Image Recognition Technology Based on Tree Boosting Methods

Rongrong Su
Statistics and Actuarial Science, University of Waterloo
Waterloo, Canada
r2sue@uwaterloo.ca

Abstract

Tree boosting is a powerful and widely used method of machine learning. In this project, I introduce three boosting algorithms, which are AdaBoost algorithm, Gradient Boosting algorithm and XGBoost. Then based on fashion-mnist dataset, I implement these three algorithms and compare their performance. It shows that the accuracy of test set of XGBoost algorithm is highest.

Introduction

Image recognition is used to perform a large number of machine-based visual tasks, such as labeling the content of images with meta_tags, performing image content search and guiding autonomous robots, self-driving cars and accident avoidance systems. While human and animal brains recognize objects with ease, computers have difficulty with the task because of lack of consciousness. The computers recognize image by recognizing features i.e. color (which is also known as pixel). In the hotdog example, the developers would have fed a computer thousands of pictures of hotdogs. The AI then develops a general idea of what a picture of a hotdog should have in it. When we feed it an image of something, it compares every pixel of that image to every picture of a hotdog it's ever seen. If the input meets a minimum threshold of similar pixels, the AI declares it a hotdog.
The most two popular machine learning algorithms for image recognition (image classification) are neural network method and tree method. Next I will start with two basic definitions.

Supervised Learning

Given a set of N training example of the form $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}$ such that x_i is the feature vector of the i^{th} example and y_i is its label (i.e., class). An algorithm to learn the mapping function from the input to the output is

$$\mathrm{Y} = f(\mathrm{X})$$

Model is how to make prediction $\hat{y}_i$ given x_i. For example, the linear model has the following form.

$$\hat{y}_i = \sum_j w_j x_{ij}$$

The objective function is the function that we what to optimize to get the best parameter.

$$\mathrm{Obj}(\Theta) = \mathrm{L}(\Theta) + \Omega(\Theta)$$

where $\mathrm{L}(\Theta)$ is training loss which measures how well model fits on training data and $\Omega(\Theta)$ is regularization which measures complexity of model.

- Loss on training data: $\mathrm{L}(\Theta) = \sum_{i=1}^{n} l(y_i, \hat{y}_i)$
  - Square loss: $l(y_i, \hat{y}_i) = (y_i, \hat{y}_i)^2$
  - Logistic loss: $l(y_i, \hat{y}_i) = y_i \ln\left(1 + e^{-\hat{y}_i}\right) + (1 - y_i)\ln\left(1 + e^{\hat{y}_i}\right)$

Optimizing training loss encourages predictive models, which means fitting well in training data at least get us to close to training data which is hopefully close to the underlying distribution.

- Regularization: $\Omega(\mathrm{w}) = \mathrm{f}(\mathrm{w})$
  - L2 norm: $\Omega(\mathrm{w}) = \lambda||w||^2$
  - L1 norm: $\Omega(\mathrm{w}) = \lambda||w||_1$

Optimizing regularization encourages simple models. A simpler model avoids overfitting and tends to have smaller variance in future predictions, making prediction stable.

Regression Tree

Regression tree is a kind of decision trees whose target variable can take continuous values (typically real number)

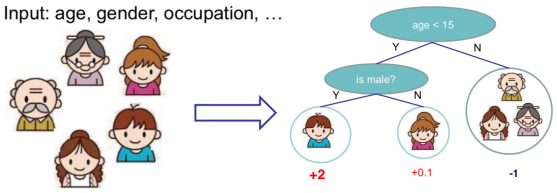

Figure 1: Regression Tree Model. The +2, +0.2, -1 is the prediction score of each leaf.

Adaptive Boosting

Boosting algorithm

A weak learner is defined to be a classifier that is only slightly correlated with the true classification (it can label examples better than random guessing). In contrast, a strong learner is a classifier that is arbitrarily well-correlated with the true classification. Boosting is based on

the theorem proposed by Robert Schapire in 1990:" a set of weak learners can create a single strong learner". Boosting refers to the process of turning a weak learner into a strong learner. More specifically, most boosting algorithms consist of iteratively learning weak classifiers with respect to a distribution and adding them to a final strong classifier. So, boosting is an ensemble technique in which the predictors are not made independently, but sequentially.

