Sparsified Trees

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

Arnaud Joly
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 techniques such as logistic regression support sparse inputs and work very efficiently with them. Decision trees however do not support sparse input feature. We propose a fast and efficient way of training decision trees on sparse datasets. Sorting datasets along features for finding the best possible split is a very expensive and at the same time an essential component in training decision trees. Sorting is specially costly when working with textual data or any dataset with gigantic feature dimensions. At the heart of our training algorithm lies working with matrices with sparsely represented columns and limited sorting of non-zero parts of columns. We have implemented this method and merged it into the machine learning package *scikit-learn*. Our method adds sparse input support to the existing tree training algorithms with a significant decrease in the training time.**

I. Introduction

Decision trees along with their ensemble variations such as Adaboost, Random Forest and Gradient Boosting are some of the most robust and widely-used supervised machine learning and data mining techniques [6], [7], [1]. Decision trees can be used both in classification and regression problems. A tree is a flowchart structure that guides the input feature vector from the root to a leaf; and the leaf contains the necessary information for predicting the target variable. The root and each intermediate inner node correspond to a feature and a splitting criterion that determine the path of the input down the tree. As learning a tree from training dataset is known to be NP-complete [3], a heuristic is used instead. The training dataset is recursively split into homogenous parts[6]. For the case of classification with numerical features the training data is recursively partitioned into two parts by picking a single feature and a threshold that can best split the data into two homogenous groups in terms of the target variable. The optimality of the split can be determined by different splitting scores, information gain and Gini impurity are popular choices [2]. Finding the best threshold of a feature requires sorting the dataset along the feature and computing the splitting score for all distinct feature values.

Currently decision trees do not support sparse input in *scikit-learn* [5] and many other packages; the input is required to be densified first. Memory is not the only challenge, training time is the real predicament. Training decision trees on datasets with a large feature dimension such as textual datasets is a challenge as the dataset needs to be sorted along most of the features at least once. We exploit sparsity of features to avoid sorting the entire dataset; only nonzero parts of a feature are sorted and the dataset is rearranged accordingly. Moreover datasets are stored in a column-wise sparsely-represented matrix for memory efficiency.

II. Decision Trees

In this section we briefly go over training decision trees and in the next section we will show how to tweak this algorithm for support of sparse inputs. Let X be a matrix of dimension $n \times d$ where d is the dimension of the feature space and n is the number of training instances, i.e. each row corresponds to an instance and each column corresponds to a feature. Moreover we let Y be a vector of size n that contains the target variables, e.g. the target variable of instance i is at $Y[i]$. Note that the in the function *FIT* below, $min\_samples\_split$, $min\_samples\_leaf$ and max_depth are regularization parameters for prevention of overfitting. $min\_samples\_split$ is the minimum number of training instances required in a node before a split is carried on the node, $min\_samples\_leaf$ is the minimum number of instances required in the node and max_depth is the maximum depth of the tree. As soon as there are too few instances in the current node or the best split of the current node leave at least one of the children with too few instances or the tree becomes too deep no further splitting is carried out and the current node becomes a leaf node. The tree can be grown alternatively in a best-first search manner in which case max_leaf_nodes is needed as well. $max\_features$, $splitter$ and $criterion$ are other parameters of the model. Since our tweak is independent of these parameters we only cover training decision trees for classification in a depth-first search manner with information gain. Our change works well with all tree variations. For more details refer to the latest merge in *scikit-learn* in the tree and random-forests sections [5].

To avoid rearranging the whole matrix at the time of sorting, we use an auxiliary data structure *samples* to rearrange the indices of the training dataset only, keeping X and Y unchanged. Initially $samples[i] = i$ for $i \in \{0, 1, \cdots, n-1\}$, where n is the number of training instances. Each node is specified by two numbers $start$ and end. These two numbers along with $samples$ specifies the indices of X, Y that are in the current node: $samples[start : end]$. Note that we use this name to be consistent with *scikit-learn* [5].

```
function FIT($start = 0, end = n, Node, depth$)
    if $|end - start| < min\_samples\_split$ or $depth > max\_depth$ then
        $Node.IsLeaf = True$
        $Node.Class =$ $Most\ frequent$
          $class\ of\ Y[samples[start : end]]$
        return $Node$
    end if
    $f, t = BestSplit(start, end)$

    Rearrange $samples[start : end]$
    such that instances in $samples[start : end]$
    with feature $f$ less than $t$ are before
    those that are great than $t$.
    Let $mid$ be an index in the rearranged samples
    such that $X[samples[start : mid], f]$ are less than
    $X[samples[mid : end], f]$

    if $|mid - start|$ or $|end - mid| < min\_samples\_leaf$ then
        $Node.IsLeaf = True$
        $Node.Class =$ $Most\ frequent$
          $class\ of\ Y[samples[start : end]]$
        return $Node$
    end if
    $Node.Feature = f$
    $Node.Threshold = t$
    $Node.LeftNode = new\ Node()$
    $Node.RightNode = new\ Node()$
    FIT($start, mid, Node.LeftNode, depth + 1$)
    FIT($mid, end, Node.RightNode, depth + 1$)
    return $Node$
end function
function BESTSPLIT(start, end)
    $BestThreshold = 0$
    $BestScore = -\infty$
    $BestFeature = 0$
    for $f \in \{1, \cdots, d\}$ do
        $Sort(X, samples, f, start, end)$
        for $i \in [start + 1 : end]$ do
            $t = X[samples[i], f]$
            $Y_L = Y[samples[start : i]]$
            $Y_R = Y[samples[i : end]]$.
            $H_L = \mathrm{H}(Y_L)$
            $H_R = \mathrm{H}(Y_R)$
            $S = H(Y) - \left(\frac{|i-start|}{|end-start|}H_L + \frac{|end-i|}{|end-start|}H_R\right)$
            Where $H(\cdot)$ is the entropy
            $T = t$
            if $S > bestScore$ then
                $BestScore = S$
                $BestThreshold = T$
                $BestFeature = f$
            end if
        end for
    end for
    return $BestFeature, BestThreshold$
end function
function SORT($X, samples, f, start, end$ )
    Rearrange $samples[start : end]$
    by sorting $X[samples[start : end], f]$,
    keep $X$ and $Y$ unchanged.

end function
```

