DATA SCIENCE II: Machine Learning MTH 9899 Baruch College

Lecture 4: Boosting, Neural Network Gradient Descent Learning

Adrian Sisser

May 2, 2018

Outline

Outline

Boosting

Boosting is the idea of combining a set of weak learners to form a stronger learner.

It a type of ensemble, since it's a collection of learners, but is different in many important aspects.

Boosting differs because it trains the learners in sequence, with a focus on errors made by the prior learners.

AdaBoost

AdaBoost was the first successful implementation of boosting.
It applies to a binary classification problem:

$$Y = -1, +1$$

We will learn weak learners, with a goal of combining them all
into a final learner:

$$f_j : X \to -1, +1$$

$$F_i(x) = \sum_{j=1}^{i} \beta_j f_j(x)$$

AdaBoost

function TRAINADABOOST(X, Y)
    $W_1(j) \leftarrow \frac{1}{N}$ $\triangleright$ N is the number of points in X
    $F_0(X) = 0$
    for $i \in 1..T$ **do** $\triangleright$ i indicates the Boosting Round
        $f_i \leftarrow$ TRAINWEAKLEARNER(X, Y, W_i)
        $\epsilon_i \leftarrow \sum_j W_i(j)I(Y_j \neq f_i(x_j))$
        $\alpha_i \leftarrow \frac{1}{2} \ln \frac{1-\epsilon_i}{\epsilon_i}$
        $W_{i+1}(j) \leftarrow W_i(j) \exp(-\alpha_i y_j f_i(x_j))$
        $W_{i+1}(j) \leftarrow \frac{W_i(j)}{\sum_k W_i(k)}$ $\triangleright$ Normalize the weights to sum to 1
        $F_i(X) \leftarrow F_{i-1}(X) + \alpha_i f_i(X)$
    end for
    return F_T
end function

So now we ask, where did those update rules come from? Let's define an error function for F_j:

$$E = \sum_{i=1}^{N} e^{-y_i F_j(x_i)} \qquad \text{Exponential Loss}$$

$$= \sum_{i=1}^{N} e^{-y_i(F_{j-1}(x_i) + \alpha_j f_j(x_i))} \qquad \text{Using the def of } F_j$$

$$= \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-y_i \alpha_j f_j(x_i)}$$

$$= \sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} \qquad \text{Split up into right/wrong examples}$$

$$= \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} \left(e^{\alpha_j} - e^{-\alpha_j}\right)$$

Now, we want to minimize the expression from before:

$$E = \sum_{i=1}^{N} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} \left(e^{\alpha_j} - e^{-\alpha_j} \right)$$

In the algo, we said:

$$
\begin{aligned}
W_j(i) &= W_{j-1}(i) e^{-\alpha_{j-1} y_i f_{j-1}(x_i)} \\
&= [W_{j-2}(i) e^{-\alpha_{j-2} y_i f_{j-2}(x_i)}] e^{-\alpha_{j-1} y_i f_{j-1}(x_i)} \\
&= W_0 e^{\sum_{k=0}^{j-1} -\alpha_k y_i f_k} \\
&= e^{-y_i F_{j-1}(x_i)}
\end{aligned}
$$

By telescoping the prior statement ...

The left side of the error above has no dependence on f_j. The term on the right is just the weighted sum of the incorrect examples according to f_j. Given that, we try to minimize the sum of the weights of the incorrect points.

Let's look at how we choose α, the weight we use when we add the new learner f_j to the ensemble. We'll start with an earlier vesion of the equation, since it will make solving it easier.

$$
\begin{aligned}
E &= \sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} \\
\frac{\partial E}{\partial \alpha_j} &= -\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} e^{-\alpha_j} + \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} e^{\alpha_j} \\
0 &= -\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)} + e^{2\alpha_j} \sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)} \\
\alpha_j &= \frac{1}{2} \ln \frac{\sum_{f_j(x_i)=y_i} e^{-y_i F_{j-1}(x_i)}}{\sum_{f_j(x_i) \neq y_i} e^{-y_i F_{j-1}(x_i)}} \\
\alpha_j &= \frac{1}{2} \ln \frac{1 - \epsilon_j}{\epsilon_j}
\end{aligned}
$$

We've now seen what AdaBoost is really doing. It's a greedy optimization at each iteration - where we train a learner to best fit the weighted points, and the weight of points increases as they are misclassified.

