De-Mystifying XGBoost Part II

Digging into the Hyper parameters

Deb

Mar 19, 2020·24 min read

Prelude

This is a continuation to De-Mystifying XGBoost Part I. If possible, keep a print out handy as I will be referring to it all throughout this blog.

In this blog, I will be discussing the intuition behind the concepts and various parameters we use while training a XGBoost model. Often, we train such models blindly. The algorithm is so logical, as you must have understood from Part I, that even done blindly, with all the checks and balances (as we will discuss more below), it is hard to come out with a sloppy model.

Or is it? What do you think?

At the end of this blog, we should be able to see the parameters more visually and use intuition while adjusting them and reasoning why a model is not doing so good on unseen data. I will only discuss a handful of parameters in this story and the rest will be discussed in Part V. Also, you should get the answers to all the Quiz questions in Part I.

Let's get started ..

Intuition

Gradient

The most intuitive thing we take forward from the math in Part I, is the importance of gradient. It is the basis of how XGBoost creates and expands trees. As we now know, leaf scores are = -G/(H+lambda) and loss in a node is proportional to $-G^2/(H+lambda)$. Lambda, remember? The L2 regularization parameter for leaf scores.

Hence, in order to decrease loss from parent to child, we need to decrease loss in child nodes, and thus (with a negative sign) the value of mod G needs to be more. G =summation of (p-y). So the tree function will need to find column value conditions, such that, the mod sum of (p-y) is large in each child. And that happens when all the (p-y)s are of the same sign - right? Think about it.

Right, so intuitively, a good split condition will try to separate out samples of a label into a one child node and samples of opposite label into its corresponding sibling node. More they are mixed up, smaller will be the sum of (p-y). Thus good intuitive split conditions minimize the loss too.

Intuitively again, the weight or delta score, is proportional to the negative of the gradient. Which means ***delta y hat*** adjusts itself to the opposite of the gradient (or error). Which is what we want! Right? Correct the error.

As the trees keep correcting the overall error, ***pi*** for a sample, (which derives from ***y hat i)***, keeps changing from one tree to next. See the last pic from Part I. For a sample, ***y hat*** may increase or decrease as it moves from one tree to the next. At the end of every tree, (***pi-yi***) or gradient, is the error for that sample ***i***. Or the difference between the last best-predicted value and real label of ***i***. And the summation of ***(pi-yi)*** or error (averaged over all samples in a leaf or cover

This is why we say XgBoost learns incrementally from its error.

Hessian/Cover

We also learnt in Part I, for binary logistic loss function, the hessian is ***p(1-p)***. On a node, the sum of hessian is the sum of ***pi(1-pi)***. When

XGBoost specifies min child weight for binary classification, it is this value which is being considered as the minimum allowable value. Let's get thinking on this a bit.

Imagine on some tree, down some depth, there is a node which when split on some condition, gives two nodes, one of which have 10 samples in it. Let's say, the ***y hat*** values of these samples, before getting into the tree, are already quite matured, i.e, ***y hat t-1***s are close to their expected score or completely wrong. Let's say ***pi*** thus, is already on avg 0.95 for the ones which have label 1, and 0.05 on avg for the ones which have label 0s. And there are some wrongly classified label 1 samples with ***pi*** 0.05 and the vice-versa for label 0 samples.

What is the sum of hessians for this leaf? It is ~= 0.95*0.05*10 ~= 0.475. Even if there are some class 1 samples with ***pi*** on the wrong side and vice versa for class 0 samples, the sum of hessian would still be ~= 0.475.

If ***min child weight*** for XGBoost is set to 1, this split would not be allowed. Irrespective of whether it gives the lowest loss from all possible children nodes derived from all feature value split conditions possible on that node.

Intuitively, a low sum of hessians, or low ***cover*** means, we are considering a very slim and local condition. A low hessian or cover will result in higher ***delta y hat***s (because it is in the denominator) of -G/(H+lambda). Given a default lambda of 1, a very low hessian almost results in weights proportional to the gradient. Such weights can cause big erroneous score movement for the wrongly classified samples (their ***y hat*** will move opposite to their error or gradient, along with the samples of majority class on the node, whose ***y hats*** will move in the desired direction, negative of the overall gradient).

If, the sample size though, after the split, on each child, is >100; such a split would have been allowed, as the sum of hessians would be much greater than 1. 0.95*0.05*100 ~= 4.75. Which, intuitively means; with such a large

sample size, the split is more likely a valid split, than an overfitted one. We can expect the same characteristic in unseen data too. And a large hessian will result in a controlled score/weight (***delta y hat***) on all the samples in the node.

Or, it may be the other way around. 10 samples with probabilities on avg 0.85 and/or 0.15. Sum of hessians would be ~=1.275. In this case, too, we are OK. It is an early split, where the majority ***pi***s are not yet tending towards 1 or 0. The gradients are high but will be mixed up and overall, the optimum weight will result in a controlled score movement or delta. And such splits are not over-fitted splits. We need such splits to start correcting the gradients.

I have excels coming up soon which will let you visualize these movements.

So, for binary classification, the cover or sum of hessians on a node is sensitive to both the number of samples and to the nearness of probability of those samples (calculated till the last tree) to 1 or 0. More mature the probabilities (right or wrong), higher the number of samples needed to justify a split.

For linear regression (actually that is a good exercise if you can do the math again for linear regression), where the loss function is mean square error, the sum of hessian comes out as 1*n. Or, the number of samples. Hence for linear regression, cover, is simply the number of samples on a node.

So, to visualize a default ***min child weight*** of 1, think needing at least 10 samples around ***pi*** 0.85 and 0.15 for a split condition, or at least 20 for samples around ***pi*** 0.95 and 0.05, at least 50 for 0.98, 0.02 and so on. If you think, that is a fair node purity threshold in your use case, then 1 is a good number. You can go less or more. Generally, it is not suggested going less than 1, unless your sample size is very small, like in hundreds only.

Going more than ***min child weight*** of 1 is safe, conservative. However, the higher you go, the more difficult will be the splits with shorter trees. So

even if you have a high max depth, it will never probably reach those depths. We will discuss more depth soon.

Gradient Hessian Trade Off

As you must have felt by now, between gradient and hessian, there is a constant tug of war going on. Gradients will try to give higher mod leaf scores or ***delta y hat***s. Whereas Hessian will lower it or average it out.

When a model is just starting, we have high gradients and high hessians too. As we keep adding trees, we have lower gradients, and the hessians become small too. In fact, after many trees, when the majority of the samples have low gradient or error, hessian is almost zero and the only thing that is stopping us from an exploding weight or score is the lambda in the denominator.

