O(n)$. For more details of this

[2]http://docs.scipy.org/doc/scipy-0.16.0/reference/generated/scipy.sparse.csc_matrix.html

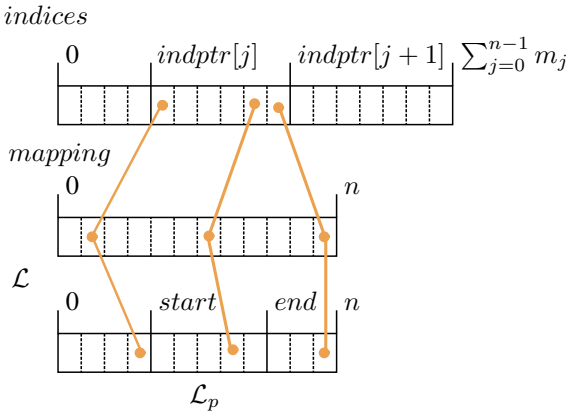

Fig. 1. The array *mapping* allows to efficiently compute the intersection between the *indices* array of the csc matrix and a sample set $\mathcal{L}_p$

algorithm refer to Algorithm 4.

In practice, the number of nonzero elements m_j of feature j could be a lot bigger than the size of a sample set $\mathcal{L}_p$. This is likely to happen near the leaf nodes. Whenever the tree is fully developed, there are only a few samples reaching those nodes. For optimal performance, one can use a hybrid intersection approach which combines the previously developed mapping intersection approach with approach (1) based on binary search. whenever $|\mathcal{L}_p| \ll m_j$, the binary approach will be faster. In this case for each row index i in $\mathcal{L}_p$, a binary search will be performed on $indices[indptr[j]{:}indptr[j+1]]$. This takes $O(|\mathcal{L}_p| \log m_j)$, as each binary search takes at most $\log m_j$ and there are $|\mathcal{L}_p|$ elements that need to be searched for in $indices[indptr[j]{:}indptr[j+1]]$. For more details of this algorithm refer to Algorithm 5.

The hybrid algorithm uses the binary search of Algorithm 5 when the cost of approach (1) is smaller than $O(m_j)$. More specifically, the hybrid approach is controlled by the following rule:

$$|\mathcal{L}_p| \times \log(m_j) < 0.1 \times m_j. \tag{8}$$

Algorithm 4 is used whenever Equation 8 is true and Algorithm 5 is used otherwise. This is exactly Algorithm 3. Note that 0.1 is optained empirically.

During the tree growth, one could remember which features are constant for a subset of the samples $\mathcal{L}_p$ and a given node t_p. For all descendant of node t_p, this will avoid the overhead of searching for a split where none exists for those features.

Finally note that for testing, the sparse data is flattened for efficient random memory access.

C. The Optimal Split

The optimal split in sparsely representable data is a hybrid solution of two sorting algorithms that are well-suited for sparse csc matrices. For a given feature j the positive and negative values of the current node $\mathcal{L}_p$ are extracted and sorted in two different lists (F_p and N_p of Algorithm 3 Lines 7 and 10) with their corresponding indices rearranged in $\mathcal{L}$. To choose the correct sorting algorithm the criterion in Equation 8 is checked to see if approach (1) is more suitable or approach (4) (See Line 6 in Algorithm 3 below). Based on this criterion either approach (1) in Line 8 (SORT_BSEARCH) is used or approach (4) in Line 11 (SORT_MAPPING). Note that since there are significantly less nonzero values in sparse data, sorting the positive and negative values separately will be very cost effective. After sorting, the best threshold is sought separately in positive values and negative values. Lines 13-20 finds the best split among the negatives values and Lines 21-28 finds the best split among negative values. The final best split s^* is the best among the two cases.

