GRADIENT BOOSTING FOR REGRESSION

Boosting

Boosting is an ensemble learning technique in machine learning used to create a strong model by combining multiple weak models. A weak model is one that performs only slightly better than random guessing (like a shallow decision tree). The main idea of boosting is to improve overall performance by sequentially training each new model to correct the mistakes made by the previous ones.

Quick Comparison with Bagging:

Let's first understand how Bagging works so we can see how Boosting is different.

Bagging (Bootstrap Aggregating):

- In bagging, we take multiple base models (e.g., d_1, d_2, d_3) and apply bootstrapping to our dataset.
- Bootstrapping means creating multiple random samples with replacement from the original dataset.
- Each model is trained on a different bootstrap sample.
- When we get a new input (query point), we ask all models to make a prediction. Their outputs are then aggregated using methods like majority voting (for classification) or averaging (for regression).
- Bagging is used when we have models that are low bias but high variance, like deep decision trees. These models tend to overfit, but bagging reduces their variance by averaging.
- The philosophy behind bagging is the "wisdom of the crowd"—many diverse opinions averaged together tend to be more accurate.

Boosting – How It's Different

Now let's break down how Boosting differs from Bagging in three key ways:

1. Training Order:
   - Bagging trains all models in parallel, independently of each other.
   - Boosting trains models sequentially—each model is trained after the previous one.
2. Data Usage:
   - In bagging, each model gets a different random subset of the data.
   - In boosting, all models usually use the same dataset, but the way they treat each data point is different.
     - We assign weights or importance to each data point.

    - If a data point was misclassified by the previous model, its weight is increased so that the next model focuses more on it.

3. Type of Base Models:

- Bagging prefers complex models that tend to overfit (high variance), like large decision trees.
- Boosting uses simple models with high bias but low variance, like shallow decision trees (called decision stumps).
    - These simple models may perform poorly on training data but can generalize well when properly combined.

Core Idea of Boosting

Here's how boosting works step by step:

1. Start by assigning equal weights to all training examples.
2. Train the first model (m_1) on the dataset.
    - m_1 will make some mistakes — it may classify some data points incorrectly.
3. Increase the weights (importance) of the misclassified points.
    - This makes the next model (m_2) focus more on the hard examples that m_1 got wrong.
4. Train the second model (m_2) on the same dataset, but now it focuses more on the difficult points.
    - m_2 tries to correct the errors of m_1.
5. Again, update weights: increase them for the points that m_2 got wrong.
6. Train a third model (m_3) to correct the mistakes of m_2.
7. This process is repeated for several rounds.
8. In the end, we combine the predictions of all models using a weighted vote (for classification) or weighted sum (for regression).

What Happens to Bias and Variance ?

- Initially, we use high bias models (like small trees), which may underfit.
- As boosting goes on, each model reduces the error of the previous one, so the bias gradually decreases.
- Because each model is simple and we're not using random sampling like bagging, the variance stays low.
- Eventually, boosting creates a model that has low bias and low variance, which leads to high accuracy.

Summary (Sticky Points):

- Boosting = sequential correction of errors.
- Same dataset used each round, but with adjusted weights.
- Focuses on bias reduction.
- Uses high bias, low variance models.
- Final model is a weighted combination of all weak models.
- Converts weak learners into a strong learner.

What is Gradient Boosting ?

Gradient Boosting is an ensemble learning technique used in machine learning that builds a strong predictive model by combining several weak learners (typically decision trees). It works by adding models one by one, where each new model is trained to correct the mistakes of the combined previous models.