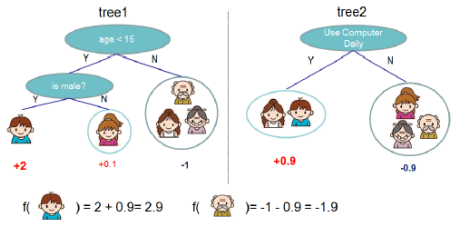

Figure 2: Tree Ensemble Model. The final prediction for a given example is the sum of predictions from each tree.

Adaptive Boosting

Adaptive algorithm fits a weak classifier to weighted versions of the data iteratively. At each iteration, the data is reweighted such that misclassified data points receive larger weights. Suppose we have a data set $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n)\}, where\ x_i \in \mathbb{R}^d, y_i \in \{-1,1\}$ and a set of weak classifiers $\{f_1, f_2, \dots, f_T\}, where\ f_j(x_i) \in \{-1,1\}$. After the $(t-1)^{th}$ iteration our boosted classifier is a linear combination of the weak classifiers of the form:

$$\hat{y}_i^{(t-1)} = \theta_1 f_1(x_i) + \theta_2 f_2(x_i) + \cdots + \theta_{t-1} f_{t-1}(x_i)$$

The resulting model can be written as flowing.

$$F(x) = \hat{y} = \hat{y}^{(T-1)} + \theta_T f_T(x) = \sum_{t=1}^{T} \theta_t f_t(x)$$

The loss function AdaBoost uses is exponential loss function.

$$\mathrm{L}(y, \hat{y}) = \exp(-y * \hat{y}) = \exp(-y * \hat{y})$$

We define the total error E of all data points as the sum of exponential loss on each data point, given as follows

$$\mathrm{E} = \sum_{i=1}^{N} e^{-y_i \hat{y}_i^{(t)}}$$

Letting $w_i^1 = 1\ and\ w_i^{(t)} = e^{-y_i \hat{y}_i^{(t-1)}}$ for m > 1 (w_i is the weight for each point) we have

$$\mathrm{E} = \sum_{i=1}^{N} w_i^{(t)} e^{-y_i \theta_t \hat{y}_i^{(t)}}$$

We can split this summation between those data points that are correctly classified by $f(x)$ (so $y_i \hat{y}_i^{(t)} = 1$) and those that are misclassified (so $y_i \hat{y}_i^{(t)} = -1$). Then the error E has the form as follows.

$$\mathrm{E} = \sum_{y_i = k_{(t)}(x_i)} w_i^{(t)} e^{-\theta_t} + \sum_{y_i \neq k_{(t)}(x_i)} w_i^{(t)} e^{\theta_t}$$

To determine the desired weight θ_t that minimizes E with the $\hat{y}_i^{(t)} = f_{(t)}$, we differentiate error E.

$$\frac{\partial \mathrm{E}}{\partial \theta_t} = \frac{\partial (\sum_{y_i = f_{(t)}(x_i)} w_i^{(t)} e^{-\theta_t} + \sum_{y_i \neq f_{(t)}(x_i)} w_i^{(t)} e^{\theta_t})}{\partial \theta_t}$$

Setting this to zero and solving for θ_t yields.

$$\theta_t = \frac{1}{2} \ln \left(\frac{\sum_{y_i = f_{(t)}(x_i)} w_i^{(t)}}{\sum_{y_i \neq f_{(t)}(x_i)} w_i^{(t)}} \right)$$

Here we define l_t as follows.

$$l_t = \frac{\sum_{y_i \neq f_{(t)}(x_i)} w_i^{(t)}}{\sum_{i=1}^{N} w_i^{(t)}}$$

So we can rewrite θ_t as follows.

$$\theta_t = \frac{1 - l_t}{l_t}$$

The final classifier is $F(x) = sign(\sum_{t=1}^{T} \theta_t f_t(x))$
The exact AdaBoost algorithm is shown in Alg.1

