Demystifying Maths of Gradient Boosting

This article discusses the concepts behind the math of Gradient Boosting algorithm

Krishna Kumar Mahto · Follow
Published in Towards Data Science · 10 min read · Feb 25, 2019

Introduction

Boosting is an ensemble learning technique. Conceptually, these techniques involve: **1. learning base learners; 2. using all of the models to come to a final prediction**. Ensemble learning techniques are of different types and all differ from each other in terms of how they go about implementing the learning process for the base learners and then using their output to give out the final result. Techniques that are used in ensemble learning are Bootstrap Aggregation (a.k.a. Bagging), Boosting, Cascading models and Stacked Ensemble Models. In this article, we shall discuss briefly about Bagging and then move on to Gradient Boosting which is the focus of this article. There are a lot of sources which explain the steps in the algorithm of Gradient Boosting. But if you try finding a source that explains what does each step really do that makes the entire algorithm work, you probably will find articles which use *squared error* as an example to do so. Those explanations are very nice but the problem is that they focus so much on squared error that they almost fail to convey a *generalised* idea. Gradient Boosting is a generic model which works with any loss function which is differentialble, however, seeing it work with a *squared loss model* alone does not completely explain what it does during the learning process. In this article, I intend to explain the algorithm through a more generic approach.

> ***Note:*** *Base models are also referred to as base learners in literature. They are one and the same. However, I have used the term Base Learners for the base models and 'Model' for the function constituted by the base learners.*

Bagging

Bagging is a combination of two subsequent steps:

i. Bootstrap sampling of the dataset, into *M* subsets. Each one of these *M* subsets are then used to learn a model. These models are called as base learners/models.

ii. Taking a majority vote to declare the final prediction value.

Since in bagging, a subset of the dataset is used to train a base model, each of the base learners is likely to overfit (since each model has lesser examples to learn from, they may not generalise well). Taking majority vote is gives a model that has a variance which is the average of the variances of all base learners (figure 1).

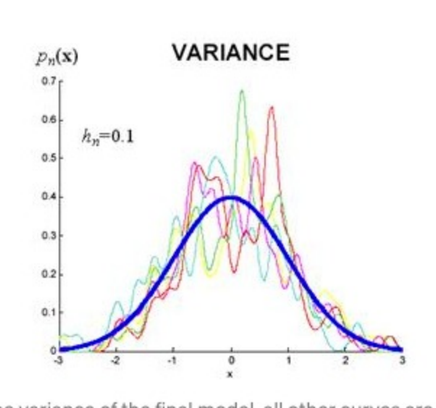

Fig. 1. Blue curve rerpesents the variance of the final model, all other curves are variances of the base learners (source: https://www.quora.com/What-does-Bagging-reduces-the-variance-while-retaining-the-bias-mean)

Boosting is quite different from Bagging in its approach of training base learners and using them to give final results. Bagging learns base learners from independently bootstrapped subsets of data, and hence we can train all the base learners simultaneously in a parallel environment. Boosting, on the other hand, trains the base learners sequentially- models are trained one after the other. Therefore, training base learners in parallel is impossible. Moreover, in a Boosting algorithm we start with a high bias model. The actual model is first initialised with a constant value. It is then progressively made less biased by adding base learners to it. We shall see how Gradient Boosting goes about learning a final model that has a much lower bias given an appropriate number of base learners.

Top highlight

Gradient Boosting

The idea of Additive modelling:

Additive modelling is at the foundation of Boosting algorithms. The idea is simple- form a complex function by adding together a bunch of simpler terms. In Gradient Boosting, a number of simpler models are added together to give a complex final model. As we shall see, gradient boosting learns a model by taking a weighted sum of a suitable number of base learners.

Gradient Boosting algorithm

Input: training set $\{(x_i, y_i)\}_{i=1}^{n}$, a differentiable loss function $L(y, F(x))$, number of iterations M.

Algorithm:

1. Initialize model with a constant value:

$$F_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma).$$

2. For $m = 1$ to M:
   1. Compute so-called *pseudo-residuals*:
   $$r_{im} = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)} \quad \text{for } i = 1, \ldots, n.$$
   2. Fit a base learner (e.g. tree) $h_m(x)$ to pseudo-residuals, i.e. train it using the training set $\{(x_i, r_{im})\}_{i=1}^{n}$.
   3. Compute multiplier γ_m by solving the following one-dimensional optimization problem:
   $$\gamma_m = \arg\min_{\gamma} \sum_{i=1}^{n} L\left(y_i, F_{m-1}(x_i) + \gamma h_m(x_i)\right).$$
   4. Update the model:
   $$F_m(x) = F_{m-1}(x) + \gamma_m h_m(x).$$
3. Output $F_M(x)$.

