# A Quick Tour of XGBoost

*& Understanding Why It Works So Well*

Keifer Lee
keifer@nymlr.com

August 31st, 2026

## Contents



## 1 What is This for?

In the world of applied ML, a good chunk of the tasks fall into the domain of tabular data thanks to how handy the tabular form is. It shows up everywhere, from banking [8], to marketing [9], healthcare [10], and retail [11], just to name a few – it's simply a tabular world out there! Therefore, this is a well-studied domain with a plethora of known approaches to tackle it at scale, and one of the most successful ones with widespread adoption and great baseline performance, is the Gradient Boosted Decision Trees (GBDT) class of methods, of which XGBoost is one of its most popular instantiations. A short, non-exhaustive listicle of where it shows up in practice:

- Kaggle dominance – in winning tabular solutions since 2015 [1, 3, 4]
- Higgs boson challenge, high-energy physics [5]
- Credit scoring and loan default forecasting [6]
- Medicare fraud detection [7]
- The default tabular baseline in benchmarks [12–14]

- Still ahead of tabular foundation models past $\sim 10^4$ rows [15, 16]
- Ahead of tuned deep nets and deep sequence models at scale [13, 14]

The question then arises, what makes them such effective algorithms and how do they work under the hood? (Per my own experience, the concept and theory of Decision Trees [19] are widely known amongst practitioners as it's widely taught in schools, GBDT however, much less so!) This short essay sets out to answer this exact question from an applied perspective instead of a dense rigorous treatment, as there are already many existing texts on that [17, 18], and I am personally much more interested in knowing what makes it tick and the reason behind its unreasonable effectiveness out-of-the-box. The end goal is to provide a sufficient technical exposition in an intuitive yet grounded way, and then leverage that understanding to answer the aforementioned question, which hopefully will help my fellow practitioners make better decisions these tools in their day-to-day work.

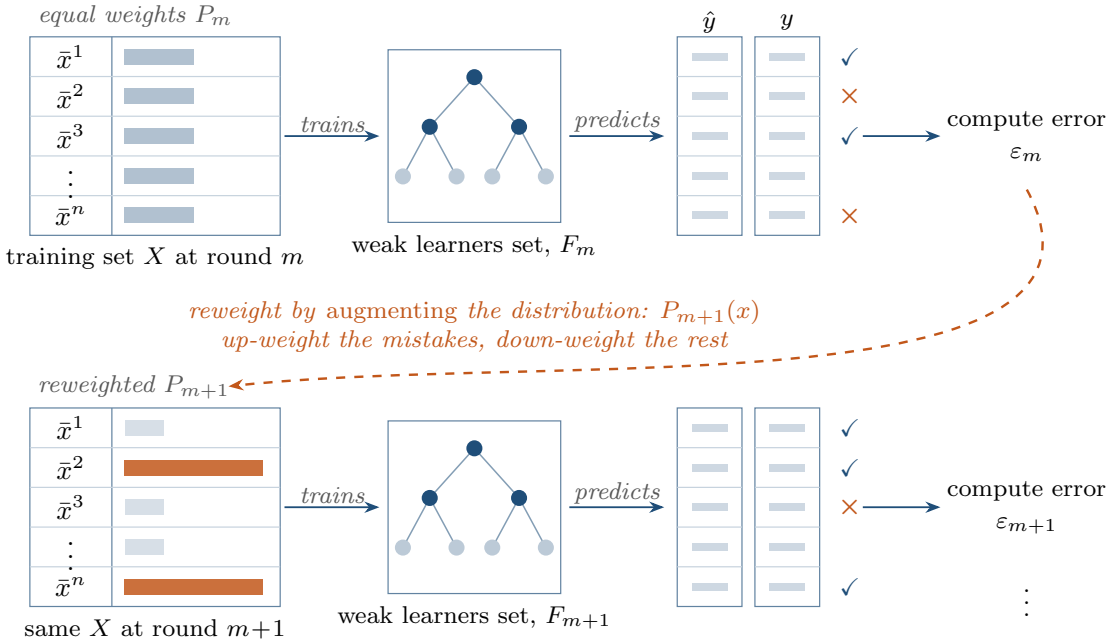
## 2 A Little Historical and Ontological Detour

The idea of iterative refinement by *combining* a collection of weak learners to yield a stronger learner is a pretty intuitive one in retrospect, and this is exactly how GBDT works – it *combines* many Decision Trees [19] into a "super-tree" in a specific way such that the super-tree is more powerful than its constituents.

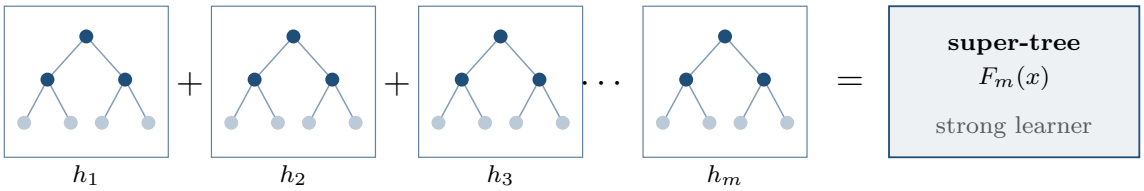
One of its roots can be traced back to Kearns and Valiant (1989; J. ACM 1994) [20] in PAC learning (which has one of the funniest names in my opinion – Probably Approximately Correct Learning, ha!) when they tussle with the question of whether it's possible to create a strong learner (model) with arbitrarily low error by combining only weak learners, that individually, are only marginally better than random!

Turns out – yes you can [21], and this gave birth to the ensemble class of methods we all know and love today. I'll skip over straight to the landmark result – for brevity – of AdaBoost (Freund and Schapire, 1997) [22], and this miraculous algorithm, although simple in concept, was shown to be able to drive training error to 0 exponentially-fast, and more interestingly its test time error continues to improve even after the training error has hit 0 [23]! In short and with great deal of simplification, AdaBoost works by maintaining a distribution over its training samples. As the successive weak learners get trained and "combined" into the wider structure of the super-learner, the training samples used to fit each successive weak learners are reweighted (up or down weighted) so as to prioritize – give a *boost* (!) – the samples where it's making mistakes. The idea, is to learn with more *emphasis* on the "hard" samples to get them right next time; for output, it uses a simple weighted voting mechanism amongst weak learners (each weighted by a scalar, $\alpha_m$) to decide on what the final value of the super structure would be. Figure 1 illustrates this process – pretty straightforward!

*(a) A single boosting round where one new weak learner (tree) is added to the set, $F_m$*




*(b) Final super-structure by ensembling (cumulative summing) of all the trees*




*each tree, $h$ is fit on the* reweighted *data of its own round, weighted by $\alpha_m$ (set by its error $\varepsilon_m$) and added to the set: $F_m = F_{m-1} + \alpha_m h_m$*

**Figure 1:** AdaBoost visualized. (a) Shows iterative round $m$ and $m+1$ and how each weak learner's training sample weights, $P_m$ is conditioned on the error from the previous round's learner, $\varepsilon_{m-1}$ (b) Illustrates how these successive weak learners (individual trees), $h_{1,\ldots,m}$ are then ensembled to create the final *strong-learner*, $F_m(x)$.