Algorithm 1: AdaBoost Algorithm

for i=1 to N **do**
  $w_i^1 \leftarrow 1$
end
for t=1 to T **do**
  find $f_t(x)$ that minimize e_t
    $f_t(x) = argmin\ e_t$
    $e_t = \frac{\sum_{i=1}^{N} w_i I(y_i \neq f_t(x_i))}{\sum_{i=1}^{N} w_i^{(t)}}$
  $\theta_t = \frac{1}{2} \log \left(\frac{1 - e_t}{e_t} \right)$
  for i=1 to N **do**
    $w_i^{(t+1)} = \frac{w_i^{(t)}}{z_t} e^{\theta_t I(y_i \neq f_t(x_i))}$ where $z_t = \sum_{i=1}^{N} w_i^{(t)} e^{\theta_t I(y_i \neq f_t(x_i))}$
  end
end
$f(x_i) = \mathrm{sign}(\sum_{t=1}^{T} \theta_t f_t(x_i))$

The next is a example of implementation of AdaBoost algorithm.

I(N=10)	1	2	3	4	5	6	7	8	9	10
X(feature)	0	1	2	3	4	5	6	7	8	9
Y(label)	1	1	1	-1	-1	-1	1	1	1	-1

Table1. Information of data

We define weight distribution $D^1 = (w_1^1, w_2^1, \dots, w_{10}^1)$ and assume each point has the same weight: $w_i^1 = 0.1, i = 1, 2, , 10$

Now we train the first weak classifier. When the split point is 2.5 the e_t is the minimum. So, the first weak classifier is $f_1(x)$ and the error e_1 is 0.3.

$$f_1(x) = \begin{cases} 1, & x < 2.5 \\ -1, & x > 2.5 \end{cases}$$

The weight of $f_1(x)$ is $\theta_1 = \frac{1}{2}\log\left(\frac{1-e_1}{e_1}\right) = 0.4236$.

The new weight vector is calculated by

$$w_i^{(2)} = \frac{w_i^{(1)}}{z_1} e^{\theta_1 I(y_i \neq f_1(x_i))}, \qquad i = 1,2, \dots, 10$$

$$D^2 = (0.0715, 0.0715, 0.0715, 0.0715, 0.0715, 0.0715, 0.1666, 0.1666, 0.1666, 0.0715)$$

We notice that the weight of points which are misclassified increases.

After the first iteration, the strong classifier is given by

$$f(x) = sign\big(0.4236 f_1(x)\big)$$

After the third iteration, the strong classifier is given by

$$F^3(x) = sign\big(0.4236 f_1(x) + 0.6496 f_2(x) + 0.7514 f_3(x)\big)$$

The drawback of AdaBoost is that it is easily defeated by noisy data, the efficiency of the algorithm is highly affected by outliers as the algorithm tries to fit every point perfectly. And AdaBoost, it can only solve this equation for the exponential loss function under the constraint that $f_t(x)$ only outputs -1 or 1. While Gradient Boosting and XGBoost can be viewed as two general boosting algorithms that solve the equation approximately for any suitable loss function.

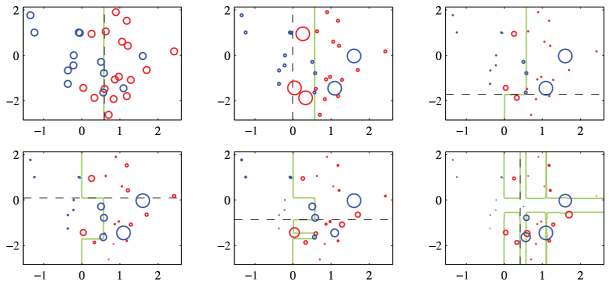

Figure 3: Illustration of the steps of AdaBoost. The decision boundary is shown in green for each step, and the decision stump for each step shown as a dashed line. The results are shown after 1, 2, 3, 6, 10, and 150 steps of AdaBoost. (Figure from Pattern Recognition and Machine Learning by Chris Bishop.)