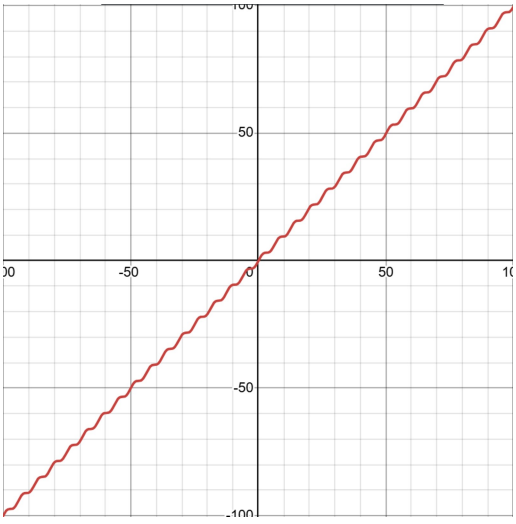
- Works for regression, classification, and even ranking tasks.
- Very effective when dealing with complex, non-linear datasets.
- Performs well with minimal hyperparameter tuning.
- Available in major libraries like scikit-learn, XGBoost, LightGBM, CatBoost.

Concept of Additive Modeling

Boosting uses the idea of additive modeling, where we build the final function by adding many small simple models step by step.

Real-Life Analogy:

Imagine trying to guess the equation of a complex graph (like the one shown below):



This complex graph might look hard at first. But suppose you realize it's a combination of two simple functions:

- y = x
- y = sin(x)

So, the complex curve could be:

$$y = x + \sin(x)$$

That's additive modeling: build complex functions by adding simpler ones.
Boosting follows the same approach — instead of solving everything in one shot, it solves small parts and adds them together.

How Gradient Boosting Works (Step-by-Step)

Let's say we have input x and output y , and we want to find the function:

$$y = f(x)$$

Step 1: Initial Prediction

Start with a **simple guess**. Usually, this is just the **mean** of the target values (for regression):

$$f_0(x) = \text{average of all } y$$

Step 2: Calculate Errors

Find the **residuals** (how wrong the prediction is):

$$\text{Error} = y - f_0(x)$$

Step 3: Train First Weak Model

Train a **small decision tree** on this error. Let it predict the error:

$$h_1(x) \approx y - f_0(x)$$

Step 4: Update Prediction

Now improve the model by adding this small correction:

$$f_1(x) = f_0(x) + h_1(x)$$

Step 5: Repeat

- Find new errors: $y - f_1(x)$
- Train next model $h_2(x)$
- Update: $f_2(x) = f_1(x) + h_2(x)$

Keep repeating this process:

$$f(x) = f_0(x) + h_1(x) + h_2(x) + h_3(x) + \cdots + h_n(x)$$

At the end, all the weak models add up to become a strong final model .

What Does "Gradient" Mean Here?

The word "Gradient" comes from gradient descent — a mathematical method used to reduce error step by step.
In gradient boosting:

- We compute the gradient of the loss function (i.e., direction of error).
- Then train the new weak model to move in the direction that reduces error.

- It’s like going downhill slowly to find the best path.

So, gradient boosting = Additive Modeling + Gradient Descent.

Final Sticky Notes

- Gradient Boosting builds a model step-by-step.
- Uses additive modeling: final model = sum of small models.
- Each model learns from mistakes of the previous one .
- Uses gradient descent to reduce error.
- Turns weak learners → strong model.
- Works well even with minimal tuning.
- Supports regression, classification, ranking .

HOW IT WORKS ?

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

1. Initialize $f_0(x) = \arg\min_\gamma \sum_{i=1}^N L(y_i, \gamma)$.
2. For $m = 1$ to M:
   (a) For $i = 1, 2, \ldots, N$ compute

$$r_{im} = -\left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}.$$

   (b) Fit a regression tree to the targets r_{im} giving terminal regions R_{jm}, $j = 1, 2, \ldots, J_m$.
   (c) For $j = 1, 2, \ldots, J_m$ compute

$$\gamma_{jm} = \arg\min_\gamma \sum_{x_i \in R_{jm}} L\left(y_i, f_{m-1}(x_i) + \gamma\right).$$

   (d) Update $f_m(x) = f_{m-1}(x) + \sum_{j=1}^{J_m} \gamma_{jm} I(x \in R_{jm})$.
3. Output $\hat{f}(x) = f_M(x)$.