Then Friedman (2001) [25] took it a step further, and suggested the idea of fitting each new weak learner to the negative gradient of the loss with respect to the pseudo-residuals, for any differentiable loss, to drive the iterative adaptation process – hence, Gradient Boosting! This essentially recast the solution from a reweighting one, to a numerical optimization task. Written out, the ensemble after $m$ rounds is just the previous ensemble plus one more tree, and that tree is itself just a piecewise-constant function over the regions its splits carve out:

$$F_m(x) = F_{m-1}(x) \; + \; \eta \, h_m(x), \tag{1}$$

$$h_m(x) = \sum_{j=1}^{T_m} w_{mj} \, \mathbf{1}\{x \in R_{mj}\} \, . \tag{2}$$

where

| | |
|---|---|
| $x$ | is the feature vector of a single sample |
| $m$ | is the index of the current boosting round, $m = 1, \ldots, M$ |
| $F_m$ | is the ensembled ("super-tree") after $m$ rounds |
| $F_{m-1}$ | is the cumulative output from all the learner set of prior rounds $[0, \ldots, m-1]$ |
| $h_m$ | is this round's weak learner (a single tree) |
| $\eta$ | is the learning rate (or *shrinkage*) |
| $T_m$ | is the total number of leaves in round $m$'s tree |
| $R_{mj}$ | is the $j$-th leaf *region* – an axis-aligned box carved into the feature space by the nodes and leaves of this round's tree |
| $w_{mj}$ | is the leaf weight, a constant value assigned to the tree outputs on $R_{mj}$ |
| $\mathbf{1}\{\cdot\}$ | is the indicator function: 1 when $x$ falls in that region, 0 otherwise |

Note that for a single tree at round $m$, the regions $R_{m1}, \ldots, R_{mT_m}$ are disjoint and cover the whole feature space, so for any given sample on the manifold, $x$ it will only fall into a single region and hence be associated with only a single leaf weight as per (2). *Across* trees however, the regions $R_{mj}$ from different rounds $m$ can overlap and it is this cumulative iterated segmentation that produces the final margin. Then simply apply a threshold, and voila(!) out pops the class.

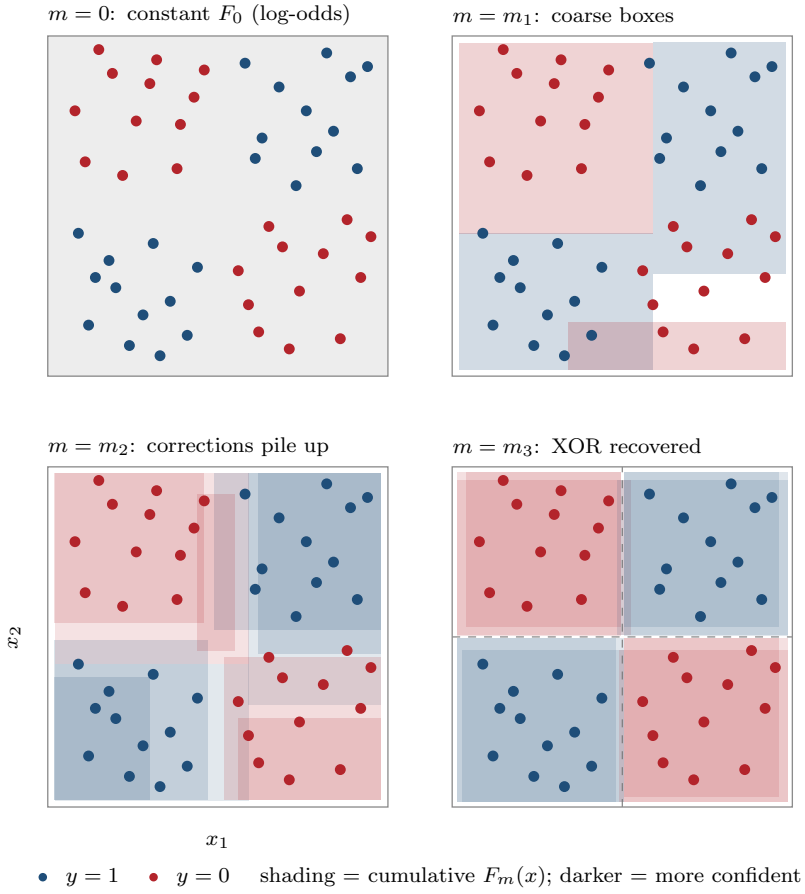



**Figure 2:** Here, I provide an illustration of this learning process on the 2-dim XOR problem. At $m = 0$, we simply initialize $F_0$ with the log-odds of the classes, which is simply 0 in this case (balanced classes). Then at successive rounds, it can be seen that the learned class boundaries get better and better, as more trees are added to the model (super-tree), and finally at round $m_3$, we stop once it gets good enough. Note that the boundaries are *axis-aligned*, perfect rectangles, as induced by (2).

# 3 XGBoost Proper

Now we have a GBM (Friedman, 2001) [25] that is nice to deal with in the form of a familiar optimization problem, where we can "intelligently" learn an "optimal" ensemble (boosting) model by stacking weak learners in an end-to-end manner – and indeed it's pretty effective! However, it still has some clunky bits and gaps, and it took Chen and Guestrin in 2016 [1] to finally bridge those gaps with the XGBoost that we know and love today. There are a lot of engineering tricks and clever reformulation that made XGBoost so much better than GBM, and so well adapted to real-world, large-scale workload we see in practice, but in the interest of time, I'll focus only on the core, and most consequential bits, providing a necessary and sufficient exposition to the question of *"how does it work?"*

## 3.1 Reformulation as a Regularized Objective for Natural Regularization

Friedman's GBM already had shrinkage, subsampling and tree-size limits, and Newton-step boosting existed (LogitBoost [24]). What XGBoost did was fold the explicit penalties $\gamma T + \frac{1}{2}\lambda\|w\|^2$ into the second-order objective so that leaf weights and split gains come in as a single closed-form parcel, and most importantly, did a lot of polishing to engineer the whole thing to scale. By reformulating the learning task as a regularized objective, it in one shot naturally deals with regularization and adverse spurious sample selection in a single pass, instead of a cascade of steps, enabling stabler learning even with tricky dataset.

