🚀 Understanding XGBoost Regression

A Step-by-Step Theoretical Walkthrough

🔵 Introduction

This notebook explains how the **XGBoost Regression algorithm** works — step by step and in simple terms.
It shows what happens inside the model: how it uses gradients and Hessians to build trees, how it decides where to split, and how predictions get better with each round.

You'll see the key formulas, example values, and clear explanations that make the process easy to follow.

XGBoost (short for *Extreme Gradient Boosting*) is a machine learning algorithm that builds decision trees one after another.
Each new tree tries to correct the errors made by the previous ones, using gradient-based optimization to improve performance.
It is known for being **fast, accurate, and highly effective** in practice.

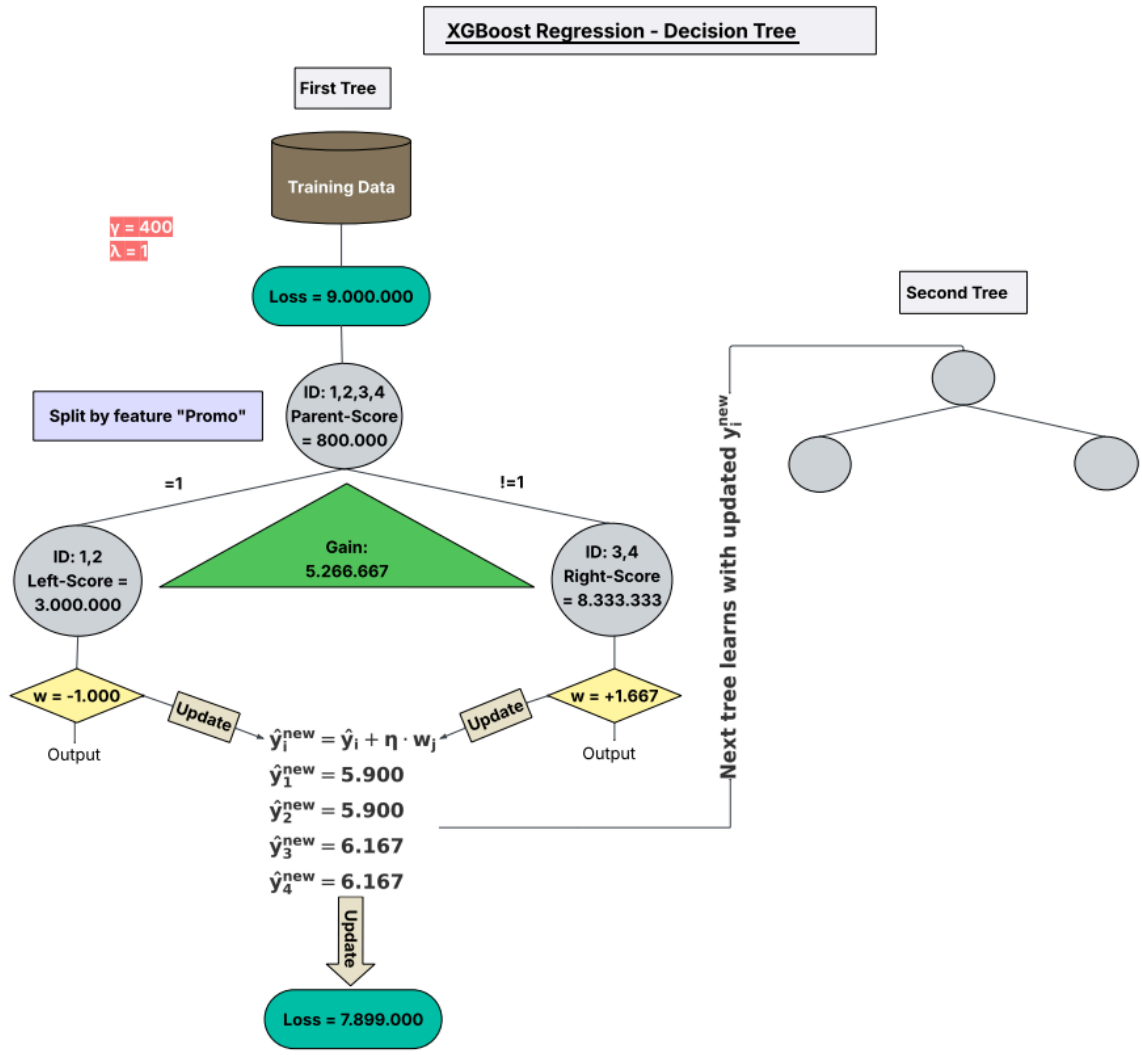
The goal of this notebook is not just to run code, but to understand what XGBoost does at each step.