III. Sparsified Decision Trees

For memory efficiency and taking advantage of sparsity we use a data structure called *scipy.sparse.csc_matrix*. This data structure has a common use in *scikit-learn*. It consists of four parts: *indices*, *indptr*, *data* and *shape* which contains the dimensions of the matrix. The indices for column f are stored in:

$$indices[indptr[f] : indptr[f + 1]], \tag{1}$$

with the corresponding values at

$$data[indptr[f] : indptr[f + 1]]. \tag{2}$$

For example if the $indptr$, $indices$, and $data$ of a (3×3) csc_matrix be the following:

$indptr = [0, 2, 3, 6]$
$indices = [0, 2, 2, 0, 1, 2]$
$data = [1, 2, 3, 4, 5, 6]$

Then the (dense) matrix itself should be:

$[1, 0, 4$
$0, 0, 5$
$2, 3, 6]$

The only part of the $BestFit$ algorithm of the previous section that changes is the way we sort the training matrix along a feature. For a given feature f we first find all instances in $samples[start : end]$ where feature f is non-zero. We, then sort the positive and negative parts separately and rearrange $samples[start : end]$ accordingly.

The challenge in finding instances in $samples[start : end]$ with non-zero values of f is that of finding the intersection of $indices[indptr[f] : indptr[f + 1]]$ and $samples[start : end]$. Depending on the size of these two sets we can either perform a binary search or an exhaustive search. For the latter end we need an additional auxiliary data structure that we call $index\_to\_samples$ which is a reverse map of $samples$ to its indices, i.e. $index\_to\_samples[j] = i$ when $samples[i] = j$.

If the size of the current node, i.e. $end - start$, is small then we can sort $samples[start : end]$ and for each instance in $samples[start : end]$ do a binary search in $indices[indptr[f] : indptr[f+1]]$, this way we can find the intersection of two sets which is the set of instance of the current node with non-zero values of feature f. On the other hand if the size of the current node is large, sorting the set might be costly and we do an exhaustive search instead. Note that we always keep $index\_to\_samples$ up-to-date. For each instance $index$ in $indices[indptr[f] : indptr[f+1]]$ if $index\_to\_samples[index]$ is not between $start$ and end then that instance is not in the current node. So we only keep instances in $indices[indptr[f] : indptr[f+1]]$ when their corresponding $index\_to\_samples$ is between $start$ and end.

We define the following two variables for the current feature f

$$n\_indices = indptr[f+1] - indptr[f] \tag{3}$$

$$n\_samples = end - start. \tag{4}$$

Sorting of $samples[start : end]$ takes $O(n\_samples \times \log(n\_samples))$ and binary search of instances of $samples[start : end]$ in $indices[indptr[f] : indptr[f+1]]$ takes $O(n\_samples \times \log(n\_indices))$. Sorting of $samples[start : end]$ happens only once in a $BestSplit$ call for all candidate features, and the binary search happens for each candidate feature. On the other hand, the exhaustive search takes $O(n\_indices)$ time. We let C be:

$$n\_samples \times \log(n\_samples) + n\_samples \times \log(n\_indices)$$

After testing with different coefficients we came up with the following rule for the candidate feature f:

```
if $C < 0.1 \times n\_indices$ then
    First sort $samples[start : end]$ if it is not already sorted, then for each instance in $samples[start : end]$ carry on a binary search in $indices[indptr[f] : indptr[f+1]]$
else
    For each index in $indices[indptr[f] : indptr[f+1]]$ check if its in $samples[start : end]$ by verifying that its $index\_to\_samples$ is between start and end.
end if
```

IV. Results

The first test was carried on the *20 Newsgroups* dataset. The dataset consists of 20000 news groups documents distributed across 20 newsgroups almost evenly [4]. Training was carried on with different parameters of max_depth, once with the csc_matrix sparse matrix and once with dense matrix. As it can be seen in Figure 1, the training time with dense inputs is much higher than the in their sparse counterparts. We verified that all 4 fitted trees are identical.