Overall, its a game of controlling the ***delta y hat*** so that it is never too big and in the right direction according to the error on each node.

Lambda

So, what to do about lambda? I would say, the default of 1 is a very good value. It keeps the scores from exploding, and at the same time, does not down weigh the gradients too much.

If we increase lambda, there is no apparent harm though. The weights or leaf scores will be smaller and hence the model will learn slower. There will be more trees, more memory, more time needed. But no harm to the model. If we decrease it though, things will be the other way around, and we might end up harming the model. How? We will discuss when discussing ***eta***.

But, just thinking about it, if lambda is less than one, and the ***y hat***s are very close to 1 or 0 (leading to a sum of hessian to be almost 0), we will end up having scores or ***delta y hat***s, which can be more than the error or gradient on a sample. Right? So we will be overshooting the gradient — back and forth. That does not sound good right?

Base_Score and Zero Depth Trees

So, now that we know how the trees are being grown. What if we have 0 depth trees? Like why try to reduce loss from ***lp*** to ***lc***. When we made the derivation in Fig 12, L or number of leaves could have been 1 too, right? Absolutely right and yes we can train 0 depth trees in XgBoost — try it out.

But what will happen? Since, all the samples will be on a single node and get the same score, think about it, what will be the best way to minimize the loss? Pure intuition says that we will need to change ***y hat***s according to the fraction of classes. So let's say we have 200 class 0 samples and 20 class 1 samples. Intuitively I will start decreasing ***y hat***s for all samples from 0 towards -5 because class 0 samples are higher in number.

That is exactly what will happen if we choose w=-G/(H). If we add that to ***y hat*** = 0, we will get one score for all samples, which will result in the minimum log loss, and the delta or w, will definitely be a negative number considering the above example. Just to give you that number, ***y hat*** = -1.63 will be the best ***y hat*** for all 220 samples, resulting in minimum LogLoss.

So, will XgBoost train for one round and stop? (given we stop at best LogLoss). What if I give more rounds? Well yes, it will stop at one round, but, only if we do not use lambda. If we use ***lambda*** regularization and ***eta*** (learning rate), XGBoost will keep reducing LogLoss for a few rounds and then it will stop once no more reduction is possible. Why ? because we know both lambda and learning rate slows down the actual ***delta y hat*** or scores.

So it will stop after one round if w = -G/H. But it will continue for few rounds if w = eta*-G/(H+1).

Cool fact — right ? ***lp-(lc1+lc2)***. And we try to maximize the gain as we build subtree after subtree.

In fact, that score (y hat = -1.63) which comes out from simple math and minimization of LogLoss, given a 200:20 class distribution, can be used as

a ***base_score*** instead of 0. That is, only if you think the class distribution has a big say (bias) in the final probability. But be aware, this assumes that your training sample class distribution will always be exactly the same as the unseen distribution, and every prediction will start with that bias. So — only use with great caution. Not recommended.

Base Score can also sometimes get some value different from 0, when using cascaded models. Like the same sample going through different feature sets and or models in sequence, where the output of the first model becomes a bias for the second model and so on.

Max Delta Step

Let's understand this with some real data. I created an illustrative data set of samples in a node. It has a label distribution of 200 samples in class 0 and 20 in class 1. Split on a condition, the node splits into two nodes. One with 185 samples and other with 35. Let's say out of the 185, there are 180 from class 0 and 5 from class 1. And out of the 35, 20 from class 0 and 15 from class 1. See Pic below. This actually looks like a good split, right? Majority fraction class 0 in child 1 and majority fraction class 1 in child 2.

I have given some prior probability values to the samples, assuming the parent node is after some trees. Hence, the ***y hat***s have moved away from being all = 0. I put most class 0 samples (150) at probability 0.1 and remaining with varying degree of errors or gradient (lines 2 and 3below). Similarly, I have put most class 1 samples at probability 0.9 (8) and remaining with varying gradients (lines 2 and 4 below). Then I calculated all the G, H, Loss and Weight/Scores assuming a lambda of 1. For the parent node and the two child nodes. See below.

	A	B	C	D	E	F	G	H
1		Parent						
2	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
3	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
4	Samples Class 0 (n0)	150	12	8	8	6	6	4
5	Samples Class 1 (n1)	1	1	1	1	2	2	2
6	G Class 0 sum(p-0)	15	2.4	2.4	3.2	3	3.6	2.8
7	G Class 1 sum(p-1)	-0.9	-0.8	-0.7	-0.6	-1	-0.8	-0.6
8								
9	H Class 0 sum(p(1-p))	13.5	1.92	1.68	1.92	1.5	1.44	0.84
10	H Class 1 sum(p(1-p))	0.09	0.16	0.21	0.24	0.5	0.48	0.42
11								
12								
13	Gain	=Parent-(Child1+Child2)			7.87421854			
14								
15		Child 1 - Class 0 Majority						
16	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
17	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
18	Samples Class 0 (n0)	138	10	7	7	5	5	3
19	Samples Class 1 (n1)	0	0	1	0	1	0	0
20	G Class 0 sum(p-0)	13.8	2	2.1	2.8	2.5	3	2.1
21	G Class 1 sum(p-1)	0	0	-0.7	0	-0.5	0	0
22								
23	H Class 0 sum(p(1-p))	12.42	1.6	1.47	1.68	1.25	1.2	0.63
24	H Class 1 sum(p(1-p))	0	0	0.21	0	0.25	0	0
25								
26								
27							Delta	Score
28								
29	new yhat	-3.5606341	-2.7506341	-2.2106341	-1.7706341	-1.3606341	-0.9506341	-0.5106341
30	new prob	0.03	0.06	0.1	0.15	0.2	0.28	0.38
31								
32		Child 2 - Class 1 Majority						
33	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
34	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
35	Samples Class 0 (n0)	12	2	1	1	1	1	1
36	Samples Class 1 (n1)	1	1	0	1	1	2	2
37	G Class 0 sum(p-0)	1.2	0.4	0.3	0.4	0.5	0.6	0.7
38	G Class 1 sum(p-1)	-0.9	-0.8	0	-0.6	-0.5	-0.8	-0.6
39								
40	H Class 0 sum(p(1-p))	1.08	0.32	0.21	0.24	0.25	0.24	0.21
41	H Class 1 sum(p(1-p))	-0.081	-0.128	0	-0.144	-0.125	-0.192	-0.126
42								
43								
44								
45								
46	new yhat	-2.1646393	-1.3546393	-0.8146393	-0.3746393	0.03536068	0.44536068	0.88536068
47	new prob	0.1	0.21	0.31	0.41	0.51	0.61	0.71

Node split into probable children on some criterion. Unbalanced data. You can access this spreadsheet at https://github.com/run2/demystify-xgboost/blob/master/DeMystifyXGBoost-Example1.xlsx. You can see the formula on each cell to check out how I calculated the values. Play around with the number of samples and or probability. The loss, weight, new probability etc will automatically change! :)

Let's dig a bit into the Gs and Hs.