Gradient Boosting

Steepest descent chooses the direction in which the function is most rapidly decreasing. We would like to do the same thing but here our solution must be a tree. Also, importantly, we don't want to simply minimize loss on the training set but generalize to new data. A potential solution is to induce a tree at the t^{th} iteration whose predictions $f_t(x)$ are as close as possible to the negative gradient. The algorithm is given by Alg.2

Algorithm 2: Gradient Boosting Algorithm

Initialize $f_0(x) = \underset{\gamma}{\operatorname{argmin}} \sum_{i=1}^{N} L(y_i, \gamma)$ where $L(y_i, \gamma)$ is the loss function

for tree t=1 to T **do**

  for data point i=1 to N **do**

    compute pseudo residuals $r_{it} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{t-1}(x)}$ $for\ i = 1, 2, \dots, N$

  end

  Fit a regression tree to the target r_{it} giving terminal regions $R_{jt}, j = 1, 2, \dots, J_t$

  for j=1 to J_t **do**

    compute $\gamma_{jt} = \underset{\gamma}{\operatorname{argmin}} \sum_{x_i \in R_{jt}} L(y_i, f_{t-1}(x_i) + \gamma)$

  end

  Update strong classifier $f_t(x) = f_{t-1}(x) + \sum_{j=1}^{J_t} \gamma_{jt} I(x \in R_{jt})$

end

The final $F(x) = f_T(x) = f_0(x) + \sum_{T=1}^{T} \sum_{j=1}^{J_t} \gamma_{jt} I(x \in R_{jt})$

XGBOOST

For a given data set $\{(x_1, y_1), (x_2, y_2), \dots, (x_n, y_n), where\ x_i \in \mathbb{R}^d, y_i \in \mathbb{R}\}$ a tree ensemble model (shown in Fig.2) is given as following.

$$\hat{y}_i = \phi(x_i) = \sum_{k=1}^{K} f_k(x_i), f_k \in \mathcal{F}$$

where $\mathcal{F} = \{f(x) = w_{q(x)}\}(q: \mathbb{R}^m \to T, w \in \mathbb{R}^T)$ is the space of regression trees. Here q represents the structure of each tree that maps an example to the corresponding leaf index and T is the number of leaves in the tree. And we use w_i to represent score on i^{th} leaf.

To learn the set of functions used in the model, we minimize the following regularized objective.

$$L(\phi) = \sum_i l(y_i, \hat{y}_i) + \sum_k \Omega(f_k) \text{ where } \Omega(f) = \gamma T + \frac{1}{2}\lambda ||w||^2 \quad (1)$$

$l(\hat{y}_i, y_i)$ is a differentiable convex loss function that measures the difference between the prediction and the target. And $\Omega(f_k)$ penalizes the complexity of the model (i.e., the regression tree functions).

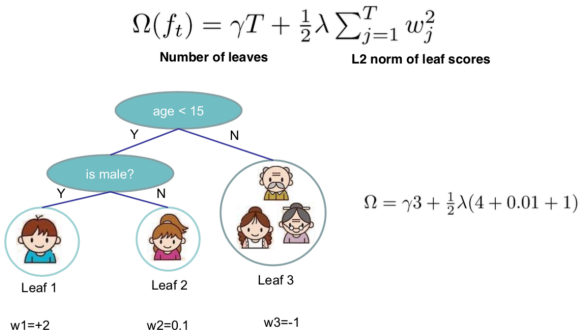

Figure 4: Model Complexity Calculation.

Instead, the model is trained in an additive manner. We rewrite Eq. (1) as follows

$$L^{(t)} = \sum_{i=1}^{N} l(y_i, \hat{y}_i^{(t-1)} + f_t(x_i)) + \Omega(f_t)$$

By Taylor expansion

$$f(x + \Delta x) \cong f(x) + f'(x)\Delta x + \frac{1}{2} f''(x)\Delta x^2$$

So Eq.(1) can be rewritten as follows.

$$L^{(t)} = \sum_{i=1}^{N} [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \Omega(f_t)$$

Here $g_i = \partial_{\hat{y}_i^{(t-1)}} L(y_i, \hat{y}_i^{(t-1)})$ $h_i = \partial^2_{\hat{y}_i^{(t-1)}} L(y_i, \hat{y}_i^{(t-1)})$

Since the constant is trivial We can remove the constant terms to obtain the following simplified objective function at step t.

