Towards Scalable Quantile Regression Trees

Harish S. Bhat
Applied Mathematics Unit
UC Merced
Merced, USA
hbhat@ucmerced.edu

Nitesh Kumar
Skytree, Inc.
San Jose, USA
nitesh@skytree.net

Garnet J. Vaz
Microsoft
Bellevue, USA
gavaz@microsoft.com

***Abstract*—We provide an algorithm to build quantile regression trees in O(N log N) time, where N is the number of instances in the training set. Quantile regression trees are regression trees that model conditional quantiles of the response variable, rather than the conditional expectation as in standard regression trees. We build quantile regression trees by using the quantile loss function in our node splitting criterion. The performance of our algorithm stems from new online update procedures for both the quantile function and the quantile loss function. We test the quantile tree algorithm in three ways, comparing its running time against implementations of standard regression trees, demonstrating its ability to recover a known set of nonlinear quantile functions, and showing that quantile trees yield smaller test set errors (computed using mean absolute deviation) than standard regression trees. The tests include training sets with up to 16 million instances. Overall, our results enable future use of quantile regression trees for large-scale data mining.**

***Keywords*-regression trees; quantile regression; online algorithm**

I. Introduction

Decision trees are one of the most widely used methods in data mining. Trees enjoy various advantages, among which are interpretability, a natural ability to handle both numerical and categorical predictors, and well-developed methods to deal with missing data. Trees are typically constructed using recursive partitioning algorithms, which can be efficient and scalable. In the present work, we focus on quantile regression trees, which are regression trees designed to model conditional quantiles of the response variable. This is in contrast to standard regression trees, which model the conditional expectation. Our primary contribution is an algorithm that constructs quantile regression trees in $O(N \log N)$ time, where N is the number of instances in the training set.

We are interested in quantile regression trees for two reasons. The first reason has to do with robustness. When tree models are applied to regression problems, the most widely used splitting criterion employs an ordinary least squares (OLS) loss function. To generate trees that are more robust to outliers in the response variable, Breiman et al. [1, Chap. 8] suggested a splitting criterion based on least absolute deviation (LAD). In this approach, when we arrive at a leaf node, the predicted value will be the median of the response variable for the instances associated with the leaf. In this paper, we generalize Breiman's framework to arrive at quantile trees. The splitting criterion uses a tilted absolute value loss function (2) that, in a natural way, allows us to develop a model for the τth quantile of the response variable. The *LAD tree* is included as a special case.

The second reason for our interest in quantile regression trees is a desire for more informative models than one obtains with OLS regression trees. Let $\mathbf{X}$ be an $N \times p$ matrix of predictors, where each row is a different instance and each column is a different predictor. Let Y, an $N \times 1$ vector, be the corresponding response variable. We regard each row of $(\mathbf{X}, Y)$ as an independent sample from the random vector $(\boldsymbol{\xi}, \eta)$. Let

$$F_{\eta|\boldsymbol{\xi}}(a|\mathbf{b}) = P(\eta \leq a|\boldsymbol{\xi} = \mathbf{b})$$

denote the conditional cumulative distribution function (CDF). Then the quantile regression tree with parameter τ is an approximation of the function $y = \Phi_\tau(\mathbf{x})$ that satisfies

$$F_{\eta|\boldsymbol{\xi}}(\Phi_\tau(\mathbf{b})|\mathbf{b}) = \tau.$$

In short, Φ_τ is the inverse of the conditional CDF, and the quantile tree is a piecewise constant approximation of Φ_τ. Developing quantile trees for a range of values for τ, we can approximate the conditional CDF of the response given the predictors. OLS regression trees seek to model $E[\eta|\boldsymbol{\xi}]$. The difference between quantile and OLS regression trees, therefore, can be understood as the difference between estimating the conditional distribution and conditional expectation. The additional information in the distribution has proven useful in various problems [2]–[4].

A classic paper on gradient boosting [5] states: *"Squared error loss is much more rapidly updated than mean-absolute-deviation when searching for splits during the tree building process."* We argue that it is for this reason that, despite the potential advantages outlined above, neither LAD nor quantile regression trees enjoy widespread use on massive data sets. In this paper, we detail an algorithm for updating the quantile loss function ρ_{QT} that enables quantile regression trees to be built in $O(N \log N)$ time;

prior implementations of which we are aware run in $O(N^2)$ or even $O(N^2 \log N)$ time.

Quantile regression trees are, of course, examples of regression trees; for a recent survey of the regression tree literature, see [6]. The present paper does not describe a complete regression tree package; such a package would provide methods to handle categorical predictors and missing data, and should also address the bias problem. For trees, the bias problem is the tendency to split on a predictor that has more possible splits, even if the predictor is completely uninformative. The present paper seeks to remedy the algorithmic obstacle identified above, enabling incorporation of the quantile loss function into more complete tree packages suitable for real data sets.

In prior work on quantile regression trees [7], [8], a polynomial approximation to the conditional quantile function is fitted to the instances in each terminal node. In this scheme, the node splitting criterion is a function of p-values calculated using linear quantile regression residuals. Hence these works do not address regression trees built using the quantile loss function. Trees built using robust M-estimator loss functions [9] are closer to what we analyze here.

Quantile trees can also be viewed as an example of a nonlinear, nonparametric method for estimating regression quantiles. Several such methods have been developed recently, including quantile regression methods featuring neural networks [10], additive models with splines [11], local smoothing [12], [13], expectation maximization [14] and forests of OLS trees [15]. If one restricts attention to LAD or median models, there are numerous nonlinear methods to consider [16]–[19]. The most relevant work is [20], which explored online algorithms for LAD trees.