For a class 0 at p= 0.1, H = 0.1*(1–0.1) = 0.09. And G = (0.1–0) = 0.1.

For class 1 at p = 0.9, H is again = 0.09 and G is = (0.9 -1) = -0.1.

So mod of G is more than H, right? So, if we have many samples which are class 0 and very few which are class 1, the leaf with majority class 0 samples will tend to have a G which is much higher than H or H + 1. In the example we can see in Child Node 1, G from class 0 samples is 32.5 (K20). Only a little bit of 1.6 G cancels out for class 1 samples. H is 21.71. So we end up having a weight or score or delta ***y hat*** = -1.36

If we allow this split, and the child nodes end up as a leaf, all the class 1 samples in Child 1 will have a ***delta y hat*** of -1.36. This means, the probability of the wrongly classified class 1 samples will jump suddenly, in an adverse direction (opposite to their gradient). For e.g, 2 class 1 samples will jump from a probability of 0.9 to 0.7. See red marked.

This is where max delta step comes in. If max delta step is set to let's say 1, this above split will not happen, even if it results in the lowest ***lc***. That will force the algorithm to search other feature value combinations where ***lc*** might be higher, but the split will be such that there is a smaller change in score.

*Note above in the excel, in Loss Parent calculation, I did not use the 1/2 factor of $-1/2*G^2/(H+1)$ as we had calculated in the math part (Fig 12 Part I). XGBoost does not apply this 1/2 factor because it is a constant multiplier for both parent and child nodes.

Eta (learning rate)

We already know that eta is a fraction to multiply into the leaf scores and by keeping the ***delta y hat*** low, we make sure that there is no sudden jump in a sample's probability due to local loss minimization in some leaf, based on some locally sensitive column value conditions.

From the above example in max delta step topic, if we use eta of 0.3, the leaf scores and ***delta y hat***s will be reduced to -1.36*0.3 = -0.4 and 0.035*0.3=0.01. Given a ***max delta step*** 1, this is an allowable delta.

So, from the class 0 miss-classified samples in child 1, the two red marked samples will only move to a ***y hat*** of 2.2–0.4 = 1.8, with a probability of 0.85; as compared to ***y hat*** and the probability of 0.84 and 0.7 respectively without eta. So overall slower learning (for correctly split samples) — but less error (for samples in the wrong split).

The crucial question here though is — does ***eta*** only control the speed of learning. I.e, is it only going to affect the number of trees needed. Larger number for smaller ***eta*** and vice versa? The answer is no.

Because of the way, every split looks at gradients and takes into consideration other factors like max delta step, gamma(see below), min child weight and other tree booster parameters, it is highly possible that a high eta or eta =1 will lead the algorithm into a position from where it will not be able to reduce the loss any more (think about overshooting the valley of a concave loss curve) and basically stop at a value of evaluation metric which is not optimal. The same effect might be seen when reducing lambda from 1.

Bottom line, keeping eta low will not harm, other than slower learning. Keeping eta high can harm with resulting in an inferior model. Obviously too low an eta, like 0.01 is also not recommendable as our model will keep training for hours. So a default value of 0.3 is a good start and we can decrease it slightly to see if the evaluation metric betters. Will have a section on this in the third part of this de-mystifying series.

Gamma

And that brings us to Gamma. Gamma, again a tree boosting parameter is the minimum gain, or ***lp-(lc)***, or ***lp-(lc1+lc2)***, allowed for a node to split further into two more nodes. Let's change the number of samples of class 0 and class 1 in each child node from the above example like below. It's a bad split. We have almost 50% of class 0 samples in each node and the same for class 1. I calculated the G H and loss for each node again.

	A	B	C	D	E	F	G	H
1						Parent		
2	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
3	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
4	Samples Class 0 (n0)	150	12	8	8	6	6	4
5	Samples Class 1 (n1)	1	1	1	1	2	2	2
6	G Class 0 sum(p-0)	15	2.4	2.4	3.2	3	3.6	2.8
7	G Class 1 sum(p-1)	-0.9	-0.8	-0.7	-0.6	-1	-0.8	-0.6
8								
9	H Class 0 sum(p(1-p))	13.5	1.92	1.68	1.92	1.5	1.44	0.84
10	H Class 1 sum(p(1-p))	0.09	0.16	0.21	0.24	0.5	0.48	0.42
11								
12								
13	Gain	=Loss Parent-(Loss Child1+Loss Child2)			0.96660735			
14								
15						Child 1		
16	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
17	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
18	Samples Class 0 (n0)	80	8	5	5	4	4	2
19	Samples Class 1 (n1)	0	1	1	0	2	1	1
20	G Class 0 sum(p-0)	8	1.6	1.5	2	2	2.4	1.4
21	G Class 1 sum(p-1)	0	-0.8	-0.7	0	-1	-0.4	-0.3
22								
23	H Class 0 sum(p(1-p))	7.2	1.28	1.05	1.2	1	0.96	0.42
24	H Class 1 sum(p(1-p))	0	0.16	0.21	0	0.5	0.24	0.21
25								
26								
27							Delta	Score
28								
29	new yhat	-3.2519952	-2.4419952	-1.9019952	-1.4619952	-1.0519952	-0.6419952	-0.2019952
30	new prob	0.04	0.08	0.13	0.19	0.26	0.34	0.45
31								
32						Child 2		
33	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85
34	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7
35	Samples Class 0 (n0)	70	4	3	3	2	2	2
36	Samples Class 1 (n1)	1	0	0	1	0	1	1
37	G Class 0 sum(p-0)	7	0.8	0.9	1.2	1	1.2	1.4
38	G Class 1 sum(p-1)	-0.9	0	0	-0.6	0	-0.4	-0.3
39								
40	H Class 0 sum(p(1-p))	6.3	0.64	0.63	0.72	0.5	0.48	0.42
41	H Class 1 sum(p(1-p))	-0.081	0	0	-0.144	0	-0.096	-0.063
42								
43								
44								
45								
46	new yhat	-3.4563285	-2.6463285	-2.1063285	-1.6663285	-1.2563285	-0.8463285	-0.4063285
47	new prob	0.03	0.07	0.11	0.16	0.22	0.3	0.4
48								

Node split into probable nodes on some criterion. Low gain. You can access this spreadsheet at https://github.com/run2/demystify-xgboost/blob/master/DeMystifyXGBoost-Example2.xlsx. You can see the formula on each cell to check out how I calculated the values

As we can see, the gain or ***lp-lc*** now is = 0.966. The gain in the previous example was around 7. Intuitively the split does not look very good too. If gamma was set to 1, this split would be ruled out. If this gain was the best gain achievable from all column value combinations, this node would not split at all. It would become a leaf node.