Algorithm 3: Find the best split

```
1: function FindBestSparseSplit(X, L_p)
2:     Let [start : end] be such that L_p = L[start : end]
3:     best = −∞
4:     for j ∈ {0, ..., m − 1} do
5:         m_j = indptr[j + 1] − indptr[j]
6:         if |L_p| × log(m_j) < 0.1 × m_j then
7:             F_p, F_n =
8:             SORT_BSEARCH(X, start, end, mapping, j)
9:         else
10:            F_p, F_n =
11:            SORT_MAPPING(X, start, end, mapping, j)
12:        end if
13:        for θ in F_n do
14:            s = (j, θ)
15:            score = ΔI(s, L[start : end]).
16:            if score > best then
17:                best = score
18:                s* = s
19:            end if
20:        end for
21:        for θ in F_p do
22:            s = (j, θ)
23:            score = ΔI(s, L[start : end]).
24:            if score > best then
25:                best = score
26:                s* = s
27:            end if
28:        end for
29:    end for
30:    return s*
31: end function
```

The SORT_MAPPING in Algorithm 4 below extracts samples with positive or negative values of feature j at the current node $\mathcal{L}_p$ represented by the slice $[start : end]$ with the aid of *mapping*. It basically looks at each row index *index* in $indices[indptr[j]{:}indptr[j+1]]$ and checks if that *index* is mapped to a number between *start* and *end* in *mapping* (Line 12). If this is the case then that row index is part of the current node and its value is appended to F_p if it is positive (Line 15) or to F_n if it is negative (Line 19). Note that after appending the non-zero value to the appropriate list (F_p or F_n deepening on the sign of the value) the function SWAP is called (Lines 17 and 20). SWAP (in Algorithm 6) makes sure that the row indices with positive values are pushed to the right of $\mathcal{L}[start : end]$ and the row indices with negative values are pushed to the left of $\mathcal{L}[start : end]$. This is done by the help of auxiliary variables $start_p$ and end_n. $start_p$ is the left most position of row indices $\mathcal{L}[start : end]$ in $\mathcal{L}$ that are positive in the given feature j and end_n is the right most position of row indices that are negative in feature j. Initially $start_p$ points to *end* (the end of $\mathcal{L}[start : end]$) and end_n points to *start* (the being of $\mathcal{L}[start : end]$). As soon as a row index with non-zero value from $indices[indprt[j] : indptr[j+1]]$ is found in $\mathcal{L}[start : end]$, then if its value is negative it will be swapped with whatever element is at end_n and end_n shifts one position to the right (Line 21) and if it is positive $start_p$ shifts one element to the left (Line 16) and the row index will be swapped with whatever element is at $start_p$. This guarantees that row indices with positive values are pushed to the end of $\mathcal{L}[start : end]$ and row indices with negative values to the beginning of $\mathcal{L}[start : end]$.

During the rearrangement of row indices in $\mathcal{L}[start : end]$, *mapping* is kept up-to-date. Finally at Line 24 all row indices with negative values are to the left of $\mathcal{L}[start : end]$ and all the row indices with positive values are to the right of it and the row indices in between correspond to samples with value zero (of feature j). Lines 25 and 26 sorts the positive and negative parts of $\mathcal{L}[start : end]$

based on their values in F_p and F_n such that the row indices in $\mathcal{L}[start : end]$ are sorted based on feature j in an ascending order (for details on the sorting algorithm refer to SORT in Algorithm 6).

Algorithm 4: Sorts $X_{\mathcal{L}[start:end]}$ along the jth feature by rearranging their indices in $\mathcal{L}[start : end]$ via $mapping$, and returns the sorted positive and negative values separately.

```
1: function SORT_MAPPING(X, start, end, mapping, j)
2:     F_p = []
3:     F_n = []
4:     start_p = end
5:     end_n = start
6:     incides = X.indices
7:     indptr = X.indptr
8:     data = X.data
9:     for k ∈ [indptr[j]:indptr[j + 1]] do
10:        index = indices[k]
11:        value = data[k]
12:        if start ≤ mapping[index] < end then
13:            h = mapping[index]
14:            if value > 0 then
15:                F_p.APPEND(value)
16:                start_p −= 1
17:                SWAP(L, h, start_p, mapping)
18:            else
19:                F_n.APPEND(value)
20:                SWAP(L, h, end_n, mapping)
21:                end_n += 1
22:            end if
23:        end if
24:    end for
25:    SORT(F_n, L, start, end_n)
26:    SORT(F_p, L, start_p, end)
27:    return F_p, F_n
28: end function
```