Gradient Boosting Algorithm: Conceptual Overview

Gradient Boosting is a machine learning technique used for regression and classification tasks. It builds a strong predictive model by combining multiple weak learners, typically decision trees, in a sequential manner.

The main goal of gradient boosting is to find a mathematical function that maps the input features (x) to the output target (y) — that is, to model the relationship y = f(x).

Gradient Boosting breaks down this complex function f(x) into a sum of simpler functions:

$$f(x) = f_0(x) + f_1(x) + f_2(x) + f_3(x) + \cdots + f_n(x)$$

This is known as additive modeling.

Step-by-Step Algorithm Overview:

- **Step 1:**
  Start by estimating an initial function, denoted as $f_0(x)$. This is usually a constant value (like the mean of target values) and serves as the starting point for the model.
- **Step 2:**
  Iteratively add new functions $f_1(x), f_2(x)$, …. , $f_n(x)$, where each function is typically a shallow decision tree. These trees are trained to minimize the error (loss) of the current model.
- **Step 3:**
  After all iterations, we combine the initial function and all the learned decision trees:
  $$f(x) = f_0(x) + \sum_{i=1}^{n} f_i(x)$$
  This final function is the complete predictive model.

Mathematical Understanding of Gradient Boosting

Let's understand the working of Gradient Boosting in more detail:

- Training Data:
  We start with a dataset of n training examples:
  $$\{(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)\}$$
- Objective:
  We aim to find a function f(x) such that the predicted output $\hat{y} = f(x)$ closely matches the true output y . To do this, we minimize a loss function L(y,$\hat{y}$) .
- Loss Function:
  The loss function L (y , f(x)) should be differentiable. A common choice for regression is the squared loss:
  $$L(y, \hat{y}) = \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$
  To simplify the mathematics (especially derivatives), we often multiply this by a constant 1/2 , which does not affect the optimization:
  $$L(y, \hat{y}) = \frac{1}{2} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$
  Here:
  - y is the actual output.

- $\hat{y} = f(x)$ is the model's prediction.

This loss is differentiable, which is essential because gradient boosting uses gradient descent on the loss to iteratively improve the model .

Now we will move to step 1 .

If we want to find out the first function $f_0(x)$, we need to solve the equation written in step one for the variable called γ. It will then become our $f_0(x)$.
It was found out by:

$$f_0(x) = \arg\min_{\gamma} \sum_{i=1}^{n} L(y_i, \gamma)$$

We have to differentiate with respect to γ and set the result to zero to find the minimum:

$$\frac{\partial L}{\partial \gamma} = 0$$

We will find this:

Using squared loss, which is defined as:

$$L(y_i, \gamma) = \frac{1}{2}(y_i - \gamma)^2$$

We take the derivative:

$$\frac{\partial f_0(x)}{\partial \gamma} = \frac{\partial}{\partial \gamma} \sum_{i=1}^{n} \frac{1}{2}(y_i - \gamma)^2$$

We can pull the constant 1/2 out and differentiate:

$$= \sum_{i=1}^{n} \frac{1}{2} \cdot 2(y_i - \gamma)(-1) = -\sum_{i=1}^{n} (y_i - \gamma)$$

Set derivative to zero:

$$-\sum_{i=1}^{n} (y_i - \gamma) = 0 \Rightarrow \sum_{i=1}^{n} (y_i - \gamma) = 0$$

Since we have three rows in our data:

R&D	Ops	Marketing	Profit
165	137	472	192
101	92	250	144
29	127	201	91

We expand the equation using these three output (Profit) values:

$$\sum_{i=1}^{3}(y_i - \gamma) = 0 \Rightarrow (y_1 - \gamma) + (y_2 - \gamma) + (y_3 - \gamma) = 0$$

Substitute values:
(192 − γ) + (144 − γ) + (91 − γ) = 0 ⇒ 192 + 144 + 91 − 3γ = 0 ⇒ 3γ = 192 + 144 + 91 = 427 ⇒ γ = 427/3 = 142.3

Final Explanation:

This value of γ is nothing but the mean of the output feature (Profit) of our dataset. So basically, γ is the mean of y— the mean of our training data output column.
And this is our $f_0(x)$ and γ . This result naturally comes because we used the squared loss function, which always leads to the mean as the optimal constant predictor in step 1.
So we solved the first part of the equation using proper derivative and data-based reasoning.