Outline

1. **Boosting**
   - Overview & AdaBoost
   - Gradient Boosting
   - XGBoost and lightGBM

2. **Optimizers**

Gradient Boosting is another boosting technique. It's applicable to problems like regression. We'll take the same approach as we did with AdaBoost. At each step, we'll add a new weak learner into the overall predictor, and find the optimal weight for it.

- You can use any weak learner, typically, we use Regression Trees
- Like other boosting algorithms, it can be slower than a normal Random Forest, because it can't be parallelized.

function $\text{TRAINGB}(X, Y)$
    $F_0(X) = 0$
    for $i \in 1..T$ **do**
        $r_i \leftarrow -\frac{\partial E}{\partial F_{i-1}}$
        $f_i \leftarrow \text{TRAINWEAKLEARNER}(X, r_i)$
        $\alpha_i \leftarrow \arg\min_\alpha E$
        $F_i(X) = F_{i-1}(X) + \alpha_i f_i(X)$
    end for
    return F_T
end function

Let's look in more detail how this is the same as AdaBoost. In AdaBoost, our loss function was $e^{-y_i F(x_i)}$ and $E = \sum_{i=1}^{N} e^{-y_i F_j(x_i)}$. Since

$$r_j = -\frac{\partial E}{\partial F_{j-1}}$$

$$r_j = \sum_{i=1}^{N} y_i e^{-y_i F_{j-1}(x_i)}$$

$$r_j = \sum_{i=1}^{N} y_i W_j(i) \qquad \text{From earlier slide}$$

So, we can see that r_j is just the weighted inputs from earlier.

What happens in the case of regression? Let's define an error function:

$$E = \sum_{i=1}^{N} \frac{1}{2}(y_i - F_j(i))^2$$

Now let's calculate r_j:

$$
\begin{aligned}
r_j &= -\frac{\partial E}{\partial F_j} \qquad\qquad \text{For GB} \\
&= (y_i - F_j(i))
\end{aligned}
$$

So for regression, r_j is just the residual!! Now, we just have to fit learners at each iteration to our residual, and add them in.

To add in a new residual, we need to find α, the weight.

$$E = \sum_{i=1}^{N} \frac{1}{2}(y_i - F_{j-1}(i) - \alpha_j f_j(i))^2$$

$$\frac{\partial E}{\partial \alpha_j} = \sum -f_j(i)(y_i - F_{j-1}(i) - \alpha_j f_j(i))$$

There is no easy way to solve this. Instead, in a general Gradient Boosting algorithm, the α value is effectively a learning rate - another hyperparameter that we have to tune.

Outline

1 **Boosting**

- Overview & AdaBoost
- Gradient Boosting
- XGBoost and lightGBM

2 **Optimizers**

XGBoost

Last week, we talked about Extra Trees as one of the best 'off the shelf' modelling options. **XGBoost** is an open source package that is also, arguably, the best stock ML algo. XGBoost is short for Extreme Gradient Boosting and has a few interesting features:

- Very well implemented - FAST
- Distributed - It can be split up across a cluster.
- Custom Loss Functions

XGBoost

XGBoost takes care to reduce overfitting by regularizing - they add a penalty to avoid growing overly complex trees.

$$\Omega(f) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$$

Here γ and λ are hyper parameters, T is the number of leaves, and w_j is the value in each leaf of the tree. The first component simple penalizes tree size. The second part is more interesting: We can think of it as penalizing us for vastly different values in the leaves - so we need a significant improvement in loss to justify it.