Source: https://en.wikipedia.org/wiki/Gradient_boosting

Pseudo-residuals

In the algorithm, step 2(1) mentions about computing 'pseudo-residuals'. Although there is hardly any concrete conceptual definition of what a pseudo-residual is, but mathematically what is mentioned is how you define it. However, I feel the name is kind of borrowed from the difference *(y_actual - y_predicted)* which is often referred to as *residual error* which we get by taking a derivative of square loss function ***L*** w.r.t. the predicted value ***F(x_i)*** for *ith* example.

$$L = (y_i - y_p)^2$$
$$\frac{\partial L}{\partial y_p} = -2(y_i - y_p)$$
$$-\frac{\partial L}{\partial y_p} = 2(y_i - y_p)$$
$$\text{where } F(x_i) = y_p$$

Fig.2. Negative of Derivative of squared loss w.r.t. hypothesis function gives the residual for ith example

In optimization problems, constants attached to objective functions do not affect the optimal points, so the factor of *'2'* as shown in figure 2 does not actually matter and can safely be dropped (only when we are solving for optimization). Probably no loss function differentiates to give residual, yet, in the case of squared loss, its differentiation gives a function that is closest to the *residual error 'visually'*. May be the name has been borrowed from this. Nonetheless, *gradient boosting has nothing to do with the derivative of loss function w.r.t. hypothesis being equal to the residual error.*

How the algorithm works

We may have got an idea of one of the things we do in gradient boosting - taking derivatives of the loss function w.r.t. the hypothesis function. Needless to say, the loss function has to be differentiable w.r.t. the hypothesis function. As the algorithm says, gradient boosting takes the training set and a loss function as inputs. We denote the final model that we shall get at the end of the algorithm by ***F_M(x)***.

> *Notations*

Some of the notations used in the algorithm taken from wikipedia (posted above) are somewhat inconsistent. So, I shall use my own notations for those variables.

F_M(x): Final model that will be learned by taking weighted sum of *M* base learners (additive modelling).

F_m(x): Model obtained by adding *m (=1, 2,…, m)* base learners and the initial constant value of *F_M(x)*.

> ***Step 1****. Initialise model with a constant value.*

We find a constant model *F_0 = γ* that fits to the actual y-values. Why a constant model? This is where the idea of boosting starts to manifest- going from a high bias to a low bias model. We start with a constant function (no other function can have more bias than a constant function, unless the dataset is so boring that even a constant model fits it) and then go on through a number steps finding a function that has a reasonably low bias. In some texts, you may find initialising the model with *0* (zero). That is also a constant function, but quite possibly we can start with a slightly better option. Let's say the initial model is a constant function *γ*. The loss function for a constant function then is as follows:

$$\sum_{i=1}^{n} L(y_i, \gamma)$$

Therefore, *γ_optimal* is determined by the solving following optimization problem:

$$F_0(x) = \gamma_{optimal} = \min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

This is for sure better than a model initialised with zero. Particularly for squared loss error, *F_0(x)* is equal to the mean of the actual *y-values*, i.e., *F_0(x) = y_mean* when squared loss is used.

Often in texts, this model is not counted as one of the base learners although the final model is going to be an *additive model* obtained from base learners and this model will also be one of addends. There is one difference though- all the models that are called as base learners in the literature, are fitted on the pseudo-residuals and not on the *y-values* of the actual dataset (we shall see how it is done in step 2). Since *F_0(x)* is fitted on the *y-values*, probably that is why it is not considered as one of the base learners in the literature.

At this point ***F_M(x) = F_0(x)***. ***F_M(x)***, however, is subject to updations until all the *M* base learners are added to it with suitable weights.

> ***Step 2:*** *This step is to be followed for each base learner from m = 1 to m=M. Note that gradient boosting adds one model at a time, one after the other.*

Gradient boosting algorithm has to be configured with a suitable value of hyperparameter *M (number of base learners)* prior to execution.

> *Step 2.1. Compute pseudo-residuals:*

Pseudo-residuals are computed for each *ith* training example :

$$r_{im} = -\left.\frac{\partial L(y, F_{m-1}(x))}{\partial F_{m-1}(x)}\right|_{x=x_i,\ y=y_i} \quad for\ all\ i = 1, 2, \ldots, n$$

Note that *F_m-1(x)* is the model obtained by adding *m-1* weighted base learners and the initial constant function. The *m_th (m superscript th) base learner* has not yet been added. Each residual calculation *r_im* for *ith* training example corresponding to the current base learner *m* (which **will be** trained and added in step 2.2) is done on the weighted sum of base learners from *1 to m-1* **and** the initial constant function (step 1). Recall that after step 1, *F_0(x) = γ* does not include any term corresponding to any base learner (recall that *F_0(x)* is most often not called as a base learner in the literature. It is treated only as an initial value of the model).