Overall, ***lp*** and ***lc*** depend on $-G^2/(H+\text{lambda})$. So ensuring a good loss reduction basically means getting a split, where there is a significant mod of G. Or, the gradient of the same sign (+ or -) in each child of a split. We already discussed this when discussing gradient.

Finding the best gamma or minimum gain allowed, below which, splits will not be allowed, is not easy. We need to be really sure about our data size and distribution to start playing with it. However, if max depth is set to a high value, having a positive gamma value, will prune the trees automatically, so that there is no unnecessary splitting with very low loss reduction.

Even if we do not set gamma, the inherent nature of the algorithm, of finding the split with the maximum gain, will avoid low gain splits. Gamma is an extra protection to make sure the trees are conservative. Especially good to use, if we plan to use many conservative models and ensemble them later.

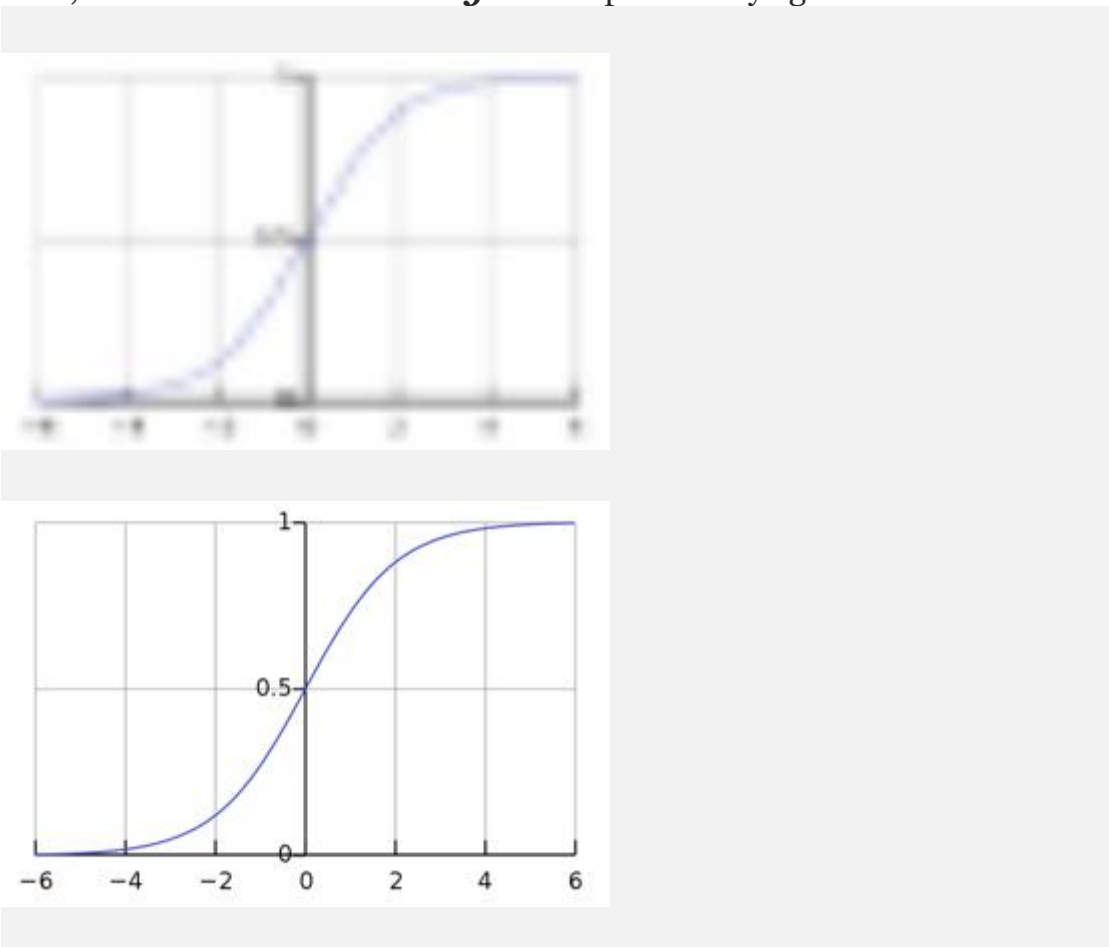
There is often a confusion between min child weight and gamma. Hopefully, there is no such confusion if you have reached till here. If you notice Child 2 from example2, the sum of hessian is 9.6 (K42). If min child weight was 10, this would have become an invalid split. Min child weight makes sure a leaf has enough cover, leading to controlled weight or score. Gamma makes sure

we are not splitting a node such that the overall gain from parent to children is very low. So they are not the same.

Scale pos weight and AUC

OK, this one, I must admit was tricky to understand. I applied it some times, without understanding the head or tails of it and felt guilty every time. But now that we have all the math under our wings, and especially the excel sheets to play around, this should be as easy as “having rice and lentil soup”.