The SORT_BSEARCH in Algorithm 5 below has the same effect as SORT_MAPPING. The only different between this function and SORT_MAPPING is that instead of iterating over each row index $index$ in $indices[indptr[j]:indptr[j + 1]]$ and checking whether it is in $\mathcal{L}[start : end]$ or not, the algorithm iterates the other way around. It iterates over each row index $index$ in $\mathcal{L}[start : end]$ and checks whether it is in $indices[indptr[j]:indptr[j+1]]$ or not by performing a binary search (Line 14). If the binary search found the element the same procedure is carried as in SORT_MAPPING. The positive and negative values are appended to F_p and F_n respectively and the row indices are shuffled around in $\mathcal{L}[start : end]$ by SWAP in a way that the row indices with positive values of feature j are pushed to the right of $\mathcal{L}[start : end]$ and row indices with negative values are pushed to the left of $\mathcal{L}[start : end]$. Finally as in SORT_MAPPING, the left and right parts of $\mathcal{L}[start : end]$ that now contains row indices with negative and positive values respectively are sorted separately to make $\mathcal{L}[start : end]$ entirely sorted in an ascending order of the values of feature j.

Algorithm 5: Sorts $X_{\mathcal{L}[start:end]}$ along the jth feature by rearranging their indices in $\mathcal{L}[start : end]$, via a binary search, and returns the sorted positive and negative values separately.

```
1: function SORT_BSEARCH(X, start, end, mapping, j)
2:     F_p = []
3:     F_n = []
4:     start_p = end
5:     end_n = start
6:     incides = X.indices
7:     indptr = X.indptr
8:     data = X.data
9:     indices_j = indices[indptr[j] : indptr[j + 1]]
10:    data_j = data[indptr[j] : indptr[j + 1]]
11:    L = SORT(L, start, end)
12:    for h ∈ [start : end] do
13:        index = L[h]
14:        i = BINARYSEARCH(index, indices_j)
15:        Returns the position of index in indices_j,
16:        and -1 if it is not found.
17:        if i ≠ −1 then
18:            if data_j[i] > 0 then
19:                end_p −= 1
20:                F_p.APPEND(value)
21:                SWAP(L, h, start_p, mapping)
22:            else
23:                F_n.APPEND(value)
24:                SWAP(L, h, end_n, mapping)
25:                end_n += 1
26:            end if
27:        end if
28:    end for
29:    SORT(F_n, L, start, end_n)
30:    SORT(F_p, L, start_p, end)
31:    return F_p, F_n
32: end function
```

Algorithm 6: Auxiliary functions

```
1: function SWAP(L, p_1, p_2, mapping)
2:     L[p_1], L[p_2] = L[p_2], L[p_1]
3:     mapping[L[p_1]] = p_1
4:     mapping[L[p_2]] = p_2
5: end function
1: function SORT(F, L, p_1, p_2, mapping)
2:     Sort F and L[p_1 : p_2] by increasing values of F
3:     for p ∈ [p_1 : p_2] do
4:         mapping[L[p]] = p
5:     end for
6: end function
```

IV. EXPERIMENTS

In this section we will compare the training time of the proposed tree training algorithm with the traditional training algorithm. Note that for each of the experiments below we verified that the decision trees returned by two algorithms are identical. In the first part of the experiments we have tested our proposed algorithm on some real data and compared the running time to the traditional tree training algorithm as it was used in *scikit-learn* before our change was incorporated in. In the second part of the experiment we tested our algorithm on some synthetic data with varying density to show how our algorithm can benefit from sparsity.

A. Density

The density of a matrix X is the percentage of its nonzero entries and the sparsity is the percentage of its zero entries. As we will see below training of decision trees on datasets that are quite sparse (with density below 0.01) is significantly faster using our algorithm. Furthermore in the *Synthetic Data* section we compare running time of decision trees with sparse csc matrices and dense arrays as a function of density. Note that in all the experiments below we made sure that both algorithms (induction of decision trees with a sparse csc matrix and the traditional induction of decision trees with dense arrays) lead exactly to the same model structure and have the same generalization performance.