II. General Tree Algorithm

Given the predictors $X_1, \ldots, X_p$ and the response variable Y, regression trees seek to minimize the deviance

$$S_{\text{node}} = \sum_{i \in \text{node}} \rho_{\text{model}}(y_i - \theta_{\text{model}}), \tag{1}$$

summed across all leaf nodes. Here ρ_{model} is a loss function, i.e.,

$$\rho_{\text{OLS}}(x) = x^2, \quad \rho_{\text{QT}}(x) = \begin{cases} (\tau - 1)x & x < 0 \\ \tau x & x \geq 0, \end{cases} \tag{2}$$

and θ_{model} is the associated M-estimator,

$$\theta_{\text{model}} = \operatorname*{argmin}_{y^*} \sum_{i \in \text{node}} \rho_{\text{model}}(y_i - y^*).$$

We equate a node with the indices of associated instances, $\{i_1, \ldots, i_\nu\}$. For the same node, we have an associated vector of response values, $\mathbf{y} = (y_{i_1}, \ldots, y_{i_\nu})$. The M-estimators for the two loss functions are

$$\theta_{\text{OLS}} = \mu(\mathbf{y}), \qquad \theta_{\text{QT}} = F_{\mathbf{y}}^{-1}(\tau).$$

Here $\mu(\mathbf{y})$ is the mean of $\mathbf{y}$ and $F_{\mathbf{y}}^{-1}(\tau)$ is the empirical τth quantile of $\mathbf{y}$.

Once the model and the loss function ρ have been chosen, the tree can be built using a recursive algorithm. Fix $\delta > 0$ and positive integers n_m and n_c. The parameter n_m decides the minimum number of sample points in $\mathbf{y}_i$ required to attempt a split at node i. The parameter n_c is the minimum number of sample points required at any node. The parameter δ determines a sufficient level of relative decrease in summed deviance. If one compares with the *tree* package [21] in R, our parameters n_m, n_c, and δ correspond to the tree package's minsize, mincut, and mindev, respectively.

Initially, let all instances be contained in a single node, the *root* node. We compute S_{root} via (1). Then the recursive splitting algorithm is:

S1) If $|\text{node}| \leq n_m$, then label node as a *leaf*. Go to S6.

S2) Calculate the node deviance S_{node} using (1). When we split on a fixed predictor X_i, let l and r denote the sets of instances that branch to the left and right, respectively. Let L and R denote the collection of all distinct sets l and r that are obtained by considering all binary splits over all predictors X_i. Set

$$\phi(l, r) = S_{\text{node}} - (S_l + S_r)$$

and calculate

$$(l^*, r^*) = \operatorname*{argmax}_{l \in L, r \in R} \phi(l, r), \quad \Delta_{\text{node}} = \phi(l^*, r^*).$$

S3) If $\Delta_{\text{node}} \leq \delta S_{\text{root}}$, then label node as a *leaf*.

S4) If $|l^*| \leq n_c$ or $|r^*| \leq n_c$, then label node as a *leaf*.

S5) If node is not yet a *leaf*, then delete node and split it by labeling l^* as node and r^* as node.

S6) If all nodes are labeled as leaves, then STOP. Else, move to next node. Go to S1.

The condition in Step 3 prevents the building of deep trees; this is in contrast to alternative methods that first build deep trees and later prune them.

The most time-consuming step in the tree-building procedure above is Step 2. To compute the values of (l^*, r^*) we need to consider every possible split. The most efficient way is to first sort the values. If the node has n elements, this operation takes $O(n \log n)$ time. OLS trees can then compute (l^*, r^*) in linear time using an online algorithm for updating both the mean and $\phi(l, r)$ across all possible splits. In order to achieve the same time complexity using the QT loss, we present our algorithm. Note that maximizing $\phi(l, r)$ is equivalent to minimizing $S_l + S_r$; we will use this to choose the best split.

III. Quantile Tree Algorithm

Consider the recursive splitting algorithm from the previous section using, as two cases, each of the two loss functions in (2). Aside from the initial calculation of S_{root},

the only real difference between the two cases will occur at Step 2. Here we provide an algorithm to solve the optimization problem in Step 2 with the QT loss function ρ_{QT}. We focus on finding the optimal split at one node for one predictor variable. Given such an algorithm, we obtain the best predictor among all predictors by direct comparison.

To start, assume we have two vectors X_0 and Y_0 that hold, respectively, the predictor and response at all nodes of the tree. We first find the sort order denoted by ord for the vector X_0 and obtain two vectors $X = X_0[ord]$ and $Y = Y_0[ord]$. In order to break the problem of computing S_l and S_r into an online algorithm, we will need to compute the τth quantile in an online fashion. What we mean by an online algorithm here is as follows: suppose we have already computed the value of S_l for a certain number of points. Now we add a new point into the left child. We would like to update S_l without having to sort all the values again and then computing the value of S_l from scratch.

A. Online Update of Quantiles

In general, computing the quantile in an online manner is considered expensive since it requires at least a partial sort of the data. However, in the present case, if we are able to obtain the new quantile for each insertion in logarithmic time, then the cost incurred overall is of the same order as sorting. This cost of sorting is incurred even when we use the OLS loss. We use this already incurred cost to update the quantile without increasing overall complexity.