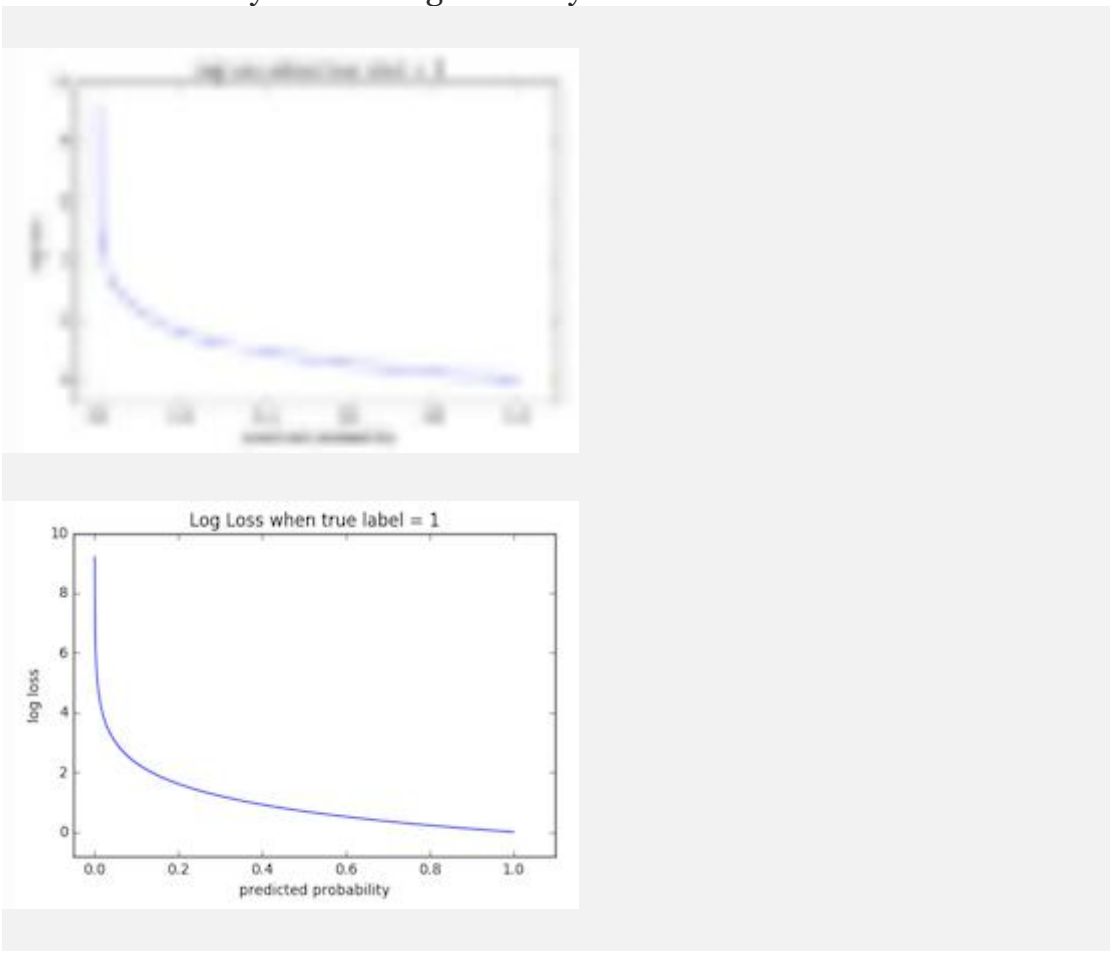
First, let's recollect the effect of ***y hat*** on probability again.



Sigmoid Function. Fig 17.

As we can see, for samples which have class 1, or positive label, at ***y hat*** is ~= 5, they are almost perfectly classified. So for all such samples, the ***delta y hat***s in the trees should be pushing ***y hat***s from 0 to 5. Hence delta y hat should be positive. It will be just the reverse for samples of class 0.

Second, the nature of Logistic loss function penalizes errors more than appreciating getting something right. Check the reference from ML Cheat Sheet on Logistic Loss and how yln(p) or (1-y)ln(1-p) penalizes errors. For a quick idea see the pic below which represents yln(p). (1-y)ln(1-p) is going to look the same as you can imagine surely.



yln(p) vs p. As you can see when p is less for label 1 samples, the loss is very high. Extreme.

This penalization and correction are ingrained in the model. At every split, XGBoost will try to correct the wrongly classified samples more than pushing the samples which are already doing well to their desired values.

If we have a bad split, which is against such penalization and correction, like above in example 2, it is possible, as you will notice, that the ***delta y hat***s fail to follow the desired nature of the movement. Both the child nodes have ***delta y hat***s as negative. Whereas if you notice example 1, the ***delta y hat*** for child 1, which has more of class 0 samples, had a negative value,

and the reverse for leaf 2, has more of class 1 samples. This also shows that we had a bad split.

Now, in an unbalanced class situation, sometimes, based on what we want to achieve, the above penalization and correction do not work out well, even in a good split. If you notice, child 2 from example 1 has only a 0.03 (without eta and 0.01 with eta) positive ***delta y hat*** movement. Whereas the child 1 has a 1.36 negative movement. This kind of restricted movement for minority class samples, is not desired, specifically, if we are more concerned with the accuracy of classification at a threshold as compared to an overall reduction of loss.

Let's discuss that a bit.

First, why is the ***delta y hat*** so small on the child or possible leaf with majority class 1 samples ?. It is because there is barely any overall G. Even though it is a leaf with majority fraction class 1 samples, there will always be some misclassified class 0 samples. As class 0 samples are so many in comparison to class 1 samples, even the misclassified ones can be as many as properly classifies class 1 samples — hence they will cancel out a lot of the overall G. Hence small ***delta hat***.

In contrary, the child with majority fraction class 0 samples, has a lot of samples. The overall gradient is heavily biased and positive (27.7 at K33). The misclassified class 1 samples are very few, to outweigh this gradient. And thus, the ***delta y hat*** is quite high and negative (-1.27).

So XGBoost is getting biased by the number of samples. It is trying more to correct the errors. And since there are so many class 0 samples, it is ignoring pushing the class 1 samples towards 1. It is more concerned about pushing the class 0 samples towards 0.

This will result in lowering LogLoss. But the class 1 samples, properly classified or not will kind of lie mixed up within a small band of probability,

along with wrongly classified class 0 samples. There will not be a strong threshold to separate them out.

If we are more concerned about the accuracy of classification or TPR at FPR, this above behavior does not work well. Mostly, in the unbalanced class situation, the positive class is rare. Like predicting an earthquake. Or classifying malware. These models need to operate at a low FPR, because calling a clean file as malware and deleting it, is far more severe than letting a malware go.

For such cases, it is more important to get the samples of the positive label, classified correctly, move away from the samples of the negative label, as far as possible. So that we have a dividing line or threshold with more TPR at a low FPRs across this dividing line.

It is not so important where that line is (between 0 and 1) and where the individual probabilities are. Even if all the class 0 samples are between 0.7 and 0.91 and all the class 1 samples are between 0.85 and 0.95, we can have a good threshold at say 0.9. But the log loss will be pretty bad (see above pic) because the class 0 samples are far from 0. But that may be OK in our use case.