B. Real Data

In this section we will assess the computational performance of the decision tree algorithm on datasets composed dominantly of categorical or textual features, which makes them very sparse, and on datasets that are more dense. We will compare the learning time

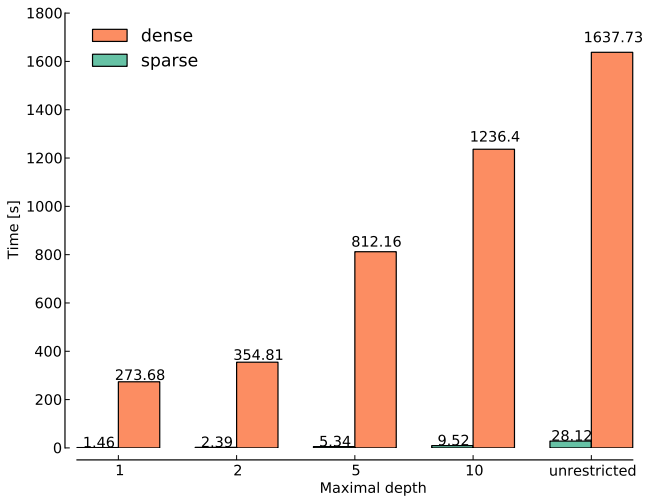

Fig. 2. Leveraging the input sparsity significantly speed up decisions tree induction both with shallow and deep trees on the *20 Newsgroups* dataset. Note that the dataset is very sparse (density = 0.001).

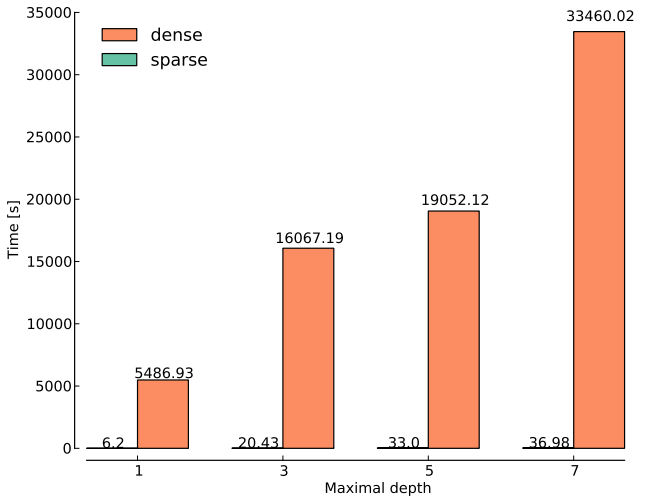

Fig. 3. Leveraging the input sparsity significantly speed up decisions tree induction both with shallow and deep trees on the *cup* dataset. Note that the dataset is quite sparse (density = 0.014).

between the decision tree learnt using a sparse csc matrix and a dense array. The first two experiments are on very sparse datasets, namely on the *20 Newsgroups* dataset [11] and on the *KDD cup 1999* dataset[2]. The *20 Newsgroups* dataset consists of $n = 11314$ document on 20 topics. Each text document was transformed into sparse tf-idf vectors of size $m = 130107$ with a density of 0.001.
In tree-based ensemble methods, decision trees are either shallow trees as used in boosting methods or deep trees as in random forest ensemble methods. Figure 2 shows that properly taking into account the sparsity of the input space allows to speed up the learning from 58 times for a fully grown tree up to 188 times for a decision stump. Furthermore, both algorithm leads exactly to the same decision tree structure and have the same generalization performance. The next dataset with low density is the *KDD cup 1999* dataset which consists of $n = 96367$ instances. The dataset contains both numerical and categorical data. The feature size of the dataset with categorical features transformed to binary features is $m = 20025$ with an average density of 0.014. Unlike the previous dataset, the task at hand for this dataset is regression. We will compare the learning