The update of quantiles should be done with regards to efficiency, requiring the use of certain well-studied data structures. The following two requirements on a data structure are essential to our algorithm:

- Insertion/deletion time should be at most $O(\log n)$.
- Finding the max/min should be at most $O(\log n)$.

Any data structure that permits the above bounds will work for our algorithm. In the present work, we use heap-based priority queues. Priority queues built with heaps are a well-studied data structure, offering $O(1)$ access to the priority element and $O(\log n)$ time for insertion/deletion.

We use two heaps denoted by H_{low} and H_{high}, which correspond to max and min heaps, respectively. The max heap H_{low} allows access to the maximum element of the heap (denoted by $H_{\text{low}}[top]$) in $O(1)$ time and the min heap H_{high} allows access to the minimum element of the heap (denoted by $H_{\text{high}}[top]$) in $O(1)$ time. We denote by N_P the first P points in Y. We can easily obtain the τth quantile if we place $N_P^- = \lceil (N_P - 1)\tau \rceil$ points in H_{low} and the remaining $N_P^+ = N_P - N_p^-$ points in H_{high}. The τth quantile can now be computed as

$$q_P = H_{\text{low}}[top] + [H_{\text{high}}[top] - H_{\text{low}}[top]] \times (\tau(N_P - 1) - (\lceil (N_P - 1)\tau \rceil - 1)). \quad (3)$$

The value of the quantile q_P is a function of both the set of points P and the desired quantile τ, but the choice of quantile is fixed at the start of the algorithm and does not change. For notational clarity, we refer to q as a function of P only. Define $R = \{P \cup s\}$, where $s = Y[N+1]$ is the new point we wish to insert. In order to update the quantile when we insert s, we first check if $s \leq H_{\text{low}}$. If this is true, we insert s into H_{low}, else we insert it into H_{high}. After insertion, we need to ensure that there are exactly N_r^- points in H_{low} and N_r^+ points in H_{high} such that the τth quantile can still be obtained using (3). If this condition is violated, we need to pop the top value from H_{low} and insert it into H_{high}, or vice versa, to ensure the condition holds. This provides us with an online algorithm for updating the quantile when we insert one value at a time. Using (3) we can now obtain the updated quantile q_R. In this manner, starting with a single point $Y[1]$ inserted into H_{low}, we can update the quantile until we have inserted all the points.

B. Online Update of Quantile Loss Function

We now describe how we can update $\rho(x)$ when we insert a new point using the previous value. We denote by P^- and P^+ the sets of points in H_{low} and H_{high}, respectively. Equation (2) can be written as

$$QAD_{q_P,P} = \tau \sum_{P^+} (y_i - q_P) + (\tau - 1) \sum_{P^-} (y_i - q_P). \quad (4)$$

where $QAD_{q_P,P}$ corresponds to the general expression $\rho(x)$. We can now simplify this expression to obtain

$$\begin{aligned} QAD_{q_p,P} &= \tau \sum_{P^+} (y_i - q_p) + (\tau - 1) \sum_{P^-} (y_i - q_p) \\ &= \tau \sum_{P^+} y_i - N_p^+ \tau q_p + (\tau - 1) \sum_{P^-} y_i - N_p^-(\tau - 1) q_p \\ &= \tau \sum_{P^+} y_i + (\tau - 1) \sum_{P^-} y_i - q_p \left[N_p^-(\tau - 1) + N_p^+ \tau \right] \end{aligned}$$

Suppose we have already computed $QAD_{q_p,P}$ and we now wish to update the value when we insert the data point s as before with $R = \{P \cup s\}$. Consider first the case where the point is added to the left heap H_{low}. We have two possibilities to consider to obtain $QAD_{q_R,R}$.

Case 1. Addition of the point s does not cause any points to be moved from H_{low} to H_{high} in the update of the quantile. In this case we obtain

$$\begin{aligned} QAD_{q_R,R} &= \tau \sum_{R^+} (y_i - q_R) + (\tau - 1) \sum_{R^-} (y_i - q_R) \\ &= \tau \sum_{P^+} (y_i - q_R) + (\tau - 1) \sum_{P^-} (y_i - q_R) \\ &\quad + (\tau - 1)(s - q_R) \\ &= \tau \sum_{P^+} y_i + (\tau - 1) \sum_{P^-} y_i \\ &\quad - q_R \left[N_p^-(\tau - 1) + N_p^+ \tau \right] + (\tau - 1)(s - q_R) \\ &= QAD_{q_P,P} + (q_P - q_R) \left[N_p^-(\tau - 1) + N_p^+ \tau \right] \\ &\quad + (\tau - 1)(s - q_R). \end{aligned} \quad (5)$$

Case 2. When we add the point s to H_{low} we need to rebalance the heaps by moving $H_{\text{low}}[top]$ to the heap H_{high} in order to obtain the new quantile. Let ℓ denote the value of $H_{\text{low}}[top]$. We have, analogous to (5):

$$\begin{aligned} QAD_{q_R,R} &= \tau \sum_{R^+}(y_i - q_R) + (\tau - 1)\sum_{R^-}(y_i - q_R) \\ &= \tau \sum_{P^+}(y_i - q_R) + \tau(\ell - q_R) + (\tau - 1)\sum_{P^-}(y_i - q_R) \\ &\quad - (\tau - 1)(\ell - q_R) + (\tau - 1)(s - q_R) \\ &= \tau \sum_{P^+} y_i + (\tau - 1)\sum_{P^-} y_i - q_R\left[N_p^-(\tau - 1) + N_p^+\tau\right] \\ &\quad + \ell - q_R + (\tau - 1)(s - q_R) \\ &= QAD_{q_P,P} + (q_P - q_R)\left[N_p^-(\tau - 1) + N_p^+\tau\right] \\ &\quad + (\ell - q_R) + (\tau - 1)(s - q_R) \end{aligned} \tag{6}$$

If the point s to be added is greater than $H_{\text{low}}[top]$ then we insert it into H_{high}. Following a similar procedure, we obtain two update equations for each of two cases.