At this point, we have computed the pseudo-residual values for each training example.

> *Step 2.2. Fit **base learners** on the pseudo residuals*

For this step, a new dataset is derived from the given dataset. The dataset *D_modified* is defined as shown in figure 3.

$$D_{modified} = \{(x_i, r_{im}) : i = 1, 2, .., n\}$$

Fig. 3. Modified dataset

A base learner *h_m(x)* is trained and fitted on this dataset.

At this point, we have everything required to determine *F_m(x)*. We do it as following:

$$F_m(x) = F_{m-1}(x) + \gamma h_m(x)$$

Why does doing this make sense?

It is straightforward to see why this equation makes sense if we compare it with weight updations done in *Gradient Descent* (figure 4). Weights in gradient descent are moved in the direction in which loss *L* decreases. How much the weights should be moved is governed by α (the learning rate). Similar to that, the function *h_m(x)* is fitted to the rate of change of loss *L* w.r.t. *F_m-1(x)*. The function *h_m(x)* (which is expected to approximate the behaviour of derivate of loss w.r.t. *F_m-1(x)* suitably) represents the direction in which the *loss function* decreases w.r.t. *F_m-1(x)*. γ corresponds to the hyperparameter α in terms of the utility (both determine by what amount update should be made). This is similar to the weight updation equation in Gradient Descent , except that γ is trainable while α is a hyperparameter. We shall see how an optimal value of γ is learned in step 2.3 .

$$W \leftarrow W - \alpha\frac{\partial L}{\partial W}$$

Fig. 4. Weight updation in Gradient Descent

Note that γ is the only parameter w.r.t. which *F_m(x)* needs to be optimized on the original dataset. However, base learners learn more parameters on dataset *D_modified* since they are the actual functions which are added together to give the final model.

At this point, we have a model *F_m(x)* which will be used in the next step to compute *y_pred* values.

> *Step 2.3. Find the optimum γ*

We shall take the the original dataset *D* (Figure 5).

$$D = \{(x_i, y_i) : i = 1, 2, .., n\}$$

Fig. 5. Original dataset

We take the model *F_m(x)* on the original dataset *D*, then compute the loss *L*. Note that this loss is a function of *γ*.

$$\sum_{i=1}^{n} L(y_i, F_m(x_i))$$

We are supposed to find *γ_optimum*. This can be done by solving the following optimization problem that minimizes the :

$$\gamma_{optimum} = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, F_m(x_i)) = \underset{\gamma}{argmin} \sum_{i=1}^{n} L(y_i, F_{m-1}(x_i) + \gamma h_m(x_i))$$

At this point we have the optimum value of *γ*.

> *Step 2.4. Model Updation*

The model *F_m(x)*, thus is obtained as:

$$F_m(x) = F_{m-1}(x) + \gamma_{optimum} h_m(x)$$

At the end of each iteration number *m, F_M(x)* is updated to the value of *F_m(x)*.

Step 3. Step 2 is run for each base model *m = 1 to M*. After *M* iterations of step 2, we obtain the final model *F_M(x)*.

Why the name 'Gradient Boosting'?

We just saw the role 'gradient' plays in this algorithm- we always fit a base learner to the gradient of the loss function w.r.t. the model *F_m-1(x)*. The term 'boosting' refers to the fact that a high bias model, which performs really bad on the dataset, is boosted to finally become a reasonable classifier and possibly a strong classifier. In general, Boosting is a family of algorithms in which a number *'weak classifier'* (one which has an error rate of just under 0.5) base learners are combined to give a strong classifier (one which has an error rate close to 0). In gradient boosting, we start with a constant model (which maybe an extremely weak classifier depending on the dataset). At the end of an *m_th* iteration of learning a base learner, we end up with a weak learner, but relatively a better classifier *F_m(x)* which progressively moves towards becoming strong classifier.

Footnotes

I hope this article makes sense. My goal is not just to put down an algorithm, I intend to demystify the math (as title says) and bring forth what lies beyond the mathematical equations. So, if I failed to convey the conceptual ideas behind the math anywhere, kindly drop a comment, it would be really appreciable.

Thank you for reading!

Machine Learning · Data Science · Artificial Intelligence · Mathematics · Gradient Boosting

Written by Krishna Kumar Mahto
Follow
210 Followers · Writer for Towards Data Science
Software Engineer-Intern at EIG | Machine Learning Enthusiast | Python | VIT |
linkedin.com/in/krishna-kumar-mahto