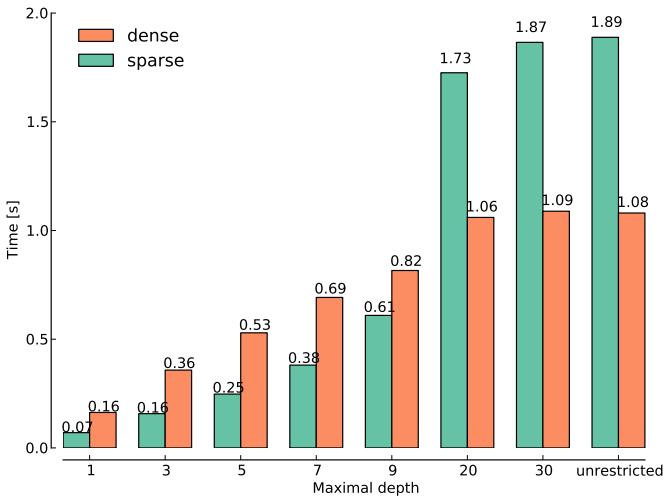

Fig. 4. Leveraging the input sparsity does not speed up training of deep trees on the *adult* dataset. Note that the dataset is quite dense (density = 0.11).

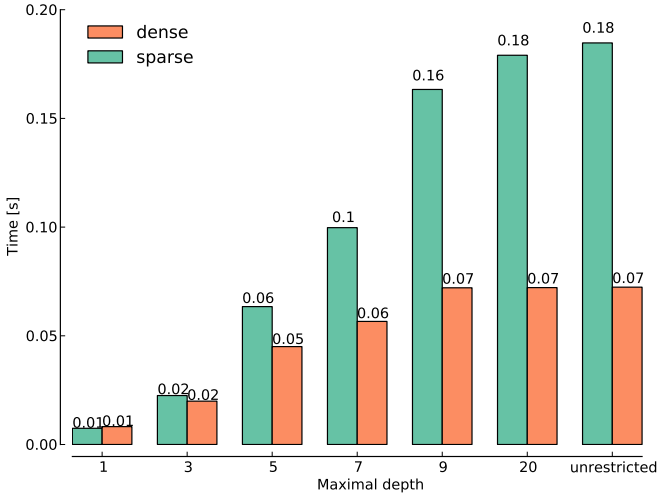

Fig. 5. Leveraging the input sparsity does not speed up training trees on the *tic* dataset. Note that the dataset is very dense (density = 0.44).

time between the decision tree regressor learnt using a sparse csc matrix and a dense array. As Figure 3 shows, taking into account the sparsity of the input space allows to speed up the learning by a factor between 800 and 900. Furthermore, both algorithm lead exactly to the same decision tree regressor structure and have the same generalization performance.

Our dense datasets are the *Adult* dataset[1] with a density of 0.12, $n = 32561$ instances and $m = 145$ features, and the *TIC* dataset[1] with a density of 0.44 and $n = 4000$ instances and $m = 85$ features. As Figure 4 shows, for shallower trees sparse trees are approximately learnt twice faster than their dense counterparts, but for fully grown trees it is the opposite. Finally as Figure 5 shows decision trees are learnt faster by dense input data rather than by input data sparsely represented.

C. Synthetic Data

As a another experiment, we generated random binary classification tasks with $n = 100000$ samples and $m = 1000$ features. The input matrices are sparse random matrices whose nonzero elements

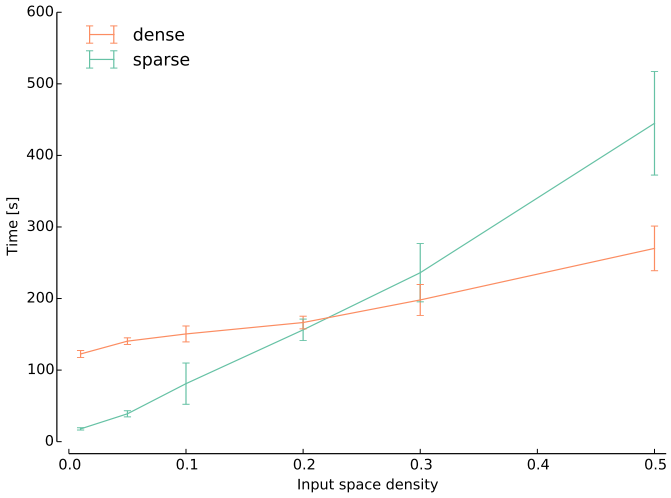

Fig. 6. Significant speed up is achieved by the sparsity-aware decision tree algorithm whenever the density is below 0.2 (or sparsity over 0.8).