Case 1. No movement of points from H_{high} to H_{low} when updating the quantile:

$$\begin{aligned} QAD_{q_R,R} &= QAD_{q_P,P} \\ &\quad + (q_P - q_R)\left[(\tau - 1)N_p^- + \tau N_p^+\right] + \tau(s - q_R) \end{aligned} \tag{7}$$

Case 2. Move $\ell = H_{\text{high}}[top]$ to H_{low} when updating the quantile:

$$\begin{aligned} QAD_{q_R,R} &= QAD_{q_P,P} + (q_R - \ell) + \tau(s - q_R) \\ &\quad + (q_P - q_R)\left[(\tau - 1)N_p^- + \tau N_p^+\right] \end{aligned} \tag{8}$$

These four cases can be represented compactly using a single update equation. Let $\mathcal{I} = 1$ if a value has to be moved from the left heap to the right or vice versa and 0 otherwise. Let $\mathcal{J} = 1$ if we move a value from the left heap to the right and $\mathcal{J} = -1$ if we move a value from the right heap to the left. Let $\mathcal{K} = 1$ if the value was inserted into the left heap and 0 otherwise. Then all four cases can be summarized by

$$\begin{aligned} QAD_{q_R,R} &= QAD_{q_P,P} + (1 - \mathcal{K})\tau(s - q_R) \\ &\quad + (q_P - q_R)\left[(\tau - 1)N_p^- + \tau N_p^+\right] \\ &\quad + \mathcal{I}\mathcal{J}(\ell - q_R) + \mathcal{K}(\tau - 1)(s - q_R). \end{aligned} \tag{9}$$

Equation (9) provides us with a streaming update equation to update the value of the QT loss obtained from the left child, as we move one point at a time.

So far, we have only discussed the case of adding points sequentially into the heaps. To find (l^*, r^*) we need to remove points sequentially from the right child, starting with all the points. Deletion of floating-point values from heaps can cause trouble; we avoid this problem simply by inserting points in reverse order.

Given $\rho(x)$ for all splits, we obtain the optimal split (l^*, r^*) using a linear scan.

procedure LEFTQAD(Y, QAD, τ)
  initialize $H_{\text{low}}, H_{\text{high}}$
  set $QAD[1] = QAD[2] = 0.0$
  insert $Y[1]$ into H_{low}
  set $q_P = Y[1]$
  for $i = 2, 3, \ldots, N + 1$ **do**
    if $Y[i] \leq H_{\text{low}}[top]$ **then**
      insert $Y[i]$ into H_{low}
    else
      insert $Y[i]$ into H_{high}
    end if
    RebalanceHeaps($H_{\text{low}}, H_{\text{high}}, \tau$)
    q_R = FindQuantile($H_{\text{low}}, H_{\text{high}}, \tau$)
    $QAD[i] = QAD[i - 1]$+update using (9)
    set $q_P = q_R$
  end for
end procedure

procedure REBALANCEHEAPS($H_{\text{low}}, H_{\text{high}}, \tau$)
  $n_l \leftarrow$ size(H_{low})
  $n_r \leftarrow$ size(H_{high})
  $\psi \leftarrow \lceil((n_l + n_r - 1)\tau)$
  if $n_l > \psi$ **then**
    pop $H_{\text{high}}[top]$ and insert into H_{low}
  else if $n_l < \psi$ **then**
    pop $H_{\text{low}}[top]$ and insert into H_{high}
  end if
end procedure

procedure FINDQUANTILE($H_{\text{low}}, H_{\text{high}}, \tau$)
  $n_n \leftarrow$ size(H_{low}) + size(H_{high})
  $\psi \leftarrow \lceil((n_n - 1)\tau)\rceil$
  $q \leftarrow H_{\text{low}}[top] + (H_{\text{high}}[top] - H_{\text{low}}[top])(\tau(n_n - 1) - (\psi - 1))$
end procedure

Figure 1. Pseudocode for the online algorithm LEFTQAD that computes the QAD vector for the left child.

The pseudocode for computing the QAD vector for the left child is described in Figure 1. The input Y is the response vector, sorted based on the sort order of the predictor vector. QAD is initialized to zero for computing the left child. After each insertion we need to rebalance the heaps and obtain the new quantile values. Once rebalanced, we update the quantile value (see the paragraph after (3), and we also update the QAD via (9). To start, we set $QAD[0]$ and $QAD[1]$ equal to zero. In order to compute the complete QAD vector, i.e., including the right child, the algorithm is very similar, but we insert points in the reverse order and add the changes to the QAD vector. In the end, a linear scan identifies the best possible split.

