Leveraging Input Space Sparsity to Scale Tree-Based Models

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 involve sparsely representable input space. State-of-the-art tree-based ensemble algorithms and implementations have focused on tasks with dense input space, while at most emulating random memory access on sparse input data. We propose a fast and an efficient splitting algorithm to leverage input sparsity within decision tree methods. We exploit the a-priori knowledge that each column have very few non- zero elements and show how learning time is significantly decreased.**

I. Introduction

High dimensional supervised learning problems, e.g. in text or image annotation, are more frequent than ever. It consists in searching a mapping between an input 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 all 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 a 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. All these methods have in common that they use randomized decision trees as a base learner. This building block is a hierarchical model which divide the input space through a serie 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 non zeros allows large speed up. 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 inclusion to 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. Background

A. Induction of decision trees

We denote by $\mathcal{X}$ an input space and by $\mathcal{Y}$ the output space. Without loss of generality, we suppose that $\mathcal{X} = \mathcal{R}^p$ where p denotes the number of features. Learning samples are represented by a pair of matrix $(X, Y) \subseteq (\mathcal{X}, \mathcal{Y})_{i=0}^{n-1}$, where each row corresponds to a sample and each column to a feature or an output variable.

A decision trees [6] is built by recursively maximizing the average reduction of an impurity measure, such as the variance,

$$\Delta I(s, \mathcal{L}) = I((Y_i)_{i \in \mathcal{L}}) - \frac{|\mathcal{L}_r|}{|\mathcal{L}|} I((Y_i)_{i \in \mathcal{L}_l}) - \frac{|\mathcal{L}_l|}{|\mathcal{L}|} I((Y_i)_{i \in \mathcal{L}_r}))$$

where s is a binary partition of the input space which divide the sample set $\mathcal{L}$ into $\mathcal{L}_l$ and $\mathcal{L}_r$. 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.

In the context of ensemble, tree 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

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

$$s^* = \arg \max_{s \in Q(\mathcal{L}_p, X)} \Delta I(s, \mathcal{L}_p).$$

      Make t_p an internal node given splitting rule s.
      Partition $\mathcal{L}_p$ into $\mathcal{L}_r$ and $\mathcal{L}_l$ given s^*.
      Create two empty nodes t_r and t_l child of t_p.
      $stack$.PUSH((t_r, $\mathcal{L}_r$))
      $stack$.PUSH((t_l, $\mathcal{L}_l$))
    end if
  end while
  return τ
end function

Algorithm 2: Search for the best split

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

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

5:     Sort $\mathcal{L}$ and $\mathcal{X}_j$ by increasing values of $\mathcal{X}_j$.
6:     Generate all possible splitting rules

$$Q(\mathcal{X}_j) = \{((x_j \leq \nu), (x_j > \nu)) | \nu \in \mathcal{X}_j)$$

7:     **for** s in $Q(\mathcal{X}_j)$ **do**
8:       Evaluate impurity reduction of splitting rule s

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

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**

B. 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 non zero elements, an $indices$ array stores the row index of each non zero 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.

III. GROWTH OF DECISION TREES ON SPARSE INPUT DATA

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.

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 $m_j = indptr[j+1] - indptr[j]$ non zero 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 non zero elements;
2) in $O(|\mathcal{L}_p| \log |\mathcal{L}_p| + m_j \log |\mathcal{L}_p|)$ by sorting the sample set $\mathcal{L}$ 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.

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

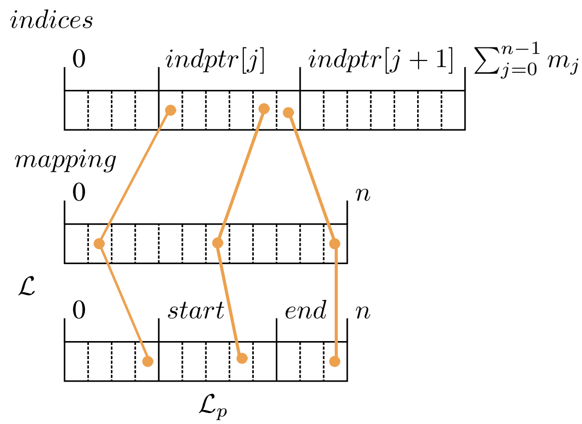

Fig. 1. The array *mapping* allow to efficiently compute the intersection between the *indices* array of the csr matrix and a sample set $\mathcal{L}_p$

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 non zero 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 p is done in $O(1)$ by setting $mapping[\mathcal{L}[p]]$ to p. 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 $O(n)$.