are drawn uniformly in $[0, 1)$. Their density are ranging from 0.01 to 0.5. Each point is averaged over 20 experiments and the maximal depth of the decision tree is restricted to 20. As illustrated on Figure 6, the sparsity aware decision tree induction algorithm exploits the sparsity structure to be trained faster than its dense counterpart. However whenever the input space density is over 0.2, the extraction of the nonzero values in the sparse csc matrix becomes expensive. This suggests that the sparse decision tree induction algorithm is particularly suited for sparsely representable data such as text documents.

V. Related Work

The main contribution of this paper is on scaling up tree-based models in presence of large and sparsely representable data. Many researchers in the field of machine learning and statistics have contributed to the development of decision tree-based algorithms and their ensemble variables such as random forests and AdaBoost. Hastie et al[10], Friedman et al [4], Quinlan [13] and Shapire et al [14] are some of the main contributors of the field. Their work however did not address scalability. Apache Mahout Spark MLlib is one of the only projects that addresses this issue. They train random forests by building multiple decision trees in parallel [7]. The individual trees however do no scale [15]. There are Scalable implementations of other machine learning algorithms such as SVM and Logistic Regression, and LibLinear [6] is a good example. With our algorithm incorporated into *scikit-learn* [5], the open source now supports scalability of tree-based algorithms on sparsely representable data.

.

VI. Conclusion

We proposed a method for building tree-based models with sparse input support. Our method takes advantage of input sparsity by avoiding sorting sample sets of a node along a feature unless they are nonzero at that feature. This approach speeds up training substantially as sorting is a costly but an essential and ubiquitous component of tree-based models.

VII. Acknowledgment

Arnaud Joly is a research fellow of the FNRS, Belgium. This work is partially supported by PASCAL2 and the IUAP DYSCO, initiated by the Belgian State, Science Policy Office.

References

[1] K. Bache and M. Lichman. UCI machine learning repository, 2013.
[2] S. D. Bay, D. F. Kibler, M. J. Pazzani, and P. Smyth. The uci kdd archive of large data sets for data mining research and experimentation. *SIGKDD Explorations*, 2:81, 2000.
[3] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.
[4] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen. *Classification and regression trees*. CRC press, 1984.
[5] L. Buitinck, G. Louppe, M. Blondel, F. Pedregosa, A. Mueller, O. Grisel, V. Niculae, P. Prettenhofer, A. Gramfort, J. Grobler, et al. Api design for machine learning software: experiences from the scikit-learn project. *arXiv preprint arXiv:1309.0238*, 2013.
[6] C.-C. Chang and C.-J. Lin. Libsvm: A library for support vector machines. *ACM Trans. Intell. Syst. Technol.*, 2(3):27:1–27:27, May 2011.
[7] H. Das, E. Sparks, A. Talwalkar, and X. Meng. http://databricks.com/blog/2014/09/29/scalable-decision-trees-in-mllib.html, 2014.
[8] Y. Freund and R. E. Schapire. A desicion-theoretic generalization of on-line learning and an application to boosting. In *Computational learning theory*, pages 23–37. Springer, 1995.
[9] J. H. Friedman. Greedy function approximation: a gradient boosting machine. *Annals of Statistics*, pages 1189–1232, 2001.
[10] T. Hastie, R. Tibshirani, and J. Friedman. *The elements of statistical learning: data mining, inference and prediction*. Springer, 2 edition, 2008.
[11] T. Joachims. A probabilistic analysis of the rocchio algorithm with tfidf for text categorization. Technical report, DTIC Document, 1996.
[12] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, et al. Scikit-learn: Machine learning in python. *The Journal of Machine Learning Research*, 12:2825–2830, 2011.
[13] J. R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.
[14] R. E. Schapire and Y. Singer. Improved boosting algorithms using confidence-rated predictions. In *Machine Learning*, pages 80–91, 1999.
[15] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'10, pages 10–10, Berkeley, CA, USA, 2010. USENIX Association.