### The objective function

For a regularized objective at round $m$, and with the previous $m-1$ trees frozen (so their $\Omega$ terms are constant and dropped), we have

$$\mathcal{L}^{(m)} \;=\; \sum_{i=1}^{n} \ell\Big(y_i,\; \hat{y}_i^{(m-1)} + h_m(x_i)\Big) \;+\; \Omega(h_m) \tag{3}$$

where, $i$ is the sample index, and $\hat{y}_i^{(m-1)}$ is the output from $F_{m-1}$. Also, for simplicity, we're just gonna deal with a binary classification task, therefore the loss is simply the Binary Cross-Entropy.

$$\ell(y_i, \hat{y}_i) \;=\; -\,y_i\hat{y}_i \;+\; \ln\Big(1 + e^{\hat{y}_i}\Big)\,. \tag{4}$$

### The regularizer $\Omega(\cdot)$

$$\Omega(h_m) \;=\; \gamma T \;+\; \tfrac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \;+\; \alpha \sum_{j=1}^{T} |w_j| \tag{5}$$

$\Omega(h_m)$ is the regularization term for learner $h_m$ (writing $T$ and dropping subscript for brevity), and can be broken down as follows:

- **Leaf-count penalty** $\gamma T$ (`gamma`) – penalizes the *number* of leaves. This is a simple additive cost per leaf on the number of total leaves, $T$, so as to prevent over-splitting of nodes and to make sure each split is done only with sufficient loss reduction in return.

- **L2 shrinkage** $\frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2$ (`lambda`) – penalizes the *magnitude* of the assigned leaf value. It effectively tries to *push* each $w_j$ toward 0, so more emphasis is placed on meaningful samples only. This is most aggressive where $H_j$ (Hessian or curvature) is small (lots of spurious samples).
- **L1 sparsity** $\alpha \sum_{j=1}^{T} |w_j|$ (`alpha`) – also penalizes magnitude and additionally encourages sparsity: it can drive a $w_j$ exactly to 0.

As a whole, the above should be a familiar expression for those with some optimization background: this is just a regularized objective. As an aside, the $\frac{1}{2}$ on the L2 term is just a mathematical nicety – it cancels on differentiation and keeps the later expression for $w^*$ tidy for those picky statisticians. Also, everything below assumes $\alpha = 0$.

### 3.2 From First Order to Second Order Estimation

XGBoost also uses 2nd order optimization rather than 1st order so as to account for not only the gradient, $G$, but also the curvature of the loss landscape, $H$ (the Hessian, in fancy-speak). This is super helpful throughout the learning process, such that it enables the learners to not only determine the direction of steepest descent, but also one that's nicely conditioned. To see why it matters, let's define the gradient and curvature terms first:

#### Gradient $G_i$ and Hessian $H_i$

Both are derivatives of the (binary cross entropy) loss *with respect to the model's output*, evaluated at the current margin, $\hat{y}_i^{(m-1)}$:

$$G_i = \left.\frac{\partial \ell(y_i, \hat{y})}{\partial \hat{y}}\right|_{\hat{y}=\hat{y}_i^{(m-1)}} = \sigma\left(\hat{y}_i^{(m-1)}\right) - y_i = p_i - y_i, \tag{6}$$

$$H_i = \left.\frac{\partial^2 \ell(y_i, \hat{y})}{\partial \hat{y}^2}\right|_{\hat{y}=\hat{y}_i^{(m-1)}} = p_i\,(1 - p_i). \tag{7}$$

where

- $G_i \in (-1, 1)$ – a signed error. $G_i < 0$ means the margin must increase and vice versa.
- $H_i \in (0,\, 0.25]$ – the local curvature, maximal at $p_i = 0.5$ (most uncertain). This effectively acts as the support mass of a leaf – large where the model is least confident, and it *dampens* the step $w^*$ there rather than amplify it, and vice versa.

Basically, the gradient provides the signal of "where to go" with respect to the current loss landscape, and the curvature then characterizes where the most *interesting* regions are, given what the model currently know (and don't know).

#### Optimal leaf weight $w_j^*$

Next, we see where $G$ and $H$ is directly applied - specifically in computing the optimal leaf weights $w_j^*$ of each leaf $j$ for a particular tree, $h_m$. Very conveniently, each leaf $j$ contributes $G_j w_j + \frac{1}{2}(H_j + \lambda) w_j^2$ to the objective (we'll derive this in (13) next section) where $\lambda > 0$ is a

regularizing term. Therefore, by setting the derivative to zero, the optimal weight $w_j^*$ admits a closed form solution:

$$\frac{\partial}{\partial w_j}\left[G_j w_j + \tfrac{1}{2}(H_j + \lambda) w_j^2\right] \;=\; G_j + (H_j + \lambda) w_j \;=\; 0 \tag{8}$$

$$\boxed{w_j^* \;=\; -\frac{G_j}{H_j + \lambda} \;=\; -\frac{\sum_{i \in I_j} (p_i - y_i)}{\sum_{i \in I_j} p_i (1 - p_i) + \lambda}} \tag{9}$$

Obviously, the second derivative is $H_j + \lambda > 0$, so this is a minimum. Note that $\lambda$ serves to dampen the weights of leaves built on little Hessian mass (e.g. nothing much to learn). So now, we have a way to condition the learning on the "informativeness" of certain samples at given round with $H$, and *control* for how much emphasis should be placed on it with $\lambda$.

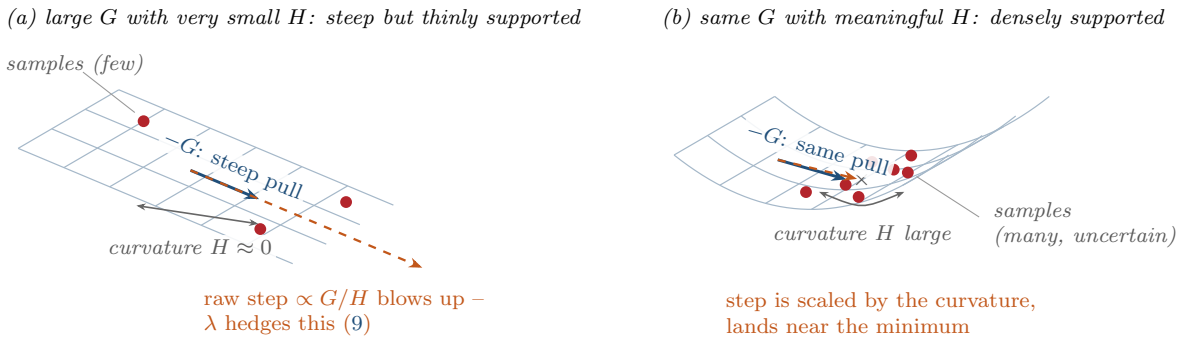



**Figure 3:** This figure illustrates the importance of considering the curvature, $H$ in learning and the role $\lambda$ plays. In both (a) and (b), the gradient, $G$ is the same, however in (a) the curvature is $\sim 0$ with thin support, comprised of confidently classified samples (*confidently wrong* for spurious samples), leading the model to take large overconfident steps without regards for the *saliency* of the data driving the learning. In (b), we have another area of the feature space where it's better supported with more uncertain (*salient*) samples and hence large $H$, leading to a better minimum with just the right step-size.

## 3.3 Very Smart Plumbing

Chen and Guestrin really went the extra mile to make sure that XGBoost is practically useful. One good example, is how it puts into practice what we've just covered, to efficiently evaluate the gradient, $G$ and curvature $H$, in its objective function!

