Scalable Learning of Tree-Based Models on Sparsely Representable Data

Fares Hedayati
Elance-oDesk
Dept. of Data Science
441 Logue Ave, Mountain View, CA 94043
Email: fares19@elance-odesk.com

Arnaud Joly
Dept. of EE & CS & GIGA-R
University of Lige
Belgium
Email: a.joly@ulg.ac.be

Panagiotis Papadimitriou
Elance-oDesk
Dept. of Data Science
441 Logue Ave, Mountain View, CA 94043
Email: papadimitriou@elance-odesk.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 will be incorporated in the next version of *scikit-learn*, the most popular open source Python machine learning library.**

I. Introduction

High dimensional supervised learning problems such as text or image annotation are common in many research and practical problems. Such problems require defining a mapping between the raw input space and a target vector space, where each dimension is called a feature to some categorical or numerical output variable. A sample is a input-output pair. While those datasets have very high dimensional input space, 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. In order to work efficiently with such data, efficient matrix formats have been developed with fast operations, such as dot product, and low memory footprints.

Tree-based ensemble models, such as adaboost [1], random forest [2] or gradient tree boosting [3], are some of the most robust and widely-used supervised machine learning. What all these methods have in common is that they use randomized decision trees as a base learner. This building block is a hierarchical model which divide the input space through a series of binary splitting rules which partition the input space. Predictions of a decision tree is obtained by following the tree structure until reaching a leaf. In the ensemble framework, those models are either averaged [2] or learnt sequentially [1], [3].

Many models, such as linear or nearest neighbors model, could directly benefit from the input sparsity by formulating the entire algorithm through a set of dot products. However this is not possible for tree based methods, most machine learning packages don't 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* [4], [5] 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 provides our empirical implementation study and Section V 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. For the ease of notation we let $\mathcal{L} = (X, Y)$. 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. 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 this dataset. The Gini

impurity measure is defined as:

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

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 [6] is built by recursively splitting the sample set into to nodes by maximizing the average reduction of an impurity measure:

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

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 $\mathcal{L}$ into $\mathcal{L}_l$ and $\mathcal{L}_r$

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

and

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

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 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(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: Make t_p an internal node given splitting rule s.
13: Partition $\mathcal{L}_p$ into $\mathcal{L}_r$ and $\mathcal{L}_l$ given s^*.
14: Create two empty nodes t_r and t_l child 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. 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 taking 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). In order to avoid serval 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 $\theta = X_{i,j}$ $\theta = X_{i+1,j}$. 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. For more details of this algorithm refer to Algorithm 2 below.

Algorithm 2: Search for the best split

1: **function** FINDBESTSPLIT($\mathcal{L}_p$)
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 of splitting rule 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 format. It is a general format to represent compactly sparse matrices using three arrays: a $data$ array stores the value of each nonzero elements, an $indices$ array stores the row index of each nonzero elements and an $indptr$ array which stores the beginning and end of each columns in the $data$ and the $indices$ arrays.

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

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

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.

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 4 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.

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 have to be able to perform the intersection between the sample set $\mathcal{L}_p$ which have reached the node and the $m_j = indptr[j+1] - indptr[j]$ nonzero elements of the feature j as to generate a set of possible splitting rules. 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;

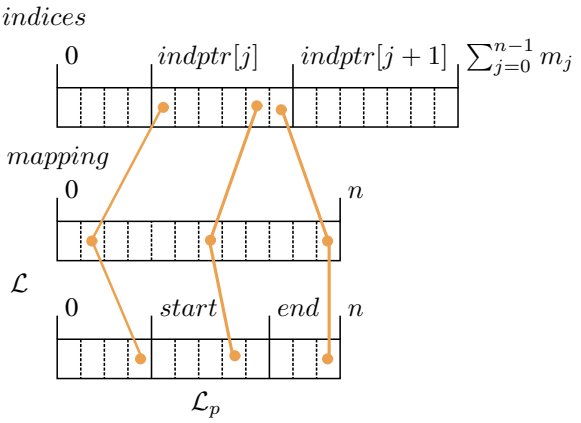

Fig. 1. The array $mapping$ allow to efficiently compute the intersection between the $indices$ array of the csc matrix and a sample set $\mathcal{L}_p$