The second set of tests were carried on synthetic data. Random matrices were generated with different shapes and densities where a density is the percentage of non-zero values of a feature. For example a density of 0.1 means that every feature is only 10% of the time non-zero. Each matrix was represented both in a sparse and dense format. Decision trees were fitted once with the sparse and once in the dense format. Table I contains the training time of these matrices (the last two columns). As it can be seen in this table the training time of sparse matrices is much lower that their dense counterparts when the density of matrix is quite low, i.e. less 0.1. For no-quite sparse matrices, e.g. $density = 0.5$, the training time with dense inputs is lower than their sparse counterparts. As this table suggests csc_matrix should only be used in sparse data, e.g. textual data.

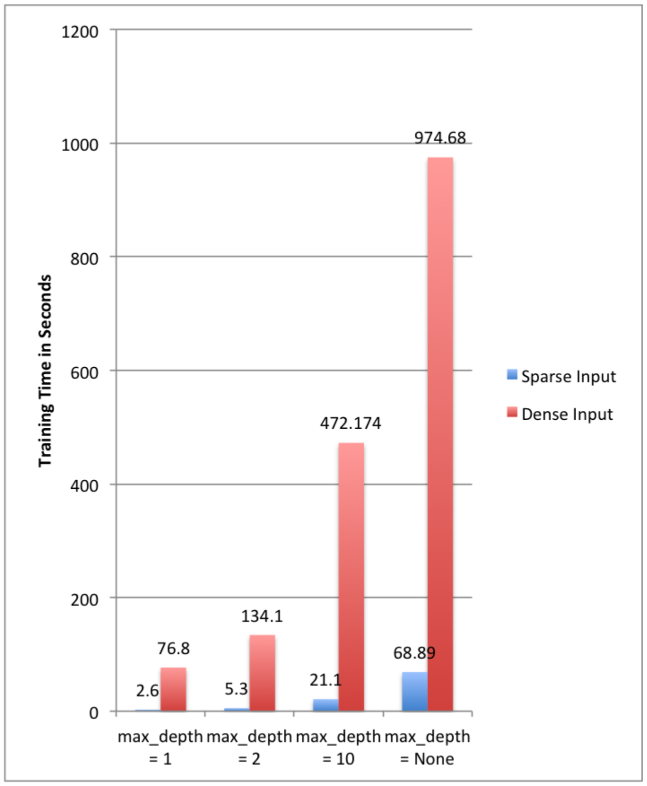

Fig. 1. Training Time for Different max_depth on Sparse vs. Dense Input (20 News Groups)

TABLE I. Training Time in Seconds with Sparse and Dense Inputs

Dataset Size	Feature Size	density	Sparse Input	Dense Input
10000	1000	0.01	4.74	25.07
100000	100	0.01	7.26	24.65
100000	1000	0.01	89.13	507.86
10000	1000	0.05	9.82	16.14
100000	100	0.05	21.14	28.27
100000	1000	0.05	256.68	541.00
10000	1000	0.1	13.42	12.38
100000	100	0.1	37.81	24.65
100000	1000	0.1	437.14	370.60
10000	1000	0.5	28.97	14.02
100000	100	0.5	100.22	28.13
100000	1000	0.5	949.14	383.62

V. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank ...

References

[1] Breiman, Leo (2001). “Random Forests”. Machine Learning 45 (1): 5–32.

[2] Deng,H.; Runger, G.; Tuv, E. (2011). “Bias of importance measures for multi-valued attributes and solutions”. Proceedings of the 21st International Conference on Artificial Neural Networks (ICANN2011). pp. 293–300.

[3] Hyafil, Laurent; Rivest, RL (1976). “Constructing Optimal Binary Decision Trees is NP-complete”. Information Processing Letters 5 (1): 15–17.

[4] Ken Lang, “Newsweeder: Learning to filter netnews”, *Proceedings of the Twelfth International Conference on Machine Learning* ,pp. 331–339.

[5] Pedregosa, F. and Varoquaux, G. and Gramfort, A. and Michel, V. and Thirion, B. and Grisel, O. and Blondel, M. and Prettenhofer, P. and Weiss, R. and Dubourg, V. and Vanderplas, J. and Passos, A. and Cournapeau, D. and Brucher, M. and Perrot, M. and Duchesnay, E. “Scikit-learn: Machine Learning in Python”, JMLR 12, pp. 2825–2830, 2011.

[6] Rokach, Lior; Maimon, O. (2008). “Data mining with decision trees: theory and applications”. World Scientific Pub Co Inc

[7] Yoav Freund and Robert E.Schapire, “A decision-theoretic generalization of on-line learning and an application to boosting”. Journal of Computer and System Sciences 55. 1997.