### Second-order surrogate

As nice as the objective (3) is, it unfortunately has no closed-form minimizer over tree structures and weights, and hence we need some way to go around this; the common toolkit to solve such functions are iterative and slow; this would be a tough sell for learning with large-scale dataset, which is so prevalent in the wild (e.g. transactions at Stripe each hour, number of Uber bookings per day). Therefore, XGBoost introduced a surrogate as bypass, by simply taking the second-order Taylor expansion of the function, which can be easily evaluated directly and is sufficiently representative at local-scale (which our Trees are operating at). Taylor-expand (3) to second order around $\hat{y}_i^{(m-1)}$ and drop the constant $\ell(y_i, \hat{y}_i^{(m-1)})$, and that yields:

$$\tilde{\mathcal{L}}^{(m)} = \sum_{i=1}^{n} \left[ G_i \, h_m(x_i) + \tfrac{1}{2} H_i \, h_m^2(x_i) \right] + \gamma T + \tfrac{1}{2} \lambda \sum_{j=1}^{T} w_j^2 \tag{10}$$

Every sample in leaf $j$ receives the same output $w_j$, so regroup the sum w.r.t. each leaf, then

$$\tilde{\mathcal{L}}^{(m)} = \sum_{j=1}^{T} \left[ \Big( \sum_{i \in I_j} G_i \Big) w_j + \tfrac{1}{2} \Big( \sum_{i \in I_j} H_i + \lambda \Big) w_j^2 \right] + \gamma T \tag{11}$$

Now, let's make this a bit nicer to write by defining the **leaf aggregates**

$$\boxed{G_j = \sum_{i \in I_j} G_i, \qquad H_j = \sum_{i \in I_j} H_i} \tag{12}$$

where $i$ are samples at node $I_j$, and therefore

$$\tilde{\mathcal{L}}^{(m)} = \sum_{j=1}^{T} \left[ G_j w_j + \tfrac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T. \tag{13}$$

Take a moment and notice that (13) is $T$ one-dimensional quadratics in $w_j$. The samples index, $i$ have vanished and a leaf is now just the pair $(G_j, H_j)$. This makes downstream terms much easier to organize and compute exactly with (6), (7) and (9)!

**Structure score**

We can go further, and use the same idea to also determine how a tree should *split* at each node with what we have already derived. Substitute $w_j^*$ back, and summing over all leaves we can see that the tree scores exactly

$$\boxed{\tilde{\mathcal{L}}^*(q) \;=\; -\frac{1}{2} \sum_{j=1}^{T} \frac{G_j^2}{H_j + \lambda} \;+\; \gamma T} \tag{14}$$

This is the **quality of a tree structure** $q$ (tree spawned at round $m$) – of course, lower is better. It plays the role that impurity / Gini plays in a classification tree, but derived from the actual loss directly. And from the above, we can also define the *per-node* score, $\mathcal{S}(I)$ as

$$\mathcal{S}(I) \;=\; \frac{G_I^2}{H_I + \lambda}. \tag{15}$$

What we have done so far, is to show that at the heart of XGBoost, the optimal value of a leaf and the quality of a learned tree (weak-learner) are explicitly parameterized as functions of the first, $G$ and second order, $H$ statistics for any twice-differentiable loss function - very nice! Next, we need a way to *compare* before *vs* after a split to quantify whether a split is worth it or not. We do so by introducing a new function called the *Gain* as such

$$\text{Gain} \;=\; \underbrace{\tilde{\mathcal{L}}^*(\text{before})}_{\text{unsplit}} \;-\; \underbrace{\tilde{\mathcal{L}}^*(\text{after})}_{\text{split}} \tag{16}$$

Consider splitting node $I$ into $I_L$ (left) and $I_R$ (right), such that $I = I_L \sqcup I_R$ (disjoint), and because $G_I$ and $H_I$ are simply sums of $G_i$ and $H_i$, therefore

$$G_L + G_R = G_I \qquad H_L + H_R = H_I \tag{17}$$

This makes it trivial to recover either side once we have accumulated $G$ and $H$ for one side. Also, as one node splits into two, $T$ increases by exactly 1 thereby accruing additional penalty of $1 \times \gamma$. By substituting (14) into the Gain equation and by making use of (17), we arrive at the full Gain equation as follows

$$\boxed{\text{Gain} \;=\; \frac{1}{2}\left[\underbrace{\frac{G_L^2}{H_L+\lambda}}_{\text{left child}} + \underbrace{\frac{G_R^2}{H_R+\lambda}}_{\text{right child}} - \underbrace{\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}}_{\text{parent, unsplit}}\right] \;-\; \gamma} \tag{18}$$

where

- **Left child score** $\dfrac{G_L^2}{H_L+\lambda}$ – how much loss the left branch can reduce
- **Right child score** $\dfrac{G_R^2}{H_R+\lambda}$ – how much loss the right branch can reduce
- **Parent score** $\dfrac{(G_L+G_R)^2}{H_L+H_R+\lambda}$ – how much loss the parent node can already reduce by itself without splitting
- $\boldsymbol{\gamma}$ – the additional leaf-penalty we've incurred by splitting one-to-two

XGBoost's simple yet effective strategy is to simply greedily pick the action with largest Gain! Notice the Gain is zero only when the two children want the **same** weight, $G_L/(H_L+\lambda) \approx G_R/(H_R+\lambda)$, in which case one shared weight already captures it and the split is pointless. Opposite-sign gradients, where the L/R branches want to move the margin in different directions are the clearest case of disagreement, meaning there's something worth splitting on here, and same-sign gradients with different magnitudes or curvature also produce positive Gain. In order for a split proposal to be accepted, by default, it needs to have a positive Gain value, such that

$$\max \text{Gain} > 0 \quad \Longleftrightarrow \quad \frac{1}{2}\left[\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G_I^2}{H_I+\lambda}\right] \;>\; \gamma, \tag{19}$$

so $\gamma$ is literally the minimum loss reduction a split must induce. The **routing rule** is then simply: for samples where $x_{k^*} < v^*$ go left, else right, subjected to two additional structural constraints:

$$\min(H_L,\, H_R) \;\geq\; \texttt{min\_child\_weight}, \qquad \text{depth} \;\leq\; \texttt{max\_depth}. \tag{20}$$

`min_child_weight` is a cutoff on **Hessian mass** so that only gradient induced by sufficiently meaningful or salient samples, not sample count, will be considered in the Gain equation; there's not much meaning creating more splits on already well-classified data points. For example, a child of 50 confidently-classified samples ($H_i \approx 0$) will fail the cutoff, while a child of 5 uncertain ones passes and can have a positive Gain since there's something worth learning here still. `max_depth` is pretty self-explanatory, it's just the maximal depth limit of the tree, $h_m$ that we