The following illustration shows the algorithm of XGBoost, demonstrating how predictions $\hat{y}_i$ are updated by applying the weights w_j with the learning rate η, followed by training the next tree based on these updated predictions.

Where:

- $\hat{y}_i$ = prediction for observation i
- η = learning rate (step size)
- w_j = weight (output) of leaf j
- $\hat{y}_i^{new}$ = updated prediction after applying the weight

Perhaps it feels a bit complicated right now, but as we said, we will go through everything step by step in this notebook.



🔵 How does it work?

Here is a simplified overview of the 12 key steps we will go through:

1. **Loss Function**
   Define how the model measures error (e.g., Mean Squared Error for regression tasks).

2. **Gradient & Hessian**
   Calculate the 1st and 2nd derivatives of the loss function — they guide the tree building process.

3. **Sample Data (Rossmann)**
   Use a small sample from the dataset to demonstrate how XGBoost works.

4. **Taylor Approximation**
   Approximate the loss function using a second-order formula (for better optimization).

5. **Gain Calculation & First Split**
   Choose the best feature and threshold to split the data and reduce the loss.

6. **Second Split (Left Subtree)**
   Try to split the left branch further, if it improves the model.

7. **Second Split (Right Subtree)**
   Same for the right side — check if further splitting is useful.

8. **Compute Leaf Outputs**
   Each leaf gets a score (prediction value) that minimizes the local loss.

9. **Update Predictions**
   Adjust the original prediction using the tree output and the learning rate.

10. **New Loss After First Tree**
    Recalculate total loss and compare it to the previous step.

11. **Pass to the Next Tree**
    The model builds a new tree to fix the remaining error.

12. **Load Test Data**
    Finally, we feed the real cleaned test data into the model to begin training.

In the next section, we will go through **each of these steps one by one** and explain how they work — with formulas and examples

---

🔵 1. Loss Function – What is it?

The **loss function** measures the error between:

- y_i: the true value (e.g., actual sales)
- $\hat{y}_i$: the prediction made by the model

For regression tasks like "predicting sales", we use the **Mean Squared Error (MSE)** as the loss function:

$$\mathcal{L} = \sum_{i=1}^{n} \frac{1}{2}(y_i - \hat{y}_i)^2$$

Why this formula?

- The square term $(y_i - \hat{y}_i)^2$ gives **more weight to larger errors**.
- The constant $\frac{1}{2}$ is added to **simplify the derivative**:

Without $\frac{1}{2}$: the derivative becomes $-2(y_i - \hat{y}_i)$
With $\frac{1}{2}$: the derivative is just $-(y_i - \hat{y}_i)$

→ This makes the gradient cleaner and easier to work with during optimization.

Why is there no $\frac{1}{n}$?

In classical MSE, we often write:

$$\text{MSE} = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

But XGBoost **removes the** $\frac{1}{n}$ for two main reasons:

1. It doesn't care about the **average loss per row**
   → It only wants to reduce the **total loss**.

2. The algorithm works **tree by tree**, checking:
   "Did this new tree reduce the total loss?"

Since this comparison is relative, multiplying everything by $\frac{1}{n}$ would not change the result.
So it's left out for simplicity and speed.

Inside the tree:

- The **loss function** is the main objective.
- Every new tree is trained to **reduce this loss further**.
- XGBoost uses this function together with **gradients and Hessians** in a Taylor approximation — which we'll explore in the next step.