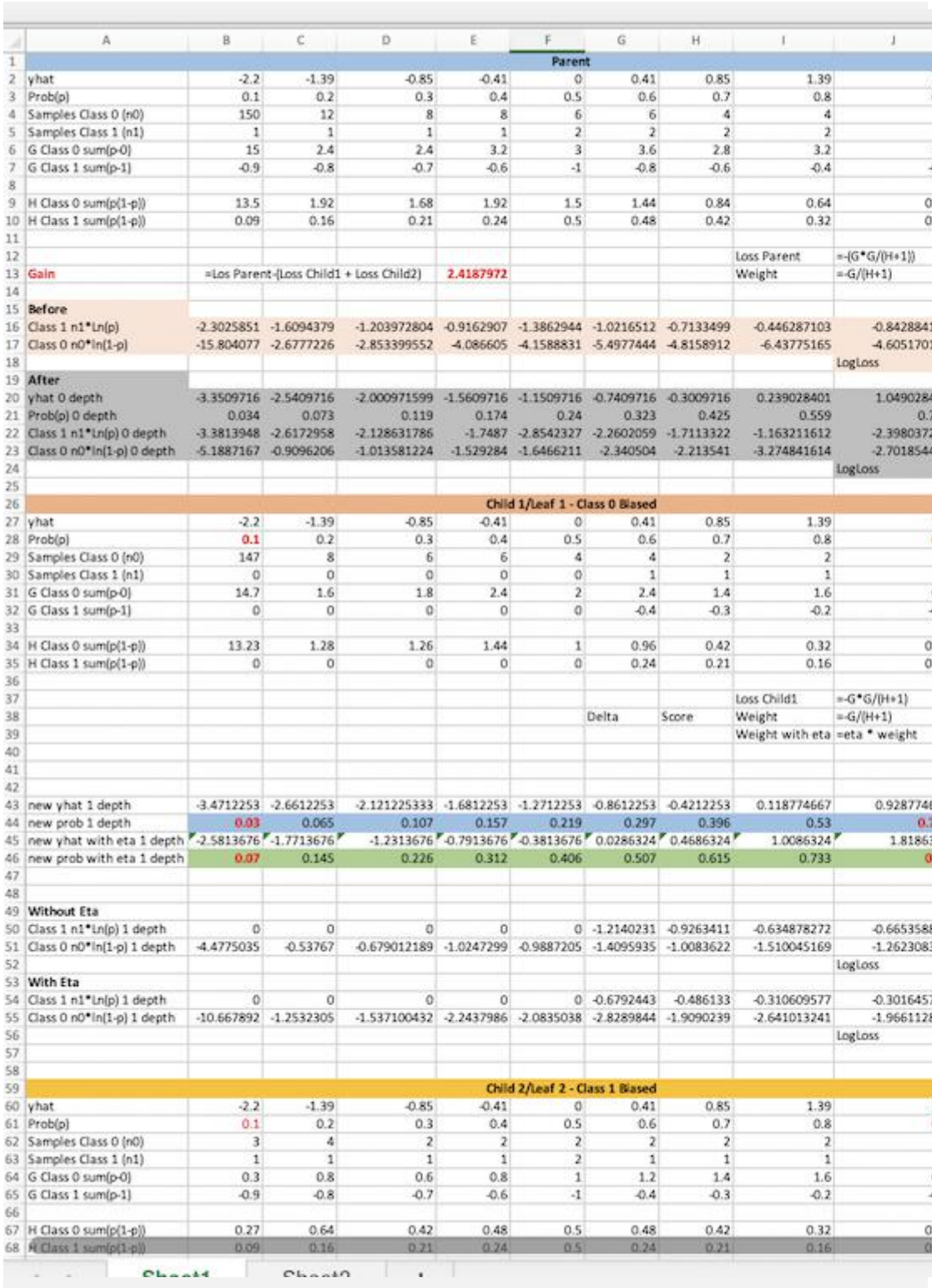
So sometimes, we want the algorithm to be more biased towards giving us better classification at a cutoff, than an overall reduction of LogLoss.

This is where ***scale_pos_weight*** comes in. To give more weight to the minority class, both in gradient and hessian, so that such samples move away from the majority class samples, sooner, in bigger leaps.

scale_pos_weight is defined as = s1 = (sum* of negative samples)/(sum* of positive samples). But there is no hard rule. It depends on how much importance you want to give to your minority samples as a whole compared to your majority samples. Some say, for highly imbalanced classes, scale_pos_weight should be s2 = sqrt(sum of negative examples)/sqrt(sum

of positive examples). We can choose values >s1 and <s2 and try out the one which fits our evaluation metric the best.