Housing				CT Slices			
Size (N)	QT	tree	rpart	Size (N)	QT	tree	rpart
1.5K	0.0202	0.0136	0.0086	5K	2.5418	0.9476	0.5641
3.5K	0.0383	0.0270	0.0152	10K	5.1707	1.9200	1.1495
5.5K	0.0603	0.0446	0.0232	15K	7.8585	2.9546	1.7697
7.5K	0.0814	0.0596	0.0292	20K	10.6167	4.0779	2.5048
9.5K	0.1016	0.0744	0.0356	25K	13.4323	5.2316	3.2416
11.5K	0.1276	0.1009	0.0488	30K	16.2734	6.4152	4.0386
13.5K	0.1518	0.1129	0.0596	35K	19.1917	7.6866	4.8185
15.5K	0.1734	0.1339	0.0723	40K	22.1166	8.8849	5.6510
17.5K	0.2035	0.1563	0.0751	45K	25.0712	10.2914	6.4583
19.5K	0.2294	0.1710	0.0945	50K	28.0457	11.3384	7.2880

Table I
Average of raw times taken by all three algorithms for the housing and CT slices data set. (QT = QUANTILE TREE.)

C. Algorithmic Complexity

For each predictor variable, we need to first find the sort order for the points in the node. This operation takes $O(N \log N)$ time where N is the number of points. The worst case complexity for the LEFTQAD algorithm (see Figure 1) involves N insertions, N deletions for rebalancing and N reinsertions. This provides an upper bound of $O(N \log N)$ for both the left and right children, the same complexity as sorting. Hence the overall complexity is $O(N \log N)$.

IV. COMPUTATIONAL RESULTS

Here we give timing and accuracy results for our quantile tree algorithm and traditional OLS tree methods. For comparison, we use both the *tree* package and the *rpart* package [22] from R. Both of these packages build trees using the OLS loss. Our implementation of quantile regression trees is written in C++ using Armadillo [23]; we connect this to R using *RcppArmadillo* [24].

A. Scalability

In the preceding section, we showed that the worst-case asymptotic complexity of our quantile tree algorithm is $O(N \log N)$, the same as for OLS trees. In order to demonstrate this with practical examples, we test the algorithm on two data sets. The first is the California housing data set [25]. This data set has 20640 samples and 8 predictor variables. The second is the CT slices data set available at the UCI machine learning repository [26]. This data set has 53500 samples and 384 predictor variables. For both tests, we set the following common parameters: $n_s = 20$, $n_c = 7$, and $\delta = 0.01$. These correspond to the default settings for rpart. The rpart package can perform 10-fold cross-validation internally, but we turn it off for this study. We also set rpart to not search for surrogate variables; this cuts the time required to build trees.

For the California housing data set, we consider subsamples of the data without replacement of increasing sizes $N \in \{1500, 2500, \ldots, 19500\}$. For the CT slices data set, the sample sizes are chosen to be $N \in \{5000, 10000, \ldots, 50000\}$. We use all the predictor variables in both cases. For each sample size we perform 100 runs of the simulation and present the averaged results. All of the above tests were performed on a machine with an Intel Core i7 processor and 4GB of memory.

In Table I, we give all exact running times for this test. In Figure 2, we plot the running time of each algorithm against $N \log N$. The plot on the left (respectively, right) is for the California housing (respectively, CT slices) data set. The excellent fit of the least squares regression line to each corresponding set of running times indicates that our quantile tree algorithm achieves $O(N \log N)$ scaling in practice, not just in theory. This is a significant improvement from a typical $O(N^2)$ or $O(N^2 \log N)$ implementation of quantile regression trees.

Returning to Table I, we consider the ratio of the quantile tree running times to the competing algorithms' running times. We compute this ratio for each of the two competing algorithms (tree and rpart) and each of the two data sets, California housing (left) and CT slices (right). In this way, we see that the quantile tree running time is within a factor of 3 (respectively, 5) of the rpart running time for the housing (respectively, CT slices) data set. In particular, the ratio for each data set does not increase as a function of N, the number of training instances.

Taken together, these results confirm that our implementation of quantile regression trees has the same theoretical and practical scaling (in terms of running time) as OLS trees built using rpart and tree.

The tree package was written with the aim of simplicity while the rpart package is the standard optimized version for growing OLS trees. When both the rpart and quantile tree algorithms run, most of the running time is spent finding splits. Our algorithm requires two passes: once for computing the left QAD, and then another to obtain the complete QAD for the right split. OLS trees need only a single pass over the data and can compute the best split with $O(1)$ memory.

There is certainly future work to be done to reduce the implicit constant in front of the $N \log N$ time complexity for our quantile tree algorithm. At the same time, based on the above, we cannot expect the running time of our algorithm to be less than a factor of 2 than that of rpart.

B. Nonlinear Quantile Estimation

The goal of a quantile regression tree is to find a piecewise constant approximation of the nonlinear quantile function Φ_τ described in Section I. To our knowledge, the quantile loss has not been used before to build quantile trees. Therefore, a natural question that arises is: how well does our algorithm estimate the nonlinear quantile function Φ_τ?