Example:
Suppose the old loss is 9.000.000 and the new loss is 7.899.000.
Even if we divide both by n, the comparison stays the same:

$$\frac{1}{n} \cdot 9.000.000 \; > \; \frac{1}{n} \cdot 7.899.000$$

So XGBoost skips the $\frac{1}{n}$ — it's not needed for deciding which tree is better.

🔵 2. Gradient & Hessian – How does the model learn?

To reduce the loss, XGBoost uses two important values from calculus:

Gradient g_i = First Derivative (slope)

The **gradient** tells us the direction in which the prediction needs to move to reduce the error.

It is the first derivative of the loss function:

$$g_i = \frac{\partial \mathcal{L}}{\partial \hat{y}_i} = \hat{y}_i - y_i$$

- If g_i is **positive** → the model **overestimated** → prediction too high
- If g_i is **negative** → the model **underestimated** → prediction too low

🧠 Gradient = **Slope** → Shows **which direction** to go.

Hessian h_i = Second Derivative (curvature)

The **Hessian** shows how fast the gradient is changing — it reflects the **curvature** of the loss function.

In the beginning, XGBoost predicts the **same value for all instances** — usually the **mean of all true target values**.

It is the second derivative:

$$h_i = \frac{\partial^2 \mathcal{L}}{\partial \hat{y}_i^2} = 1$$

For Mean Squared Error (MSE), the Hessian is always **1** (a constant).

Hessian = **Curvature** → Shows **how fast the slope is changing**.

These two values are used in the **Taylor approximation** (next step) to calculate how much the tree should change the prediction.

🔵 3. Sample Data

Let's now look at a **hypothetical example** (not real data) to understand how XGBoost uses gradients, Hessians, and loss values.

ID	Promo	Open	True Value (y_i)	Prediction ($\hat{y}_i$)	Gradient ($g_i = \hat{y}_i - y_i$)	Hessian (h_i)	Loss $\left(\frac{1}{2}(y_i - \hat{y}_i)^2\right)$
1	1	1	5.000	6.000	+1.000	1	500.000
2	1	0	4.000	6.000	+2.000	1	2.000.000
3	0	1	8.000	6.000	−2.000	1	2.000.000

ID	Promo	Open	True Value (y_i)	Prediction ($\hat{y}_i$)	Gradient ($g_i = \hat{y}_i - y_i$)	Hessian (h_i)	Loss $\left(\frac{1}{2}(y_i - \hat{y}_i)^2\right)$
4	0	0	9.000	6.000	−3.000	1	4.500.000

Initial total loss (before the first tree):

$$\mathcal{L}_{\text{old}} = 500.000 + 2.000.000 + 2.000.000 + 4.500.000 = 9.000.000$$

This is the baseline loss before training begins.
The next tree will try to reduce this number by fitting to the gradients.

🔵 4. Taylor Approximation (Simplified Objective Function)

XGBoost doesn't minimize the loss function directly.
Instead, it uses a **second-order Taylor approximation** to represent the loss function more efficiently:

$$\mathcal{L}_{\text{approx}} = \sum_i \left(g_i \cdot w + \frac{1}{2} h_i \cdot w^2 \right) + \text{const}$$

Why use Taylor Approximation?

- It turns the original loss into a **simple quadratic formula** that works well for small updates.
- It uses the **1st derivative (gradient)** g_i and **2nd derivative (Hessian)** h_i of the loss function.
- For **each leaf** in the tree, we can compute the best output value w_j that **locally minimizes the loss** for the data points in that leaf.
- This allows XGBoost to **reduce the loss step by step** — each new leaf makes a contribution to the total improvement.

Context:

Each leaf in the tree gives a constant prediction.
In the approximation, $f(x_i)$ is replaced by a constant w_j — the leaf output.

> **Goal:** Find the optimal value w_j for each leaf to minimize the local loss.

🔵 5. First Split Comparison: Promo vs Open

To determine the best first split, we compare the feature **Promo** and **Open** side-by-side.