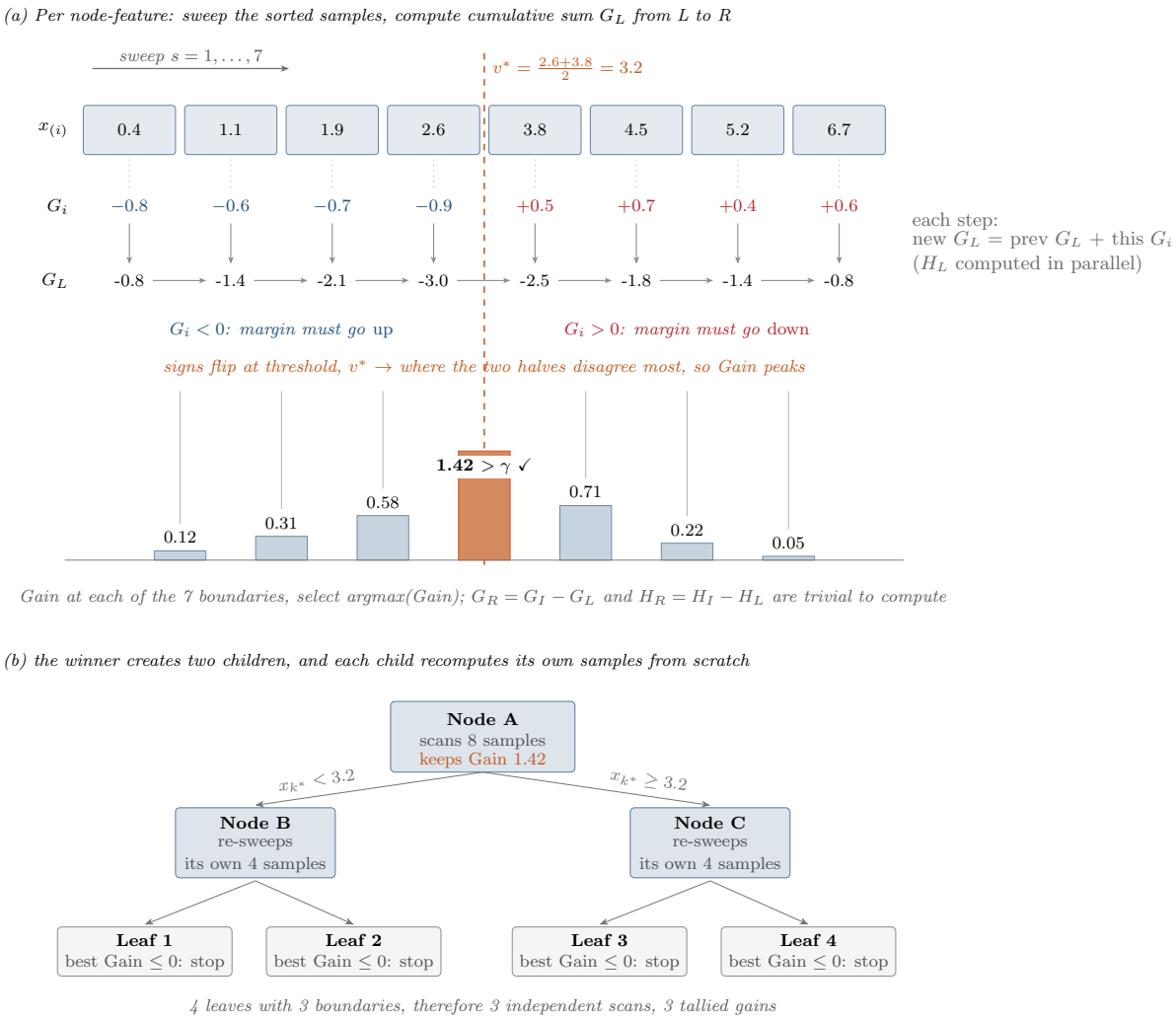



**Figure 4:** Split finding illustrated (a) shows the ops for a given node-feature pair; here the feature vector $x_k$ is first pre-sorted in ascending order, and $G_i$ is computed per sample. $G_L$ is simply the running sum and $G_R$ is the complement or residual of $G_I$. Once computed, the split threshold is decided via the Gain at its maximum; every split candidate threshold, $v$ therefore costs only $O(1)$ in this line-search. (b) illustrates how the samples are then routed to the node's children as per the rule $x_{k^*} < v^*$, and inheriting only the samples $x$ that qualifies. Recursion stops once the max depth is reached or there's no further way to split with Gain $> 0$. Note that a naive GBDT would re-sort at every child node; XGBoost pre-sorts each feature only once and the children simply re-sweep. $H$ is computed in parallel but not shown above for clarity.

explicitly define and enforce; this is to prevent overfitting since a tree of depth $D$, can model interactions up to order of $D$ [17].

In effect, all these mechanics replaces tedious per leaf optimization with simple closed form evaluation with just 2 computed scalars ($\lambda$ and $\gamma$ are fixed) that we can precompute in a batch ahead of time, and thus when it comes time to determine the split at a given node $j$, we can just do a series of fetch-and-sum! Not only that, with some clever arrangements, we can find the "optimal" split in one pass per feature with only iterative sum ops as shown below in Figure 4 for a toy 8-sample dataset.

At a node with sample set $I$, for each feature $k \in \{1, \ldots, d\}$:

1. Take $I$ in ascending (by convention) order of $x_{ik}$ (pre-sorted once, globally; the node only walks its own subset): $x_{(1),k} \leq x_{(2),k} \leq \cdots \leq x_{(|I|),k}$.
2. Precompute totals $G_I = \sum_{i \in I} G_i$, $H_I = \sum_{i \in I} H_i$.
3. Initialise $G_L \leftarrow 0$, $H_L \leftarrow 0$.
4. Then sweep the boundaries $s = 1, \ldots, |I| - 1$ and assign:

$$G_L \leftarrow G_L + G_{(s)}, \quad H_L \leftarrow H_L + H_{(s)}, \quad G_R \leftarrow G_I - G_L, \quad H_R \leftarrow H_I - H_L,$$

5. Compute Gain and greedily select over all features, $k$ at threshold $v$ that passes the cutoff:

$$(k^*, v^*) \;=\; \arg\max_{k,\, v} \; \mathrm{Gain}(k, v). \tag{21}$$

All this turns what could've been a $O(|I|^2)$ operation per feature per node if done naively (rescanning all $|I|$ samples at each of the $|I|-1$ boundaries), into a $O(1)$ procedure per boundary, and an $O(|I|)$ sweep per feature per node after a one-off $O(n \log n)$ sort per feature – very nice! The above, in turn, enables / unlocks more downstream optimization goodies such as cache-aware gradient prefetching, distributed workflow and more.

There's a slight wrinkle however in my explanation above due to simplifications for clarity. The exact sweep over every one of the $|I| - 1$ boundaries, as I described, is the *greedy* recipe of plain GBDT (and XGBoost's `exact` method). XGBoost also offers an approximation method called **weighted quantile sketch** (used for distributed / out-of-core training; the modern default `hist` uses a related histogram binning) – a method leveraging smaller sample approximation of the full thing for faster computation. Formally, for feature $k$ define the Hessian-weighted rank $r_k(z) = \sum_{i:\, x_{ik} < z} H_i \,/\, \sum_i H_i$, then pick only a handful of candidate thresholds $\{v_{k,1}, \ldots, v_{k,l}\}$ such that adjacent candidates differ in rank by less than some $\epsilon$, giving roughly $1/\epsilon$ candidates to sweep instead of $|I| - 1$. Intuitively, the idea is to build a histogram *"sketch"* with bins that hold equal amounts of *uncertainty* ($H$) instead of counts, and only test the bin edges, which is a much smaller set to sweep over!