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 $O(m_j + |\mathcal{L}_p|)$ by first creating a temporary hash table from the one array and then checking if elements of the other array 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 sampled feature and for various sample sets $\mathcal{L}_p$. Taking this into account, it's possible to improve approach (4). The idea is to maintain during the tree growth a mapping, represented at Figure 1, between the row index, the $indices$ array, of the csc matrix and the position of the related samples in the sample set array $\mathcal{L}$. Since each sample only belongs to one tree branch, a subset $\mathcal{L}_p$ of $\mathcal{L}$ can be conveniently represented by a slice $[start, end[$ of the array $\mathcal{L}$. Thus, it's possible to check in $O(1)$ if the k-th nonzero element of the csc matrix belongs to the sample set $\mathcal{L}_p$ by checking if $mapping[indices[k]]$ is in $[start, end[$. Maintaining the mapping for a given position pos is done in $O(1)$ by setting $mapping[\mathcal{L}[pos]]$ to pos. Thus we deduce that performing the intersection between the $indices$ array and $\mathcal{L}_p$ can be done in $O(m_j)$.

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 13 of Algorithm 1) given a split on feature j in $O(m_j)$ instead of $O(n)$. For more details of this algorithm refer to Algorithm 3.

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 to approach (1) based on binary search. whenever

$\mathcal{L}_p \ll m_j$, the binary approach will be faster. For more details of this algorithm refer to Algorithm 4.

The hybrid algorithm switches between 4 and 3 by the following rule:

$$(1-sorted)\times|\mathcal{L}_p|\times\log(|\mathcal{L}_p|)+|\mathcal{L}_p|\times\log(m_j) < 0.1m_j \quad (6)$$

where $sorted$ is 1 if $\mathcal{L}_p$ is sorted and 0 otherwise. Algorithm 3 is used whenever Equation 6 is true and Algorithm 4 is used otherwise.

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.

Algorithm 3: Extract nonzero values of the current node, i.e. $\mathcal{L}[start : end]$, via $mapping$, and return the positive and negative values separately. Note that at end the samples with negative values are pushed to the beginning of $\mathcal{L}[start : end]$ and the samples with positive values to its end.

function EXTRACT_NNZ_MAPPING($\mathcal{L}$, X, $mapping$, j, $start$, end)
  $positives$ = [ ]
  $negatives$ = [ ]
  $incides$ = X.indices
  $indptr$ = X.indptr
  $data$ = X.data
  for $k \in [indptr[j]$:$indptr[j+1]]$ **do**
    $index = indices[k]$
    $value = data[k]$
    if $start \leq mapping[index] < end$ **then**
      h = $mapping[index]$
      if $value > 0$ **then**
        $positives$.APPEND($value$)
        SWAP($\mathcal{L}$, h, $start_p$, $mapping$)
      else
        $negative$.APPEND($value$)
        SWAP($\mathcal{L}$, h, end_n, $mapping$)
        end_n $+= 1$
      end if
    end if
  end for
  return $(start, end_n), (start_p, end), positives, negatives$
end function

Algorithm 4: Extract nonzero values of the current node, i.e. $\mathcal{L}[start : end]$, by a binary search, and return the positive and negative values separately. Note that at end the samples with negative values are pushed to the beginning of $\mathcal{L}[start : end]$ and the samples with positive values to its end.

function EXTRACT_NNZ_BSEARCH($\mathcal{L}$, X, $mapping$, j, $start$, end, $sorted$)
  $positives$ = [ ]
  $negatives$ = [ ]
  $incides$ = X.indices
  $indptr$ = X.indptr
  $data$ = X.data
  $indices_j = indices[indptr[j] : indptr[j+1]]$
  $data_j = data[indptr[j] : indptr[j+1]]$
  if $sorted = False$ **then**
    $\mathcal{L}$ = SORT($\mathcal{L}$, start, end)
  end if
  for $h \in [start : end]$ **do**
    $index$ = $\mathcal{L}[h]$
    i = BINARYSEARCH($index$, $indices_j$)
    # Returns the position of $index$ in $indices_j$,
    # and -1 if it is not found.
    if $i \neq -1$ **then**
      if $data_j[i] > 0$ **then**
        end_p $-= 1$
        $positives$.APPEND($value$)
        SWAP($\mathcal{L}$, h, $start_p$, $mapping$)
      else
        $negative$.APPEND($value$)
        SWAP($\mathcal{L}$, h, end_n, $mapping$)
        end_n $+= 1$
      end if
    end if
  end for
  return $(start, end_n), (start_p, end), positives, negatives$
end function

function SWAP($\mathcal{L}$, pos_1, pos_2, $mapping$)
  $\mathcal{L}[pos_1]$, $\mathcal{L}[pos_2]$ = $\mathcal{L}[pos_2]$, $\mathcal{L}[pos_1]$
  $mapping[\mathcal{L}[pos_1]] = pos_1$
  $mapping[\mathcal{L}[pos_2]] = pos_2$
end function