General Gain Formula

$$\text{Gain} = \frac{1}{2}\left(\underbrace{\frac{G_L^2}{H_L+\lambda}}_{\text{Score}_{\text{left}}} + \underbrace{\frac{G_R^2}{H_R+\lambda}}_{\text{Score}_{\text{right}}} - \underbrace{\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}}_{\text{Score}_{\text{parent}}}\right)$$

- G_L, H_L: Gradient and Hessian sum for the **left** child
- G_R, H_R: Gradient and Hessian sum for the **right** child
- λ: regularization parameter (here: $\lambda = 1$)
- γ: minimum gain required to perform a split (here: $\gamma = 400$)

> The **gain** measures how much a split reduces the loss by comparing the combined scores of the child nodes with the original score of the parent node.

Split by Promo = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	1, 2	Yes	$+3.000$	2
Right	3, 4	No	-5.000	2

Gain Calculation (Promo):

$$\text{Gain}_{\text{Promo}} = \frac{1}{2}\left(\frac{3.000^2}{2+1} + \frac{(-5.000)^2}{2+1} - \frac{(-2.000)^2}{4+1}\right) = \boxed{5.266.667}$$

Split by Open = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	1, 3	Yes	-2.000	2
Right	2, 4	No	$+3.000$	2

Gain Calculation (Open):

$$\text{Gain}_{\text{Open}} = \frac{1}{2}\left(\frac{(-2000)^2}{2+1} + \frac{3000^2}{2+1} - \frac{1000^2}{4+1}\right) = \boxed{2{,}066{,}667}$$

Final Decision

Feature	Gain	$\gamma = 400$	Split Accepted?
Promo	5.266.667	✔	**Yes**
Open	2.066.667	✔	Yes (but lower Gain)

Promo is chosen as the first split, since it yields the highest gain.

Why Promo was chosen

- The feature **Promo** gives a high Gain and therefore becomes the **first split** in the tree.
- The feature **Open** is **not checked here**, just to keep things simple.
- Promo is the **best candidate** for the first split based on the Gain value.

What are γ and λ?

- λ **(lambda)** is the **regularization** term for leaf weights.
  It makes the model less sensitive to noise by **penalizing large leaf outputs**.

- γ **(gamma)** is the **minimum Gain** required to make a split.
  If a split results in a Gain **less than** γ, it is **rejected** to keep the tree small and simple.

  > Both parameters help prevent **overfitting** and improve generalization.

🔵 6. Second Split in the Left Subtree: Open vs Promo

To decide whether the **left node** (instances 1 and 2) should be split further, we compare the features **Open** and **Promo**.

Split by Open = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	1	Yes	$+1.000$	1
Right	2	No	$+2.000$	1

Gain Calculation (Open):

$$\text{Gain}_{\text{Open}} = \frac{1}{2}\left(\frac{1000^2}{1+1} + \frac{2000^2}{1+1} - \frac{3000^2}{2+1}\right) = \frac{1}{2}(500{,}000 + 2{,}000{,}000 - 3{,}000{,}000) = \boxed{-}$$

Split by Promo = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	1, 2	Yes	$+3.000$	2
Right	–	No (no separation possible)	–	–

Gain Calculation (Promo):

Not possible: both instances have **Promo = 1**, so no separation is possible.

→ Split is invalid

Feature	Gain	$\gamma = 400$	Split Accepted?
Open	-250.000	✗	No (Gain too low)
Promo	–	–	No (no separation possible)

Neither split is accepted. The left node remains unchanged.

🔵 7. Second Split in the Right Subtree: Open vs Promo

We now evaluate whether the **right node** (instances 3 and 4) should be split further.
We compare the features **Open** and **Promo**.