$$L^{(t)} = \sum_{i=1}^{N} [g_i f_t(x_i) + \frac{1}{2} h_i f_t^2(x_i)] + \gamma T + \frac{1}{2} \sum_{j=1}^{T} \lambda ||w_j||^2$$

$$= \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) w_j^2] + \gamma T$$

where $I_j = \{i | q(x_i) = j\}$ is the instance set of leaf j

For a fixed structure $q(x)$, the optimal weight w_j^* of leaf j is given by

$$w_j^* = \frac{\sum_{i \in I_j} g_i}{\sum_{i \in I_j} h_i + \lambda}$$

And the corresponding optimal value is

$$L^{(t)}(q) = -\frac{1}{2} \sum_{j=1}^{T} \frac{(\sum_{i \in I_j} g_i)^2}{\sum_{i \in I_j} h_i + \lambda} + \gamma T \quad (6)$$

Eq. (6) is the scoring function used to measure the quality of a tree structure. Fig. 5 illustrates how this score can be calculated.

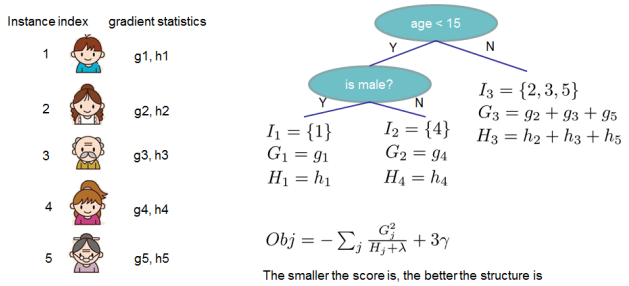

Figure 5: Structure Score Calculation. We only need to sum up the gradient and second order gradient statistics on each leaf, then apply the scoring formula to get the quality score.

But there can be infinite possible tree structures q. In practice, we grow the tree greedily.

- Start from tree with depth 0
- For each leaf node of tree, try to add a split. The change of objective after adding the split is

$$Gain = \frac{1}{2} \left[\frac{(\sum_{i \in I_L} g_i)^2}{\sum_{i \in I_L} h_i + \lambda} + \frac{(\sum_{i \in I_R} g_i)^2}{\sum_{i \in I_R} h_i + \lambda} - \frac{(\sum_{i \in I} g_i)^2}{\sum_{i \in I} h_i + \lambda} \right] - \gamma$$

A split finding algorithm enumerates over all the possible splits on all the features is call exact greedy algorithm. The exact greedy algorithm is shown in Alg. 3.

Algorithm 3: Exact Greedy Algorithm for Split Finding

Input: I instance set of current node
Input: d feature dimension
$gain \leftarrow 0$
$G \leftarrow \sum_{i \in I} g_i$, $H \leftarrow \sum_{i \in I} h_i$
for k=1 **to** m **do**
  $G_L \leftarrow 0, H_L \leftarrow 0$
  for j in sorted (I, by x_{jk}) **do**
    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$
    $G_R \leftarrow G - G_L$, $H_R \leftarrow H - H_L$
    score $\leftarrow$ max ($score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda}$)
  end
end
Output: Split with max score

The time complexity growing a tree of depth K with greedy algorithm is $O(ndK \log(n))$, which is too big. And it is impossible to efficiently do so when the data does not fit entirely into memory. So, an approximate algorithm is needed. And the algorithm is shown in Alg. 4.

Algorithm 4: Approximate Algorithm for Split Finding

for k=1 **to** m **do**

  Propose $S_k = \{s_{k1}, s_{k2}, \ldots, s_{kl}\}$ by percentiles on feature k.

  Proposal can be done per tree(global), ore per split(local).

end

for k=1 **to** m **do**

  $G_{kv} \leftarrow= \sum_{j \epsilon \{s_{k,v} \geq x_{jk} > s_{k,v-1}\}} g_i$

  $H_{kv} \leftarrow= \sum_{j \epsilon \{s_{k,v} \geq x_{jk} > s_{k,v-1}\}} h_i$

end

Follow same step as in previous section to find max score only among proposed splits.

To summarize, the algorithm first propose candidate splitting points according to percentiles of features distribution. And then the algorithm maps the continuous features into buckets split by these candidate points, aggregates the statistics and find the best solution among proposals base on the aggregated statistics. So, using less data the approximate algorithm can find the split points as accurate as those found by greedy algorithm. Fig. 6 shows the result of comparison of test AUC convergence on Higgs 10M dataset with four methods.