IV. EXPERIMENTS

A. density

B. Read Data

We will assess the computational performance of the decision tree algorithm on datasets dominantly composed of categorical or textual features, which makes them very sparse, and on datasets that are more dense. We will compare the learning time 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 [7] and on the *KDD cup 1999* dataset[8]. 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 tree as used in boosting methods or deep tree as random forest ensemble methods. The 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 for this dataset is regression. We will compare the learning 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 times. Furthermore,

both algorithm leads exactly to the same decision tree regressor structure and have the same generalization performance.

Our dense datasets are the *Adult* dataset[9] with a density of 0.12, $n = 32561$ instances and $m = 145$ features, and the *TIC* dataset[9] with a density of 0.44 and $n = 4000$ instances and $m = 85$ features. As Figure 4 shows for shallower trees sparse threes are approximately learnt twice faster than their dense counterparts, but for fully grown threes it is the opposite. Finally as Figure 5 shows decison threes 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 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.

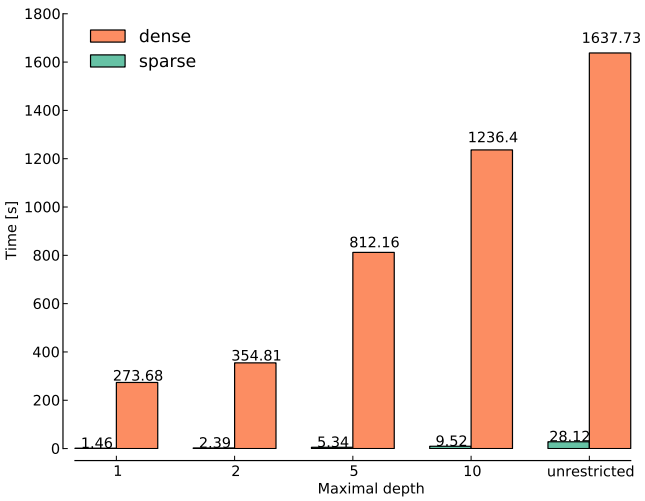

Fig. 2. Leveraging the input sparsity significantly speed up decisions tree induction both with shallow and deep trees on the *20 Newsgroups* dataset. Note that the dataset is very sparse (density = 0.001).

V. 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 essential and ubiquitous component of tree-based models.

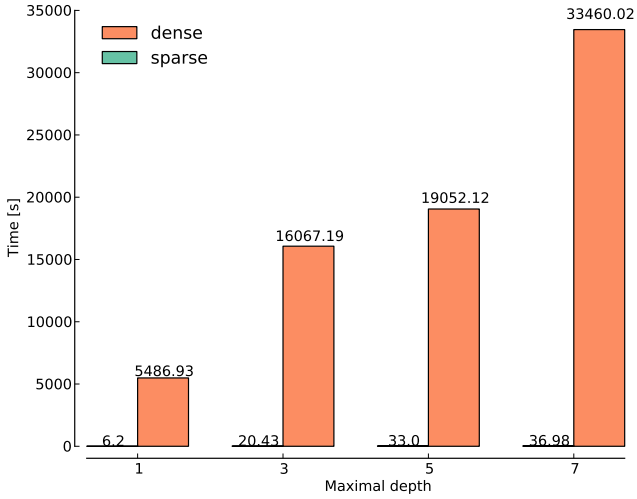

Fig. 3. Leveraging the input sparsity significantly speed up decisions tree induction both with shallow and deep trees on the *cup* dataset. Note that the dataset is quite sparse (density = 0.014).