### 3.4 Inbuilt Handling of Missing Values

Last but not least of the things to highlight, XGBoost has inbuilt mechanism to deal with missing value in an effective way (super helpful in almost all real world dataset where data is never completely observable). In short, at each node, it checks which branch is best to put the missing value data points to (L or R), and directly dumps all the missing value points to the side that maximizes the Gain.

$$\mathrm{Gain}^* \;=\; \max\Big(\, \mathrm{Gain}\big(I_L \cup I_{\text{missing}},\; I_R\big),\;\; \mathrm{Gain}\big(I_L,\; I_R \cup I_{\text{missing}}\big)\Big) \tag{22}$$

This is obvious in hindsight and almost too simple to believe, but it turns out to be very effective in practice, making XGBoost super accommodating to sloppy datasets.

Okay, the above is a lot to take in, and indeed deserves a more thorough treatment to fully illustrate its depth and profoundness. However, I'll stop here and leave it as further reading, or perhaps a fuller in-depth essay if there's a demand for it. Figure 5 provides an overview of what we've discussed here to summarize things.

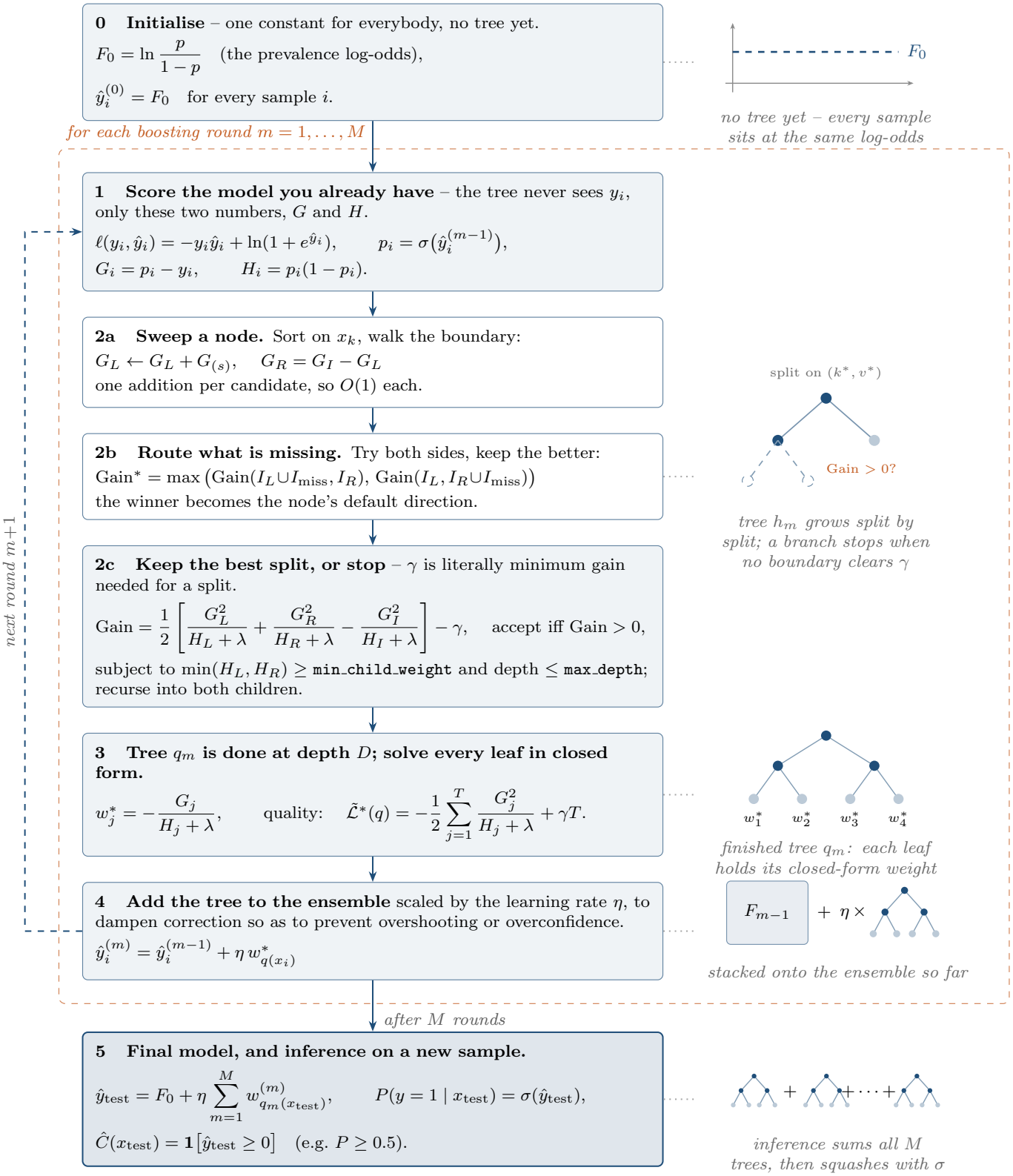



**Figure 5:** XGBoost end-to-end illustrated.

# 4 Why Does it Work so Well?

After all that exposition, we now turn our attention to the question oft-asked *"why is XGBoost so good?"*. Indeed, there's been multiple times with which I've been forwarded this question and other similar ones along the lines of *"why is X (e.g. neural net) not better than XGBoost [in tabular tasks]"*?

The answer is context-dependent as one would expect. Note that we're only discussing in the context of tabular dataset and task here, which is where XGBoost is most applied – there's little doubt that outside of this domain, Deep Neural models are the reigning champions. Now back to the question at hand – in order to focus the argument, let's narrow our discussion to *XGBoost (GBDT) vs Deep Neural Networks (DNN)* such as Transformer and Deep Tabular Network.

First, for the positives of DNN. In terms of expressivity, DNN are definitely much more powerful in this regard than XGBoost, so they can represent a far wider class of function mappings compactly [30,31]. Not only that, modern Deep Tabular Network like TabPFN [26], has shown remarkable abilities with in-context learning that's very sample efficient yet performant, being able to learn with just hundreds to thousands of samples [15, 26]. At its limit, even zero-shot transfers without any task-specific training data at all for certain domains [27]; of course, this excludes the pre-training of the transformer on a general corpus. This could be very useful indeed for tasks with very little available data or labels in cold-start settings, or for quick ad hoc tasks where one needs to make a prediction on a specific context without first training a model – hence, I'm very excited to see where this will go!