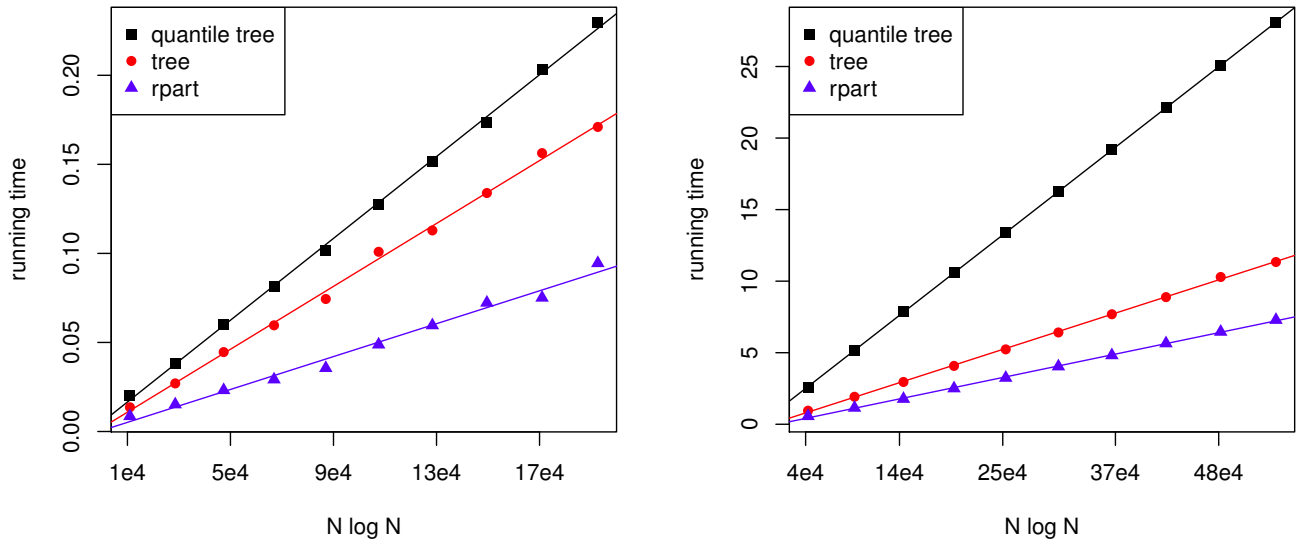

Figure 2. Time taken by all three algorithms plotted versus $N \log N$, where N is the number of instances in the respective data set. The plot on the left (respectively, right) is for the California housing (respectively, CT slices) data set. For each set of points we have plotted the least squares regression line. All points are excellent fits for their respective regression lines, indicating that the three algorithms have $O(N \log N)$ running times. The results show that our quantile tree algorithm features greatly improved scalability, as compared to a typical $O(N^2)$ or $O(N^2 \log N)$ implementation of quantile regression trees. Note that all points in this figure can be plotted using the data from Table I.

We describe here a relatively straightforward statistical test to provide a first response to this question. We start by considering two scalar random variables $\xi \in [0, 1]$ and $\eta \in [0, 1]$ with joint probability density function (PDF)

$$f(\xi, \eta) = C(1 - (\xi - 1/10)^2 - (6\eta/5 - 1/10)^2)^4. \quad (10)$$

Here C is a normalization constant that can be determined through a simple but tedious calculation. We have crafted this PDF so that the conditional quantile function (i) can be computed exactly and (ii) is a genuinely nonlinear curve the changes noticeably as a function of τ, the quantile value.

Our first step is to use a Metropolis-Hastings algorithm to generate 16 million samples from the joint PDF f. To generate the samples, we use a burn-in period of 1000, generate 320 million samples after burn-in, and then collect every 20th sample. In this way, we generate samples with reasonably low autocorrelation.

We treat each sample of (ξ, η) as a separate row (x, y) of a data matrix. We then use our quantile tree algorithm to construct $\hat{y} = \Phi_\tau(\hat{x})$ for each of the following five quantiles: $\tau \in \{0.05, 0.25, 0.5, 0.75, 0.95\}$. For all tests, we use the following parameters: $n_s = 20$, $n_c = 10$, and $\delta = 10^{-5}$.

Returning to (10), we use calculus and the laws of probability to calculate the marginal PDF $f(\xi)$, the conditional PDF $f(\eta|\xi)$, and the conditional CDF $F(\eta|\xi)$. For fixed τ and this particular conditional CDF, it is then possible to solve $F(\Phi|\xi) = \tau$ for Φ, yielding an exact conditional quantile function. We carry out this exact symbolic calculation using Mathematica.

In the left half of Figure 3, we plot the exact nonlinear quantile functions in black and the piecewise constant approximations (provided by our quantile tree algorithm) in red. The five black (and corresponding red) curves in the plot correspond, from bottom to top, to the five increasing values of τ from 0.05 to 0.95. In this case, we see good agreement between exact and approximate nonlinear quantile functions.

In the right half of Figure 3, we repeat the test but this time with a modified, "noisy" y. Specifically, we add Laplace distributed noise added to y, and then threshold values of y to ensure they are within the interval $[0, 1]$. The PDF of the noise we used is $(2b)^{-1} \exp(-|y|/b)$ with $b = 0.1$. This noise has a noticeably heavier tail than the Gaussian noise with the same standard deviation. For the noisy case, we see that the middle three quantile curves ($\tau \in \{0.25, 0.5, 0.75\}$) are still computed reasonably well, but that larger errors have crept into the extreme quantiles ($\tau \in \{0.05, 0.95\}$).