Split by Open = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	3	Yes	-2.000	1
Right	4	No	-3.000	1

Gain Calculation (Open):

$$\text{Gain}_{\text{Open}} = \frac{1}{2}\left(\frac{(-2000)^2}{1+1} + \frac{(-3000)^2}{1+1} - \frac{(-5000)^2}{2+1}\right) = \frac{1}{2}(2{,}000{,}000 + 4{,}500{,}000 - 8{,}$$

Split by Promo = 1

Side	Instance IDs	Split = 1?	$G_j = \sum g_i$	$H_j = \sum h_i$
Left	3, 4	Yes	-5.000	2
Right	–	No (no separation possible)	–	–

Gain Calculation (Promo):

Not possible: both instances have **Promo = 0**, so no separation is possible.

→ Split is invalid

Feature	Gain	$\gamma = 400$	Split Accepted?
Open	-916.667	✗	No (Gain too low)
Promo	–	–	No (no separation possible)

Neither split is accepted. The right node remains unchanged.

🔵 8. Calculate the Leaf Output (per leaf)

Derivation: Taylor Approximation

XGBoost minimizes the total loss by approximating it with a second-order (quadratic) function.
For a given leaf j, the approximated loss is:

$$\text{Loss}_j \approx \sum_{i \in j}\left(g_i w_j + \frac{1}{2} h_i w_j^2\right)$$

- g_i: first derivative (gradient) of the loss for instance i
- h_i: second derivative (Hessian)
- w_j: prediction value for the leaf j

Goal: Minimize the loss by choosing the optimal w_j

Take the derivative of the loss w.r.t. w_j and set it to 0:

$$\frac{\partial \text{Loss}_j}{\partial w_j} = \sum_{i \in j} \left(g_i + h_i w_j \right) = 0$$

Solving for w_j:

$$w_j = -\frac{\sum g_i}{\sum h_i}$$

To stabilize the solution, we add **regularization** λ to the denominator:

$$w_j = -\frac{\sum g_i}{\sum h_i + \lambda}$$

Final Formula for Leaf Output:

$$w_j = -\frac{G_j}{H_j + \lambda}$$

Where:

- $G_j = \sum g_i$: sum of gradients in leaf j
- $H_j = \sum h_i$: sum of Hessians in leaf j

Leaf Output Calculation (using $\lambda = 1$):

Leaf	Instance IDs	G_j	H_j	Leaf Output $w_j = -\frac{G_j}{H_j+1}$
Left	1, 2	$+3.000$	2	$-\frac{3.000}{2+1} = -1.000$
Right	3, 4	-5.000	2	$-\frac{-5.000}{2+1} = +1.667$

🔵 9. Prediction Update through the Tree

The prediction for each instance is updated based on the **leaf output** w_j and the **learning rate** η.
The updated prediction is calculated as:

$$\hat{y}_i^{\text{new}} = \hat{y}_i + \eta \cdot w_j$$

ID	Old Prediction $\hat{y}_i$	Tree Output w_j	Learning Rate $\eta = 0{,}1$	New Prediction $\hat{y}_i^{\text{new}}$
1	6.000	-1.000	0.1	$6.000 - 0.100 = 5.900$
2	6.000	-1.000	0.1	$6.000 - 0.100 = 5.900$
3	6.000	$+1.667$	0.1	$6.000 + 0.167 = 6.167$
4	6.000	$+1.667$	0.1	$6.000 + 0.167 = 6.167$

Notes

- The **leaf outputs** w_j were previously computed based on the gradient and Hessian sums. The **learning rate** $\eta = 0.1$ scales the influence of the new tree.
  This prevents overfitting and ensures gradual learning (typical values: 0.01–0.3).
- Predictions are **updated step by step**: each tree makes a small change to the previous prediction.

This is how **boosting** works step by step: every tree adds a small correction to the overall prediction.

🔵 10. New Loss After the Update

Instance (ID)	Feature: Promo	Feature: Open	Target y_i	New Prediction $\hat{y}_i^{\text{new}}$	Gradient $g_i = \hat{y}_i - y_i$	Hessian h_i	Loss $\frac{1}{2}(y_i - \hat{y}_i)^2$
1	1	1	5.000	5.900	+0.900	1	0.405.000
2	1	0	4.000	5.900	+1.900	1	1.805.000
3	0	1	8.000	6.167	–1.833	1	1.679.000
4	0	0	9.000	6.167	–2.833	1	4.010.000

New total loss:

$$\mathcal{L}_{\text{new}} = 7.899.000$$

Loss reduction after the first tree

The total loss has decreased due to the first tree's update.