That being said, in a majority of real-world tasks, we do have a lot of available data – think of any products such as Stripe or Instagram, and at their scale, data abundance is definitely not a problem. Plus, DNN are very compute heavy, being slow to train and serve unlike XGBoost which is super snappy [13], sufficient to meet the high SLA / TPS requirements of real-time systems without needing a GPU; great news for those with a heavy compute backlog! On the other side of the train-serve equation, non-neural models like XGBoost are also endowed with very quick training speed, paired with the ability to ingest huge dataset ($>$ 100M rows) via lazy streaming [2], thereby making it much faster to iterate with unlike the lumbering DNN.

Now, other than the obvious and logistical (compute) reasons, XGBoost also admits certain very helpful properties in its learning process as well! The flipside problem with DNN on their expressivity is the issue of overfitting, which comes at no surprise to any practitioners – it's not easy to "tame" a DNN so it wouldn't just memorize the dataset trivially. XGBoost on the other hand, is capable of dealing with the issue of overfitting naturally – as we've seen previously, all GBDT like XGBoost does, is learn how to partition the feature space into axis-aligned "rectangles" to segment the samples cleanly.

It is precisely the "constraint" of only being able to learn additive axis-aligned rectangular partitions – no rotations or bending! [12, 28] – that naturally induces a structural regularization, thereby preventing it from overfitting easily, e.g. it cannot learn arbitrarily smooth feature segmentation boundaries. Added to that, the *margin maximizing* effect of boosting in the small-$\eta$ limit [29] further forces an additional constraint in a useful way, preventing degeneration into trivial margin-agnostic segmentation that's highly irregular or jagged. In short, the structure of the trees themselves with its margin maximizing tendency provides built-in, implicit regularization mechanics to XGBoost, on top of other explicit regularizing forces such as $\gamma$, $\lambda$ and co!

Upon some consideration, one could also notice certain congruency between the tabular

datasets and the structure of XGBoost we just covered in prior sections [12]. For example, the features manifold of tabular data is typically irregular (e.g. non-smooth with sharp thresholds) such as credit scores – which have categorical spans, where a certain score-band means 'poor' while being above certain score indicates an 'excellent' credit risk profile – or glucose level – above certain values, we classify as excessive; hyperglycemia or hypoglycemia – and so on. DNN are biased towards smooth, low frequency boundaries [12, 32], whereas GBDT like XGBoost's staircase like boundaries matches this inductive bias perfectly. Also, in tabular data, columns are oftentimes *"self-describing"* or individually-meaningful – `annual_income` and `age` are by themselves salient and are widely used "abstraction" – which the axis-aligned nature of GBDT naturally assumes for free without needing to manually "impose" any additional constraints. For example, take the income and age columns of a dataset, and replace them with their sum or differences. No information has been lost, it's just written along a rotated pair of axes, which may or may not be a useful joint feature for the task at hand; worse if its destructive! Tabular data arrives already with a natural basis imposed by whoever created the columns, so a learner that is invariant to rotation has to rediscover that orientation from scratch, mixing together features with very different statistical properties along the way [12, 28] which may not yield any useful signal. GBDT on the hand, naturally leans into these given tabular priors nicely.

Last but not – exhaustively – least, is the ability of GBDT to easily and naturally ignore uninformative features, of which there usually are in any real world datasets, by simply not splitting on those features, and be able to do so for free as part of its Gain computation's factor already, unlike the DNN which could degrade due to relative influences [12]; though, this is more of a problem of plain feedforward networks as newer Transformer-based models like FT-Transformer have accounted for with their per-feature tokenizer [33].

All these taken together, are reasons why Gradient Boosted Decision Trees like XGBoost normally rank so well in tabular or structured learning task and benchmarks [13, 14], and is so often preferred by production teams even to this day – it's just a tabular-shaped solution to a tabular world, and it's zippy too!

| | XGBoost (GBDT) | DNN (Transformer, Deep Tabular) |
|---|---|---|
| **Expressivity** | Additive axis-aligned rectangle partitions – no rotations or bending | Much more powerful; represents a far wider class of function mappings compactly |
| **Small data, cold start** | Needs task-specific training data | In-context learning that is very sample efficient yet performant; at its limit even zero-shot for certain domains |
| **Data abundance** | The regime it is built for (the usual case) | Advantage narrows once data is abundant – at Stripe or Instagram scale, data is definitely not a problem |
| **Compute and latency** | Super snappy; CPU alone is sufficient to meet the high SLA / TPS requirements of large scale systems | Very compute heavy and slow; takes specialized optimization and GPUs to catch up |
| **Training and iteration** | Much faster to train, and therefore iterate rapidly | Lumbering by comparison |
| **Overfitting** | The constraint itself induces a structural regularization, plus the margin maximizing tendency and $+\lambda$ – a built-in, implicit regularization | Not easy to "tame" so it wouldn't just memorize the dataset trivially |
| **Tabular inductive bias** | Matches irregular features with sharp thresholds (credit scores, glucose levels) perfectly | Biased towards smooth, low frequency segmentations |
| **Independent columns** | The axis-aligned nature assumes this for free, without needing to manually "impose" any additional constraints | No such assumption built in |
| **Uninformative columns** | Ignored by simply not splitting on those features, essentially for free – as part of its Gain computation already | Could degrade due to relative influence (more of a problem of plain MLP; newer Transformer-based models like FT-Transformer have accounted for it with their per-feature tokenizer) |

**Table 1:** Where each family wins, in the tabular setting.

## 5 Closing Words

Now dear *Readers*, we're at the end of our journey. That being said, I must point out a few areas of deficiencies and shortcuts I have taken in this exposition for clarity and brevity, lest I be stoned by the but-ackchyually-police!

First off, the enumeration of XGBoost's mechanic is incomplete and only cover core mechanics that I believe to be necessary and sufficient. For example, there's no mention of how the tree is grown to `max_depth` and then performs backward-pruning, the full treatment of the weighted quantile sketch and approximate split finding, how it handles categorical data, how it performs multicore training, how additional improvements have allowed it to stream and train lazily for datasets that won't fit in memory, how it computes feature importance, and many more. For those, I would direct the motivated reader to the original paper [1] and the latest official documentation [2].

Furthermore, the theoretical treatment is also rather lackluster as it's not intended to be a textbook. However, there's a great deal to be gleaned from and pondered on via a more rigorous treatment in depth. For example, how the learning procedure can be seen as an $\ell_1$ walk on the axis-aligned path in the function space [29], how margin theory can help us understand why it can learn so well [23], what are its convergence properties and levers that affect it most directly, which in turns help us build intuition to understand why a *turn-up-the-rounds-to-11-and-lower-the-learning-rate* is a decent strategy in general when tuning XGBoost.

Last and most importantly in my opinion, is the omission on how the model behaves in extreme settings such as dealing with a dataset with huge prevalence imbalance – which is very common in domains such as credit-card fraud, cancer diagnosis, and many others – and as a result this impacts calibration and uncertainty quantification, which in turns affects downstream dependencies such as the policy engine and so on. Other just as important considerations are on dealing with a drifting non-stationary environment, and when uncertainty estimation is unreliable or otherwise. Perhaps these can be addressed in a future write-up on operational considerations in practice.