Let's look at example 3 below. Well, you cannot really look at it



	A	B	C	D	E	F	G	H	I	J	K	L
1						Parent					Total	
2	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85	1.39	2.2		
3	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9		
4	Samples Class 0 (n0)	150	12	8	8	6	6	4	4	2	200	
5	Samples Class 1 (n1)	1	1	1	1	2	2	2	2	8	20	
6	G Class 0 sum(p-0)	15	2.4	2.4	3.2	3	3.6	2.8	3.2	1.8	37.4	
7	G Class 1 sum(p-1)	-0.9	-0.8	-0.7	-0.6	-1	-0.8	-0.6	-0.4	-0.8	-6.6	
8										G	30.8	
9	H Class 0 sum(p(1-p))	13.5	1.92	1.68	1.92	1.5	1.44	0.84	0.64	0.18	23.62	
10	H Class 1 sum(p(1-p))	0.09	0.16	0.21	0.24	0.5	0.48	0.42	0.32	0.72	3.14	
11										H	26.76	
12									Loss Parent	=-(G*G/(H+1))	-34.172911	
13	Gain	=Los Parent-(Loss Child1 + Loss Child2)			2.4187972				Weight	=-G/(H+1)	-1.1509716	
14												
15	**Before**											
16	Class 1 n1*Ln(p)	-2.3025851	-1.6094379	-1.203972804	-0.9162907	-1.3862944	-1.0216512	-0.7133499	-0.446287103	-0.842884125	-10.442753	
17	Class 0 n0*ln(1-p)	-15.804077	-2.6777226	-2.853399552	-4.086605	-4.1588831	-5.4977444	-4.8158912	-6.43775165	-4.605170186	-50.937245	
18										LogLoss	61.379998	
19	**After**											
20	yhat 0 depth	-3.3509716	-2.5409716	-2.000971599	-1.5609716	-1.1509716	-0.7409716	-0.3009716	0.239028401	1.049028401		
21	Prob(p) 0 depth	0.034	0.073	0.119	0.174	0.24	0.323	0.425	0.559	0.741		
22	Class 1 n1*Ln(p) 0 depth	-3.3813948	-2.6172958	-2.128631786	-1.7487	-2.8542327	-2.2602059	-1.7113322	-1.163211612	-2.398037229	-20.263042	
23	Class 0 n0*ln(1-p) 0 depth	-5.1887167	-0.9096206	-1.013581224	-1.529284	-1.6466211	-2.340504	-2.213541	-3.274841614	-2.701854435	-20.818565	
24										LogLoss	41.081607	
25												
26						Child 1/Leaf 1 - Class 0 Biased						Total	
27	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85	1.39	2.2		
28	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9		
29	Samples Class 0 (n0)	147	8	6	6	4	4	2	2	1	180	
30	Samples Class 1 (n1)	0	0	0	0	0	1	1	1	2	5	
31	G Class 0 sum(p-0)	14.7	1.6	1.8	2.4	2	2.4	1.4	1.6	0.9	28.8	
32	G Class 1 sum(p-1)	0	0	0	0	0	-0.4	-0.3	-0.2	-0.2	-1.1	
33										G	27.7	
34	H Class 0 sum(p(1-p))	13.23	1.28	1.26	1.44	1	0.96	0.42	0.32	0.09	20	
35	H Class 1 sum(p(1-p))	0	0	0	0	0	0.24	0.21	0.16	0.18	0.79	
36										H	20.79	
37									Loss Child1	=-G*G/(H+1)	-35.212942	
38							Delta	Score	Weight	=-G/(H+1)	-1.2712253	
39									Weight with eta	=eta * weight	-0.3813676	
40												
41												
42												
43	new yhat 1 depth	-3.4712253	-2.6612253	-2.121225333	-1.6812253	-1.2712253	-0.8612253	-0.4212253	0.118774667	0.928774667		
44	new prob 1 depth	0.03	0.065	0.107	0.157	0.219	0.297	0.396	0.53	0.717		
45	new yhat with eta 1 depth	-2.5813676	-1.7713676	-1.2313676	-0.7913676	-0.3813676	0.0286324	0.4686324	1.0086324	1.8186324		
46	new prob with eta 1 depth	0.07	0.145	0.226	0.312	0.406	0.507	0.615	0.733	0.86		
47												
48												
49	**Without Eta**											
50	Class 1 n1*Ln(p) 1 depth	0	0	0	0	0	-1.2140231	-0.9263411	-0.634878272	-0.665358877	-3.4406014	
51	Class 0 n0*ln(1-p) 1 depth	-4.4775035	-0.53767	-0.679012189	-1.0247299	-0.9887205	-1.4095935	-1.0083622	-1.510045169	-1.262308381	-12.897945	
52										LogLoss	16.338547	
53	**With Eta**											
54	Class 1 n1*Ln(p) 1 depth	0	0	0	0	0	-0.6792443	-0.486133	-0.310609577	-0.301645779	-1.7776326	
55	Class 0 n0*ln(1-p) 1 depth	-10.667892	-1.2532305	-1.537100432	-2.2437986	-2.0835038	-2.8289844	-1.9090239	-2.641013241	-1.966112856	-27.13066	
56										LogLoss	28.908292	
57												
58												
59						Child 2/Leaf 2 - Class 1 Biased						Total	
60	yhat	-2.2	-1.39	-0.85	-0.41	0	0.41	0.85	1.39	2.2		
61	Prob(p)	0.1	0.2	0.3	0.4	0.5	0.6	0.7	0.8	0.9		
62	Samples Class 0 (n0)	3	4	2	2	2	2	2	2	1	20	
63	Samples Class 1 (n1)	1	1	1	1	2	1	1	1	6	15	
64	G Class 0 sum(p-0)	0.3	0.8	0.6	0.8	1	1.2	1.4	1.6	0.9	8.6	
65	G Class 1 sum(p-1)	-0.9	-0.8	-0.7	-0.6	-1	-0.4	-0.3	-0.2	-0.6	-5.5	
66										G	3.1	
67	H Class 0 sum(p(1-p))	0.27	0.64	0.42	0.48	0.5	0.48	0.42	0.32	0.09	3.62	
68	H Class 1 sum(p(1-p))	0.09	0.16	0.21	0.24	0.5	0.24	0.21	0.16	0.54	2.35	

https://github.com/run2/demystify-xgboost/blob/master/DeMystifyXGBoost-Example3.xlsx.

So this is what I have done.

I have taken the same 200:20 distribution. On the left, I have split the samples into two nodes without using scale pos weight. On the right I have split the samples into two nodes using the same samples per child node, but this time with additional weight given to the positive samples.

On the top left, under the Parent node, I have first shown the effect of a zero depth tree. If you notice, I used -G/H to adjust the prior ***y hat*** scores to get new ***y hat*** scores. I also calculated the prior Log Loss and new Log Loss assuming a zero depth tree. You will notice that the Log Loss decreases. The score adjustment is -1.15. It is not the same as -1.63 as discussed earlier for a 200:20 distribution because of the varying gradients. If you change all the probabilities to 0.5 in B3 to J3, you will see the -G/H value change to -1.63.

Coming back to the two child nodes, on the right side. Scale pos weight is used for calculating G and H. Everywhere for a class 1 sample, its presence in a calculation is weighted by a scale pos weight of 4.