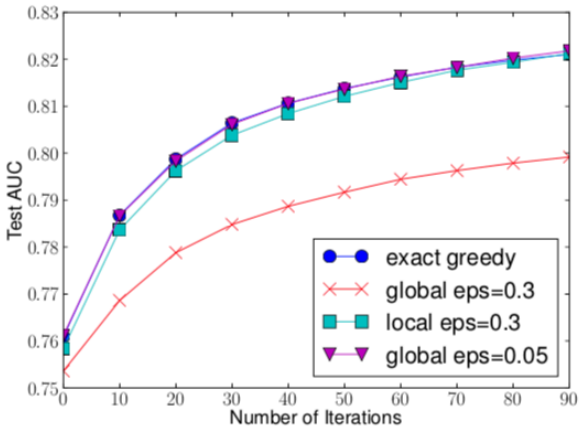

Figure 6: Comparison of test AUC convergence on Higgs 10M dataset. The eps parameter corresponds to the accuracy of the approximate sketch. This roughly translates to 1 / eps buckets in the proposal. We find that local proposals require fewer buckets, because it refine split candidates.

Sparsity-aware algorithm

In order to take care of missing values, frequent zero entries and artifacts of feature engineering (those are also known as sparsity pattern), the author propose to add a default direction in each tree node. An instance is given by Fig.7 and the algorithm of finding default direction is given by algorithm 5.

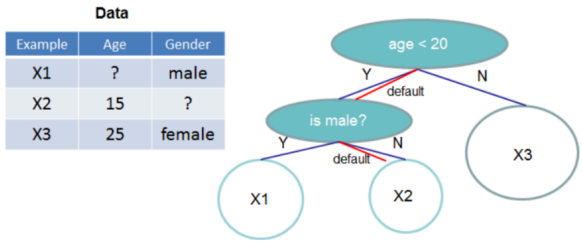

Figure 7: Tree structure with default directions. An example will be classified into the default direction when the feature needed for the split is missing.

Algorithm 5: Sparsity-aware Split Algorithm

Input: I instance set of current node

Input: $I_k = \{i \epsilon I | x_{ik} \neq missing\}$

Input: d feature dimension

Also, applies to the approximate setting, only collect statistics of non-missing ectries into buckets

$gain \leftarrow 0$

$G \leftarrow \sum_{i \epsilon I} g_i$, $H \leftarrow \sum_{i \epsilon I} h_i$

for k=1 **to** m **do**

  // enumerate missing value goto right

  $G_L \leftarrow 0, H_L \leftarrow 0$

  for j *in sorted* (I_k *ascent order by* x_{jk}) **do**

    $G_L \leftarrow G_L + g_j$, $H_L \leftarrow H_L + h_j$

    $G_R \leftarrow G - G_L$, $H_R \leftarrow \mathrm{H} - H_L$

    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

  end

  // enumerate missing value goto left

  $G_R \leftarrow 0, H_R \leftarrow 0$

  for j *in sorted* (I, *decent order by* x_{jk}) **do**

    $G_R \leftarrow G_R + g_j$, $H_L \leftarrow H_L + h_j$

    $G_R \leftarrow G - G_L$, $H_R \leftarrow \mathrm{H} - H_L$

    $score \leftarrow \max(score, \frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G^2}{H+\lambda})$

  end

end

Output: Split and default directions with max gain

XGBoost handles all sparsity patterns in a unified way. And this algorithm makes the computation complexity linear to number of non-missing entries. Fig. 6 shows the comparison of sparsity aware and a naive implementation on a sparse dataset.

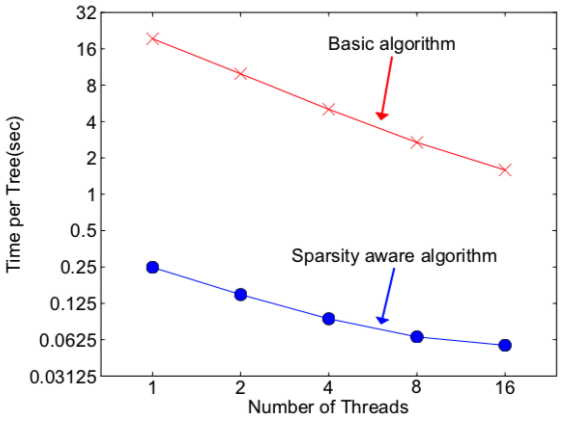

Figure 7: The sparsity aware algorithm is more than 50 times faster than the naive version that does not take sparsity into consideration.