XGBoost

The next interesting technique in XGBoost is how they model the loss given a new function f_j to add in. We can take a taylor expansion of the loss:

$$\mathcal{L}(Y, Y_j) \approx \mathcal{L}(Y, Y_{j-1}) + \frac{\partial \mathcal{L}}{\partial Y_{j-1}} + \frac{1}{2} \frac{\partial^2 \mathcal{L}}{\partial Y_{j-1}^2}{}^2$$

$$\mathcal{O} \approx \mathcal{L}(Y, Y_{j-1}) + \frac{\partial \mathcal{L}}{\partial Y_{j-1}} + \frac{1}{2} \frac{\partial^2 \mathcal{L}}{\partial Y_{j-1}^2}{}^2 + \Omega$$

Now, when we want to determine if we should add a new layer to the tree, we can see if the new split, will increase or decrease the overall objective based on just the first and second derivatives.

LightGBM

LightGBM is a newer package than XGBoost.

- It is typically significantly faster than XGBoost
- LightGBM introduced a histogram based approach to splitting which led to the speed boost. XGBoost now offers a similar splitting approximation.
- LightGBM uses a "leaf-wise" splitting approach, as opposed to "level-wise". This can lead to deeper trees so overfitting is an issue.

Overall, lightGBM seems to offer slightly better average performance and run slightly faster.

Nesterov Momentum

Nesterov Momentum is based off a simple observation about the basic momentum algorithm. From basic momentum, we know:

$$\theta_{i+1} = \theta_i - (\eta \frac{\partial E}{\partial \theta_i} + \gamma(\theta_i - \theta_{i-1}))$$

This means that we already know a large part of the update for the next step, so we can calculate the gradient starting from there, instead of the current weights:

$$\theta_{i+1} = \theta_i - (\eta \frac{\partial E(\theta_i - \gamma(\theta_i - \theta_{i-1}))}{\partial \theta_i} + \gamma(\theta_i - \theta_{i-1}))$$

AdaGrad

AdaGrad is another optimization technique, based on the idea that we don't update all parameters at the same rate. It also accounts for the fact that we want the learning rate to decrease over time.

First, let's define the sum of the squared gradients over time, G_i will be a diaganol matrix:

$$G_T = \text{diag} \sum_{j=1}^{T} (\frac{\partial E}{\partial \theta_j})^2$$

Now, we learn with this, where ϵ is a small constant to avoid dividing by zero:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_T + \epsilon}} \odot \frac{\partial E}{\partial \theta_i}$$

RMSProp

RMSProp is a variant of AdaGrad (similar to AdaDelta, which we will omit for time considerations) developed by Hinton. We basically replace the sum of squared gradients from AdaGrad with an EMA (Exponentially Moving Average) version.

$$G_i = \lambda G_{i-1} + (1 - \lambda)(\frac{\partial E}{\partial \theta_i})^2$$

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_i + \epsilon}} \circ \frac{\partial E}{\partial \theta_i}$$

Adam

Adam is short for Adaptive Momentum Estimation that extends RMSProp and momentum. Remember that momentum kept a simple EMA of gradient while RMS Prop kept an EMA of the gradient squared.

$$m_T = \beta_1 m_{T-1} + (1 - \beta_1)\frac{\partial E}{\partial \theta_T}$$

$$G_T = \beta_2 G_{T-1} + (1 - \beta_2)(\frac{\partial E}{\partial \theta_T})^2$$

These exponential moving averages are very biased initially, so Adam corrects them:

$$m_T^* = \frac{m_T}{1 - \beta_1^T}$$

$$G_T^* = \frac{G_T}{1 - \beta_2^T}$$

Adam takes these two updates, and combines them similarly to AdaGrad:

$$\theta_{i+1} = \theta_i - \frac{\eta}{\sqrt{G_i^*} + \epsilon} m_i^*$$