Overall, in both cases, the quantile tree algorithm is able to recover the broad qualitative shape of the nonlinear quantile curves. Certainly we can see strong evidence in the left plot that the quantile tree algorithm produces fairly accurate approximations to the nonlinear quantile functions corresponding to the *empirical* distribution of the data. In

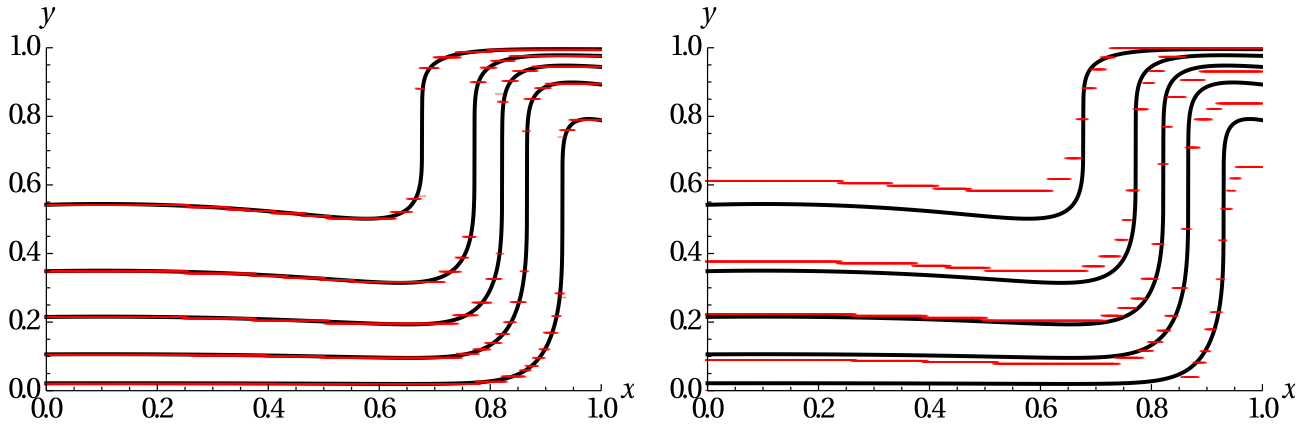

Figure 3. Here we compare exact nonlinear quantile functions (black) versus piecewise constant approximations computed using the quantile tree algorithm (red). The exact quantile functions, which are the same in both left and right plots, are computed symbolically from a known joint distribution. From bottom to top, the curves correspond to $\tau = 0.05, 0.25, 0.5, 0.75, 0.95$. In the left plot, the quantile tree algorithm is applied directly to samples from the joint distribution. In the right plot, the samples from the joint distribution are corrupted with Laplace noise and then thresholded before we apply the quantile tree algorithm.

the left plot, this empirical distribution coincides with the known PDF (10), fostering the agreement between red and black curves. The present results motivate the idea that quantile regression trees may enable one to develop richer, more informative models from big data sets than would be possible using OLS regression trees. In future work, we will explore further theoretical explanation of these ideas.

Note that better results may be possible in the current algorithm via parameter tuning—the parameters given above, including the 10^{-5} minimum reduction in node deviance—are simply our unoptimized first guesses. Finally, we note that the algorithm has no problem running on data sets with 16 million instances. In the present implementation, we are limited only by available RAM.

C. Accuracy

In this section, we contrast the use of LAD trees (the $\tau = 0.5$ case of quantile trees) against traditional OLS trees. Different loss functions lead to different trees and different predictions; here we sketch two factors that may influence appropriate usage of LAD trees.

In what follows, we define the true and predicted values by y_i and $\hat{y}_i$, respectively. We will use mean-squared error (MSE), i.e., $N^{-1}\sum_i (y_i - \hat{y}_i)^2$, and mean absolute deviation (MAD), i.e., $N^{-1}\sum_i |y_i - \hat{y}_i|$.

We build models using our quantile tree algorithm and rpart, and we estimate the test error (in both MSE and MAD metrics) using a standard 10-fold cross-validation procedure. The internal cross-validation procedure of rpart is turned off so that we may use the same folds for both the methods.

The data used to perform the tests include:

- Wine [27]: This consists of two data sets both with 11 predictors. The red wine data set has 1599 samples, and the white wine data set has 4898 samples. The real-valued response, wine quality, is between 1 and 10.
- Crime [28]: The complete data set consists of 1994 samples with 128 predictors. Eliminating columns with missing values, we reduce the number of predictors to 96. The real-valued response is a measure of crime rate that lies between 0 and 1.
- CA housing: This is the same data set used in Section IV-A. The response is log(median house price).

We use the parameters $n_s = 20$, $n_c = 7$, and $\delta = 0.01$ in our algorithm and analogous values for rpart. All results are averaged over 100 runs where each run performs 10-fold cross-validation to estimate test error.

The results are tabulated in Table II. For each data set, LAD trees outperform OLS trees in the MAD metric. Similarly, for each data set, OLS trees outperform LAD trees in the MSE metric. These results show that our initial objective, i.e., whether it is more appropriate to minimize MAD or MSE for our particular problem, should play a role in deciding which method to use. Table II also shows that, on each data set, the LAD trees are smaller in size than the OLS trees. Clearly, further work is necessary to explore these relationships.

V. Conclusion

We have presented what, to our knowledge, is the fastest algorithm to date for fitting a regression tree using the

	LAD trees			OLS trees		
	MAD	MSE	Size	MAD	MSE	Size
Red wine	0.4843	0.5670	7.68	0.5304	0.4619	19.91
White wine	0.5275	0.6553	9.00	0.6041	0.5817	11.84
Crime	0.1022	0.0340	22.39	0.1070	0.0308	34.54
Housing	0.2808	0.1382	24.03	0.2857	0.1370	27.11

Table II
Accuracy results in MAD and MSE metrics, for LAD and OLS trees.

quantile loss function (2). Computational tests verify that the algorithm runs in $O(N \log N)$ time, within a constant factor of the time required for regression trees that use the standard OLS loss function. We have verified the algorithm's ability (i) to uncover actual nonlinear quantile functions, and (ii) to produce trees with better test set error (as measured by the MAD metric) than OLS regression trees.