We calculate the reduction:

$$\Delta \mathcal{L} = \mathcal{L}_{\text{old}} - \mathcal{L}_{\text{new}} = 9.000.000 - 7.899.000 = \boxed{1.101.000}$$

Result: The first tree reduced the total loss by **1.101.000**.

🔵 11. Passing to the Next Tree

After the first update, we now have an improved prediction (\hat{y}_i^{\text{new}}).
This prediction becomes the **starting point for the next tree**.

XGBoost uses **sequential learning**:
Each new tree attempts to reduce the remaining prediction error further.

To build **efficient and well-generalizing trees**, XGBoost includes several **built-in stopping mechanisms** (parameter).
These criteria decide whether a node should be split or the growth should stop.

Important Regularization Parameter: `gamma`

- `gamma` is a **threshold for the Gain** – i.e., how useful a new split is.
- A split is **only performed if**:

$$\text{Gain} \geq \gamma$$

- This ensures that **only worthwhile splits** are accepted.

Rule of thumb:
A higher `gamma` leads to **simpler trees** that generalize better (→ less overfitting).

Additional Stopping Criteria

1. **Maximum number of trees**
   → Controlled via the parameter `n_estimators`.
2. **Early stopping**
   → Training stops if the validation error no longer improves after a certain number of rounds.
   → Controlled via `early_stopping_rounds`.
3. **Maximum tree depth reached**
   → Controlled via `max_depth`.
4. **Too few observations in a leaf**
   → Controlled via `min_child_weight`.

→ If the "sum of instance weights" in a leaf is too small, no further split is made.

5. **Minimum gain not achieved**
   → Here too, `gamma` plays a role.
   → Some libraries (e.g., LightGBM) use `min_split_gain` additionally.

6. **Learning rate too small**
   → If `eta` (learning rate) is set too low, updates become tiny → **very slow or no progress** despite many trees.

🔵 12. Load Test Data

Now we use the **real test data** to make **predictions**. This final step is about **using the trained model** to predict our final target:
Sales.

What happens during prediction?

- The cleaned test data is passed through all the trained **trees**.
- Each tree sends every test instance through its **splits** until it reaches a **leaf**.
- The **leaf output** is the prediction of that tree for that instance.

How is the final prediction made?

For each test instance:

1. The model collects the **leaf output** from **every tree**.
2. These outputs are **summed up**.
3. The result is the **final prediction** for that instance.

Mathematically:

$$\hat{y}_i = \sum_{m=1}^{M} \eta \cdot w_j^{(m)}$$

Where:

- $\hat{y}_i$ = predicted sales for instance i
- M = number of trees
- η = learning rate
- $w_j^{(m)}$ = leaf output from tree m for that instance

So, each test instance is passed through all the trees, and the predictions from the leaf nodes are **added together** to get the final result.

Example

Let's say a test instance passes through 3 trees and lands in the following leaf nodes:

Tree	Leaf Output w_j	Learning Rate $\eta = 0.1$	Contribution
1	$+12.000$	0.1	$+1.200$
2	-2.500	0.1	-0.250
3	$+12.520$	0.1	$+1.252$

Then the final prediction is:

$$\hat{y}_i = 1.200 - 0.250 + 1.252 = 2.202$$

This is the predicted **sales value** for the given test instance, based on all tree contributions.

Note:

This is the **same formula** as in step 9 (prediction update):

$$\hat{y}_i^{\text{new}} = \hat{y}_i + \eta \cdot w_j$$

But during prediction, we start from **zero** and just **sum all corrections** from each tree:

$$\hat{y}_i = \sum_{m=1}^{M} \eta \cdot w_j^{(m)}$$