That's the tour, and I hope that this has been helpful and not too dreadful of a read. For any inquiries, errors and corrections, or simply a conversation on the topic, feel free to reach out via my email and I shall make the best endeavor to reply swiftly. *Ta-ta!*

## References


[1] T. Chen and C. Guestrin, "XGBoost: A scalable tree boosting system," in *Proc. 22nd ACM SIGKDD Int. Conf. Knowledge Discovery and Data Mining (KDD)*, San Francisco, CA, USA, 2016, pp. 785–794.

[2] T. Chen *et al.*, "Using XGBoost External Memory Version," *XGBoost Documentation*, ver. 3.4.1, 2026. [Online]. Available: https://xgboost.readthedocs.io/en/stable/tutorials/external_memory.html

[3] C. Miller *et al.*, "The ASHRAE Great Energy Predictor III competition: Overview and results," *Science and Technology for the Built Environment*, vol. 26, no. 10, pp. 1427–1447, 2020.

[4] N. Erickson, L. Purucker, A. Tschalzev, D. Holzmüller, P. Mutalik Desai, D. Salinas, and F. Hutter, "TabArena: A living benchmark for machine learning on tabular data," *arXiv:2506.16791*, 2025.

[5] T. Chen and T. He, "Higgs boson discovery with boosted trees," in *Proc. NIPS 2014 Workshop on High-Energy Physics and Machine Learning*, PMLR, vol. 42, 2015, pp. 69–80.

[6] B. R. Gunnarsson, S. vanden Broucke, B. Baesens, M. Óskarsdóttir, and W. Lemahieu, "Deep learning for credit scoring: Do or don't?," *European J. Operational Research*, vol. 295, no. 1, pp. 292–305, 2021.

[7] J. Hancock and T. M. Khoshgoftaar, "Performance of CatBoost and XGBoost in Medicare fraud detection," in *Proc. 19th IEEE Int. Conf. Machine Learning and Applications (ICMLA)*, 2020, pp. 572–579.

[8] S. Lessmann, B. Baesens, H.-V. Seow, and L. C. Thomas, "Benchmarking state-of-the-art classification algorithms for credit scoring: An update of research," *European J. Operational Research*, vol. 247, no. 1, pp. 124–136, 2015.

[9] S. Moro, P. Cortez, and P. Rita, "A data-driven approach to predict the success of bank telemarketing," *Decision Support Systems*, vol. 62, pp. 22–31, 2014.

[10] S. M. Lundberg *et al.*, "Explainable machine-learning predictions for the prevention of hypoxaemia during surgery," *Nature Biomedical Engineering*, vol. 2, no. 10, pp. 749–760, 2018.

[11] S. Makridakis, E. Spiliotis, and V. Assimakopoulos, "M5 accuracy competition: Results, findings, and conclusions," *Int. J. Forecasting*, vol. 38, no. 4, pp. 1346–1364, 2022.

[12] L. Grinsztajn, E. Oyallon, and G. Varoquaux, "Why do tree-based models still outperform deep learning on typical tabular data?," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 35, 2022, pp. 507–520.

[13] R. Shwartz-Ziv and A. Armon, "Tabular data: Deep learning is not all you need," *Information Fusion*, vol. 81, pp. 84–90, 2022.

[14] D. McElfresh *et al.*, "When do neural nets outperform boosted trees on tabular data?," in *Advances in Neural Information Processing Systems (NeurIPS), Datasets and Benchmarks Track*, 2023.

[15] N. Hollmann *et al.*, "Accurate predictions on small data with a tabular foundation model," *Nature*, vol. 637, no. 8045, pp. 319–326, 2025.

[16] H.-J. Ye, S.-Y. Liu, and W.-L. Chao, "A closer look at TabPFN v2: Strength, limitation, and extension," *arXiv:2502.17361*, 2025.

[17] T. Hastie, R. Tibshirani, and J. Friedman, *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*, 2nd ed. New York, NY, USA: Springer, 2009.

[18] R. E. Schapire and Y. Freund, *Boosting: Foundations and Algorithms*. Cambridge, MA, USA: MIT Press, 2012.

[19] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone, *Classification and Regression Trees*. Belmont, CA, USA: Wadsworth, 1984.

[20] M. J. Kearns and L. G. Valiant, "Cryptographic limitations on learning Boolean formulae and finite automata," *J. ACM*, vol. 41, no. 1, pp. 67–95, 1994.

[21] R. E. Schapire, "The strength of weak learnability," *Machine Learning*, vol. 5, no. 2, pp. 197–227, 1990.

[22] Y. Freund and R. E. Schapire, "A decision-theoretic generalization of on-line learning and an application to boosting," *J. Computer and System Sciences*, vol. 55, no. 1, pp. 119–139, 1997.

[23] R. E. Schapire, Y. Freund, P. Bartlett, and W. S. Lee, "Boosting the margin: A new explanation for the effectiveness of voting methods," *Annals of Statistics*, vol. 26, no. 5, pp. 1651–1686, 1998.

[24] J. Friedman, T. Hastie, and R. Tibshirani, "Additive logistic regression: A statistical view of boosting," *Annals of Statistics*, vol. 28, no. 2, pp. 337–407, 2000.

[25] J. H. Friedman, "Greedy function approximation: A gradient boosting machine," *Annals of Statistics*, vol. 29, no. 5, pp. 1189–1232, 2001.

[26] N. Hollmann, S. Müller, K. Eggensperger, and F. Hutter, "TabPFN: A transformer that solves small tabular classification problems in a second," in *Proc. Int. Conf. Learning Representations (ICLR)*, 2023.

[27] S. Hegselmann, A. Buendia, H. Lang, M. Agrawal, X. Jiang, and D. Sontag, "TabLLM: Few-shot classification of tabular data with large language models," in *Proc. Int. Conf. Artificial Intelligence and Statistics (AISTATS)*, 2023.

[28] A. Y. Ng, "Feature selection, $L_1$ vs. $L_2$ regularization, and rotational invariance," in *Proc. 21st Int. Conf. Machine Learning (ICML)*, 2004.

[29] S. Rosset, J. Zhu, and T. Hastie, "Boosting as a regularized path to a maximum margin classifier," *J. Machine Learning Research*, vol. 5, pp. 941–973, 2004.

[30] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, no. 4, pp. 303–314, 1989.

[31] M. Telgarsky, "Benefits of depth in neural networks," in *Proc. 29th Conf. Learning Theory (COLT)*, PMLR, vol. 49, 2016, pp. 1517–1539.

[32] N. Rahaman, A. Baratin, D. Arpit, F. Draxler, M. Lin, F. Hamprecht, Y. Bengio, and A. Courville, "On the spectral bias of neural networks," in *Proc. 36th Int. Conf. Machine Learning (ICML)*, PMLR, vol. 97, 2019, pp. 5301–5310.

[33] Y. Gorishniy, I. Rubachev, V. Khrulkov, and A. Babenko, "Revisiting deep learning models for tabular data," in *Advances in Neural Information Processing Systems (NeurIPS)*, vol. 34, 2021, pp. 18932–18943.