REFERENCES

[1] L. Breiman, J. Friedman, C. J. Stone, and R. A. Olshen, *Classification and Regression Trees*. Chapman & Hall/CRC, 1984.

[2] R. Koenker, *Quantile Regression*. Cambridge University Press, 2005.

[3] C. Perlich, S. Rosset, R. D. Lawrence, and B. Zadrozny, "High-quantile modeling for customer wallet estimation and other applications," in *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, ser. KDD '07, 2007, pp. 977–985.

[4] H. S. Bhat and D. Zaelit, "Forecasting retained earnings of privately held companies with PCA and L1 regression," *Appl. Stoch. Model. Bus.*, vol. 30, no. 3, pp. 271–293, 2014.

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

[6] W.-Y. Loh, "Fifty Years of Classification and Regression Trees," *Int. Stat. Rev.*, vol. 82, no. 3, pp. 329–348, 2014.

[7] P. Chaudhuri and W.-Y. Loh, "Nonparametric estimation of conditional quantiles using quantile regression trees," *Bernoulli*, vol. 8, no. 5, pp. 561–576, 2002.

[8] Y. Chang, "Multi-step quantile regression tree," *J. Stat. Comput. Sim.*, vol. 84, no. 3, pp. 663–682, 2014.

[9] G. Galimberti, M. Pillati, and G. Soffritti, "Notes on the robustness of regression trees against skewed and contaminated errors," in *New Perspectives in Statistical Modeling and Data Analysis*. Springer-Verlag, 2011, pp. 255–263.

[10] A. J. Cannon, "Quantile regression neural networks: Implementation in R and application to precipitation downscaling," *Comput. Geosci.*, vol. 37, no. 9, pp. 1277–1284, 2011.

[11] R. Koenker, "Additive models for quantile regression: Model selection and confidence bandaids," *Braz. J. Prob. Stat.*, vol. 25, no. 3, pp. 239–262, 2011.

[12] H.-S. Oh, T. C. M. Lee, and D. W. Nychka, "Fast nonparametric quantile regression with arbitrary smoothing methods," *J. Comput. Graph. Stat.*, vol. 20, no. 2, pp. 510–526, 2011.

[13] V. Spokoiny, W. Wang, and W. Karl Hrdle, "Local quantile regression," *J. Stat. Plan. Infer.*, vol. 143, no. 7, pp. 1109–1129, 2013.

[14] Y.-H. Zhou, Z.-X. Ni, and Y. Li, "Quantile Regression via the EM Algorithm," *Commun. Stat. Simulat.*, vol. 43, no. 10, pp. 2162–2172, Nov. 2014.

[15] N. Meinshausen, "Quantile regression forests," *J. Mach. Learn. Res.*, vol. 7, pp. 983–999, 2006.

[16] P. Bloomfield and W. L. Steiger, *Least Absolute Deviations*. Birkhäuser Boston, 1983.

[17] T. E. Dielman, "Least absolute value regression: recent contributions," *J. Stat. Comput. Sim.*, vol. 75, no. 4, pp. 263–286, 2005.

[18] G. Ciuperca, "Estimating nonlinear regression with and without change-points by the LAD method," *Annals of the Institute of Statistical Mathematics*, vol. 63, no. 4, pp. 717–743, 2011.

[19] ——, "Penalized least absolute deviations estimation for nonlinear model with change-points," *Stat. Pap.*, vol. 52, no. 2, pp. 371–390, 2011.

[20] L. F. R. A. Torgo, "Inductive learning of tree-based regression models," Ph.D. dissertation, Universidade do Porto, 1999.

[21] B. Ripley, *tree: Classification and regression trees*, 2015, R package version 1.0-36. [Online]. Available: http://CRAN.R-project.org/package=tree

[22] T. Therneau, B. Atkinson, and B. Ripley, *rpart: Recursive Partitioning and Regression Trees*, 2015, R package version 4.1-10. [Online]. Available: http://CRAN.R-project.org/package=rpart

[23] C. Sanderson, "Armadillo: An open source C++ linear algebra library for fast prototyping and computationally intensive experiments," NICTA, Tech. Rep., 2010.

[24] D. Eddelbuettel and C. Sanderson, "RcppArmadillo: Accelerating R with high-performance C++ linear algebra," *Comput. Stat. Data. An.*, vol. 71, pp. 1054–1063, March 2014.

[25] R. K. Pace and R. Barry, "Sparse spatial autoregressions," *Stat. Probabil. Lett.*, vol. 33, no. 3, pp. 291–297, 1997.

[26] A. Frank and A. Asuncion, "UCI Machine Learning Repository," 2010. [Online]. Available: http://archive.ics.uci.edu/ml

[27] P. Cortez, J. Teixeira, A. Cerdeira, F. Almeida, T. Matos, and J. Reis, "Using data mining for wine quality assessment," in *Discovery Science*, ser. Lecture Notes in Computer Science. Springer-Verlag, 2009, vol. 5808, pp. 66–79.

[28] M. Redmond and A. Baveja, "A data-driven software tool for enabling cooperative information sharing among police departments," *Eur. J. Oper. Res.*, vol. 141, no. 3, pp. 660–678, 2002.