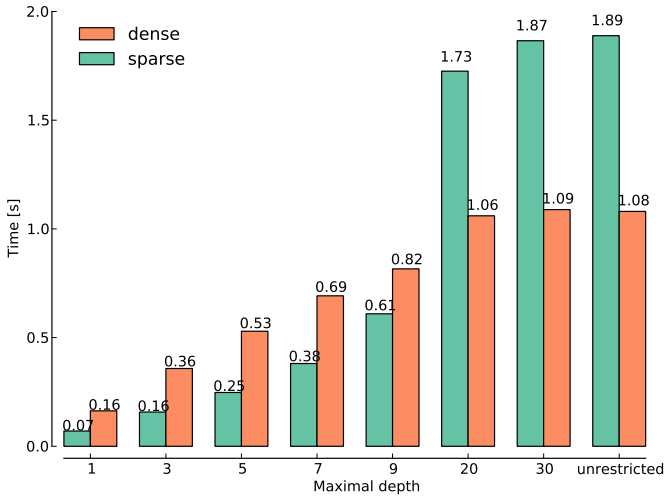

Fig. 4. Leveraging the input sparsity does not speed up training of deep trees on the *adult* dataset. Note that the dataset is quite dense (density = 0.11).

VI. 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] Y. Freund and R. E. Schapire, "A desicion-theoretic generalization of on-line learning and an application to boosting," in *Computational learning theory*. Springer, 1995, pp. 23–37.

[2] L. Breiman, "Random forests," *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[3] J. H. Friedman, "Greedy function approximation: a gradient boosting machine," *Annals of Statistics*, pp. 1189–1232, 2001.

[4] 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.

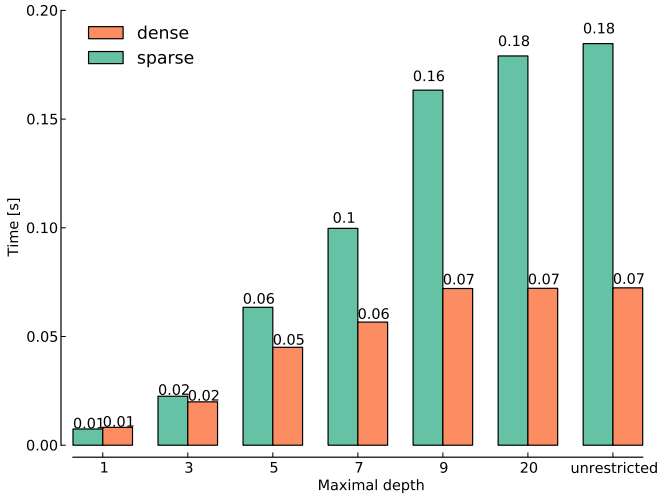

Fig. 5. Leveraging the input sparsity does not speed up training trees on the *tic* dataset. Note that the dataset is very dense (density = 0.44).

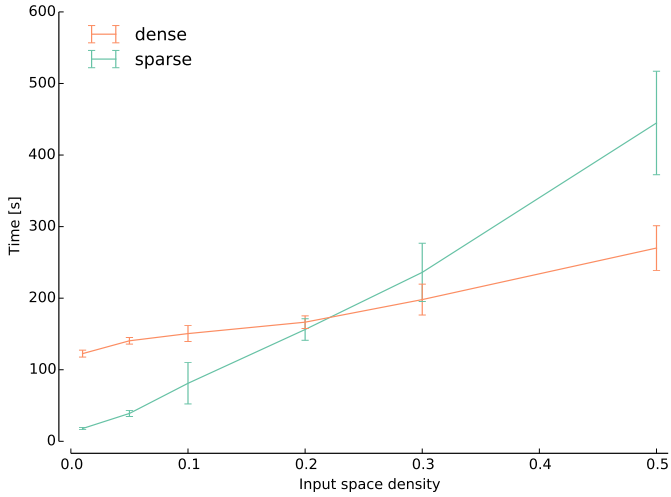

Fig. 6. Significant speed up is achieved by the sparsity-aware decision tree algorithm whenever the density is below 0.2 (or sparsity over 0.8).

[5] 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*, vol. 12, pp. 2825–2830, 2011.

[6] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and regression trees*. CRC press, 1984.

[7] T. Joachims, "A probabilistic analysis of the rocchio algorithm with tfidf for text categorization." DTIC Document, Tech. Rep., 1996.

[8] 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*, vol. 2, p. 81, 2000.

[9] K. Bache and M. Lichman, "UCI machine learning repository," 2013. [Online]. Available: http://archive.ics.uci.edu/ml