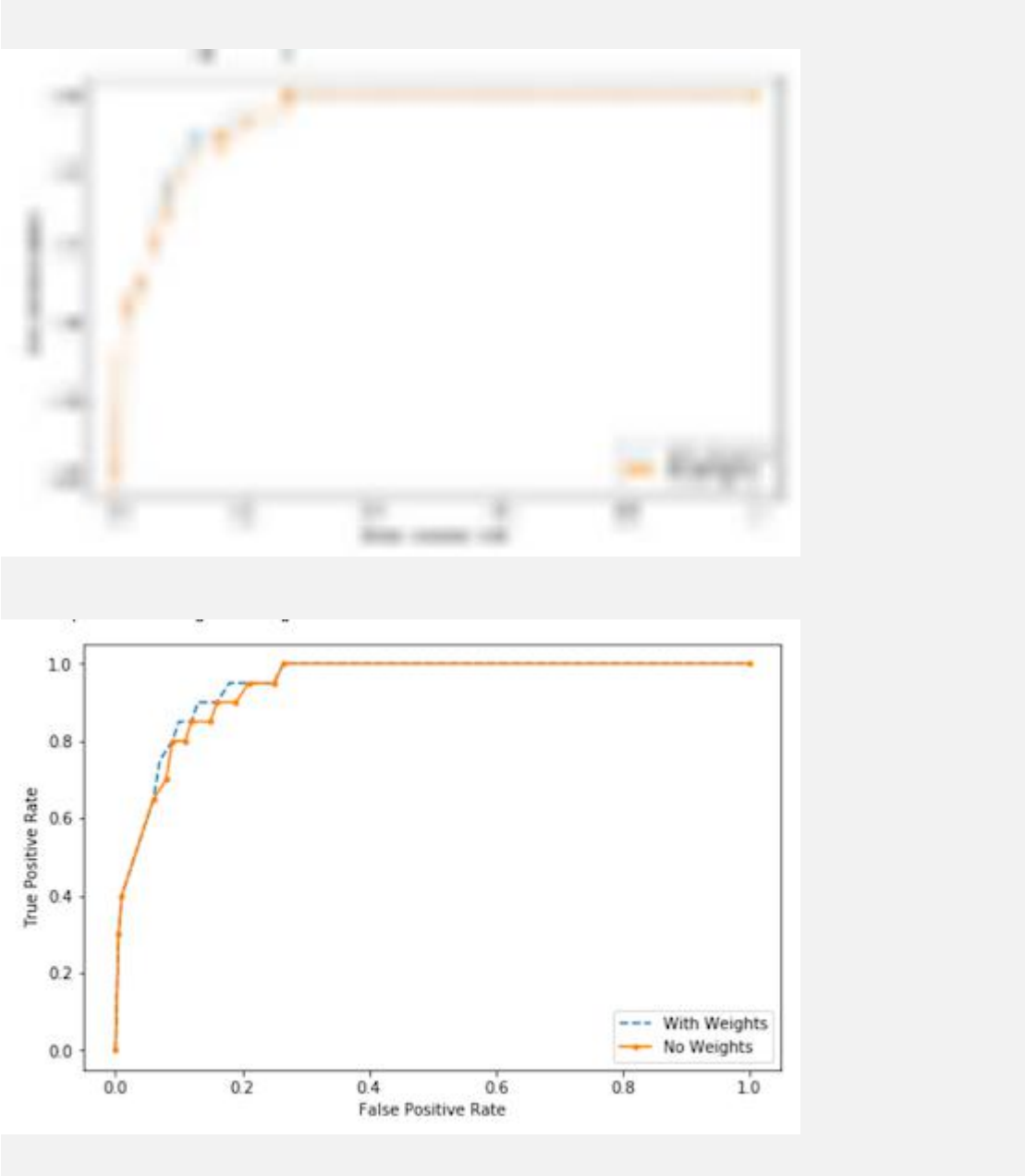
If you notice, Log Loss after split without scale pos weight reduces from 61 to 51. Whereas with scale pos weight, Log Loss reduces from 61 to 55. So evidently we are not doing very well on Log Loss reduction using scale pos weight. But what about the actual probabilities?

If we notice, on the left, for both child, the weight or delta y hat is negative. For child 1, we have a good positive G. But in child 2, even though it is a majority fraction class 1 node, the overall G could not be negative. Hence, for both the child nodes, delta y hat is only helping the class 0 samples.

On the right, it is a different picture though. We still have a negative G in child 1, but it is less pronounced. ***Delta y hat*** is -1 as compared to -1.27. But the magic happens in Child 2. Now we have a negative G. -13.4. Because of the additional weights on the positive label samples. This results in a positive ***delta y hat***, which will push the correctly classifies 1 samples towards higher ***y hat*** value and thus creates space between them and the others.

To prove my point, I then wrote some code to find the FPR/TPR/Thresholds for the two cases, with weight and without weights, also to plot their ROC. (You will find the code in Part IV of this series). For these calculations, I needed the final probabilities of the samples. I assumed that both the child nodes end up as leaves and hence their final probabilities can be derived from their prior scores (B2 to J2 or N4 to V4) added to the ***delta y hat***s (K39 K72 for children without weight and W41 W74 for children with weight).

This is what the ROC looks like. With and without Scale Pos Weight.



ROC plot for Example3 samples. With scale pos weight and without.

As you can see, the model with scale pos weight trumps the one without scale pos weight in TPR vs FPR at early FPR numbers. This is just one tree one deep. The difference is more pronounced when scale pos weight is used throughout the model of many trees. The AUC came out to be 0.956875 for the model with Scale pos weight, and 0.941375 for the one without scale pos weight.

The same concept applies if rather than using scale pos weight, individual samples are weighted. In that case, the gradient and hessian of each sample in each leaf is multiplied by the sample weight. Hence gi = wi(pi-yi) and hi = wi(pi)(1-pi) for the ith sample. For multi-label classification, we can adjust weights for all the samples as required.

One question might come up at this point. If and when we are bothered by AUC, why don't we minimize it, rather than LogLoss. The answer is, we cannot because AUC is not something we can differentiate and equate to zero. Then again AUC values can be derived at different operating points (i.e FPR values). If we target to optimize a model just to increase the AUC within an operating FPR range, the trees will be overfitting that specific criterion and hence will lose the generic nature. Hence by using weighting, we can increase AUC while remaining within the context of optimizing Log Loss.

Training Class Distribution vs Real Class Distribution

One last bit before we wrap up this topic. How does all that we learnt, work when the Training class distribution is different from Real class distribution? In the above example of 200:20 training class distribution, what if the real time unseen distribution is different? significantly? like 50:50. Or for that matter, in any such training, the class distribution of training set (validation set) and unseen real time data is significantly different?

Hopefully, you can understand now, (especially if you have played around with the excel sheets) the math and minimization of loss are sensitive to the class balance. So, the models will not perform as expected (and validated)

on the real time unseen distribution if the class distribution is significantly different.

If we know that the unseen data to predict on, is unbalanced, we will need to have a fair idea of how unbalanced it is before we start using scale pos weight. It is not necessary that the training data must have the same distribution as that of the unseen. More often, we have lots of retrospective training data, where we can have a greater number of the minority class samples. In such cases, we can either keep using the entire data and suitably adjust the weights or scale pos weight, or we can randomly down scale a class to match the unknown real time data class distribution and create multiple models and ensemble.

For e.g, if we have a 200:40 distribution in training, but we know we have a 200:20 distribution in unseen data, we can create two models of 200:20 distribution and ensemble them. Or, we can continue using the 200:40 training data set, but change scale pos weight to sqrt(200)/sqrt(20)*1/2. That last 1/2 factor being introduced because there is double the number of positive samples in the training set as in real data.

This means, as you get more and more of minority class samples in your training data, you can slowly reduce your scale pos weight. You will get comparable AUC on your validation set and holdout set, or training data.

The main thing to look out is, the holdout set should ALWAYS have the same class distribution as the real unseen data. That way before releasing a model you can always verify whether AUC or LogLoss, whatever is your evaluation metric, is performing as desired on your holdout set.

As I write, I feel, maybe I will add a part 6 to my series to talk in detail about how to do deal with imbalanced data in XGBoost. In depth.

Thanks

That's it guys, in this post. The next one will show how to quickly build a docker container where you will be able to run almost any ML stuff and use Jupyter and R in Jupyter Lab. It will be the basis of the code for Part IV where I will use real data and build models to show all that we talked about in Part I and Part II.

References

All resource files can be found at

https://github.com/run2/demystify-xgboost/tree/master