The XGBoost Advantage

- Regularization
  Standard Gradient Boosting Method has no regularization like XGBoost, therefore it also helps to reduce overfitting.
- Parallel Processing
  XGBoost implements parallel processing and is blazingly faster as compared to Gradient Boosting.
- Handling Missing Values
  With sparsity-aware algorithm XGBoost tries different things to fit missing values as it encounters a missing value on each node and learn which path to take for missing values in futures.
- Tree Pruning
  A gradient boosting method would stop splitting a node when it encounters a negative loss in the split. Thus, it is more of a greedy algorithm. XGBoost on the other hand makes splits up to the max depth specified and then start pruning the tree. This also help to avoid overfitting.
- Built-in Cross-Validation
  XGBoost allows users to run a cross-validation at each iteration of the boosting iteration process and thus it is easy to get the exact optimum number of boosting iteration in a single run. Whereas in simple Gradient Boosting we have to run a grid search and only a limited value can be tested.

Empirical evaluation

The data set I used is fashion-mnist. Fashion-MNIST is a dataset of Zalando's article images—consisting of a training set of 60,000 examples and a test set of 10,000 examples. Each example is a 28x28 grayscale image, associated with a label from 10 classes. And the labels are 0 T-shirt/top, 1 Trouser, 2 Pullover, 3 Dress, 4 Coat, 5 Sandal, 6 Shirt, 7 Sneaker, 8 Bag, 9 Ankle boot.
Each image is 28 pixels in height and 28 pixels in width, for a total of 784 pixels in total. Each pixel has a single pixel-value associated with it, indicating the lightness or darkness of that pixel, with higher numbers meaning darker. This pixel-value is an integer between 0 and 255. The training and test data sets have 785 columns. The first column consists of the class labels (see above), and represents the article of clothing. The rest of the columns contain the pixel-values of the associated image.
I try three methods to classify the images. And the results are given in table 2. We can see the XGBoost indeed performs better at running time and accuracy of test dataset.

	Running	Accuracy	Accuracy
AdaBoost	0:01:00.267	0.4493	0.4495
Gradient	0:30:11.367	0.9555	0.8659
XGBoost	0:18:23.255	0.9024	0.9076

Tabel.2 Experiment results for three algorithms

Conclusion

In brief, I present basic tree ideas and three tree boosting algorithms: AdaBoost, Gradient Boosting and XGBoost. In XGBoost greedy and approximate algorithm are proposed to find the optimal split points. And sparsity-aware algorithm is present to deal with sparse data. At last the experiment shows that the XGBoost can classify the image more accrately and avoid overfitting.

References

Michael Kearns. 1988. Thought on Hypothesis Boosting, Unpublished manuscript (Machine Learning class project, December 1988)

Robert E. Schapire. 1990. The strength of weak learnability. Machine Learning 5: 197-227

Yoav Freund. Robert E. Schapire. 1996. Experiments with a New Boosting Algorithm.In *the Thirteenth International Conference on Machine Learning*

Llew Mason. Jonathan Baxter. Peter Bartlett. Marcus Frean. Boosting Algorithms as Gradient Descent in Function Space. *Advances in Neural Information Processing Systems* 12. MIT Press: 512–518.

T Chen. S. Singh, B. Taskar, and C. Guestrin. Ecient Second-order gradient boosting for conditional random fields. *In Proceeding of 18th Articial Intelligence and Statistics Conference* (AISTATS'15), volume 1, 2015.

Tianqi Chen. Carlos Guestrin. 2016. XGBoost: A Scalable Tree Boosting System *KDD* '16, August 13-17, 2016. 785-794

Xiao Han. Kashif Rasul and Roland Vollgraf. "Fashion-MNIST: a Novel Image Dataset for Benchmarking Machine Learning Algorithms." CoRR abs/1708.07747 (2017).