In practice, the number of non zero 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.

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.

IV. Experiments

In the first experiment we trained different decision tree classifiers on the *20 Newsgroups* dataset, changing the max_depth only and keeping the default values for others, i.e. $min\_samples\_split = 2$ and $min\_samples\_leaf = 1$. The dataset consists of 20000 news groups documents distributed across 20 newsgroups almost evenly [7]. For each value of max_depth we trained two decision tree classifiers: one time with the training data represented in a csc_matrix sparse matrix and one time with the data represented in a dense matrix. For each of these pairs we verified that the trees are identical and compared their training times. As Figure 2 shows, the training time with a densely represented matrix is much higher than its sparsely represented counterpart.

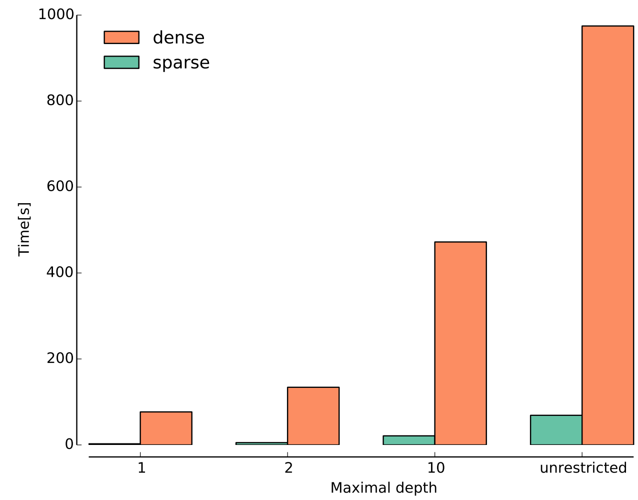

Fig. 2. Training Time for Different max_depth on Sparse vs. Dense Input (20 News Groups)

The second experiment was carried out on synthetic data. This time we trained different decision tree classifiers, with $max\_depth = 20$ and keeping the default values for other parameters. We generated random matrices of varying densities each of size 100000×1000. Density is the percentage of non-zero values for each feature. For example in a matrix with $density = 0.1$ every feature is 10% of the time non-zero. For each density we created 10 random matrices, and each of these matrices were represented in a sparse csc_matrix and a dense matrix and decision tree classifiers were fitted on both formats. Figure 3 shows the average training time for each density with one standard deviation. As the figure shows the training time of sparse matrices is much lower that their dense counterparts when the density of the matrix is quite low, i.e. less than 0.2. On the other hand when the density is high, e.g. $density = 0.5$, the training time with dense matrices input is lower than their sparse counterparts. The figure suggests that csc_matrix should only be used when the data is really sparse, e.g. with textual data.

V. Conclusion

The conclusion goes here.

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.

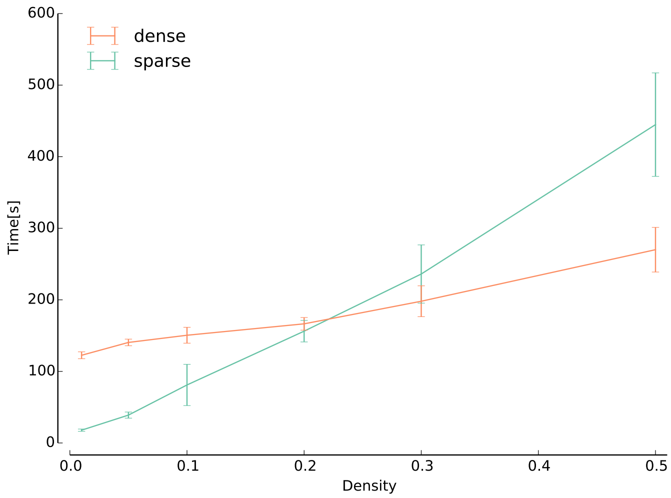

Fig. 3. Training Time of Random Matrices with $n\_samples = 100000$, $n\_features = 1000$ and Different Densities on Sparse vs. Dense Input

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.

[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.