Step 2

We are trying to find the full prediction function:
$f(x) = f_0(x) + f_1(x) + f_2(x) + \cdots + f_m(x)$
We have already found the first term, f_0(x) , which is the mean of the output column of our training data. However, the remaining terms (i.e., $f_1(x), f_2(x)$, …) are decision trees, and they are more complex.

Repeating Step 2 Multiple Times

Step 2 must be repeated for each decision tree we want to add to the model. As per Step 2(a), for each training instance i = 1,2,...,N , we must calculate a value called r_{im}, where:

- m is the index of the current decision tree (e.g., m = 1 for the first tree),
- i is the index of the training data row,
- and r_{im} is the pseudo residual, which is the gradient of the loss function with respect to the prediction.

So suppose we are now at m = 1 (the first decision tree). Our base model $f_0(x)$ was just the average, and now we are moving to fit the residuals using decision tree m_1.

We Need to Calculate:

$$r_{im} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_{m-1}}$$

Since m = 1 , this becomes:

$$r_{i1} = \left[\frac{\partial L(y_i, f(x_i))}{\partial f(x_i)}\right]_{f=f_0} = \left[\frac{\partial L(y_i, f_0(x_i))}{\partial f_0(x_i)}\right]$$

Loss Function

We use the squared loss, defined as:

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

In our case, we replace $\widehat{\boldsymbol{y}}_{\boldsymbol{i}}$ with f_0(x_i) :

$$L(y, f_0(x)) = \frac{1}{2}\sum_{i=1}^{n}(y_i - f_0(x_i))^2$$

Calculating Gradient:

To get the pseudo residual r_{i1} :

$$r_{i1} = \frac{1}{2} \cdot \frac{\partial}{\partial f_0(x_i)} \sum_{i=1}^{n}(y_i - f_0(x_i))^2 = -(y_i - f_0(x_i)) = f_0(x_i) - y_i$$

So each pseudo residual is the difference between the model prediction and the actual value.

Using Our Data:

From the table:

R&D	Ops	Marketing	Profit (y)	f_0(x)
165	137	472	192	142.33
101	92	250	144	142.33
29	127	201	91	142.33

Let's calculate the residuals:

$r_{11} = f_0(x_1) - y_1 = 142.33 - 192 = -49.67$
$r_{21} = f_0(x_2) - y_2 = 142.33 - 144 = -1.67$
$r_{31} = f_0(x_3) - y_3 = 142.33 - 91 = 51.33$

Summary of Step 2

We subtracted the prediction from the actual target value. In gradient boosting, these differences are called pseudo residuals, because they are gradients of the loss function.
These residuals become the target values for our next model (the first decision tree), which learns how to correct the mistakes of the previous model $f_0(x)$.

Step 2(b) – Building the Second Model m_1

Now that we have completed model 1 (i.e., $f_0(x)$), which simply predicted the mean of the target column, we move on to build the second model, m_1.
This second model is a decision tree regressor, specifically trained to fix the errors made by the first model.

What's Different in m_1 ?

- The input features (R&D, Ops, Marketing, etc.) remain the same.
- The output/target column is now not the Profit column, but the residuals (also known as pseudo residuals) computed from $f_0(x)$.

So in essence, we are training the second model on the mistakes made by the first model.
This is the core idea of gradient boosting — each new model is trained to correct the errors of the ensemble so far.

Summary

- Model 0: $f_0(x)$ is the mean prediction.
- Residuals: $r_{i1} = f_0(x_i) - y_i$
- Model 1: m_1 is a regression tree trained on input features and residuals from model 0.
- Goal of m_1 : Learn a function that approximates the residuals, so we can update the model as:

$$f_1(x) = f_0(x) + \alpha \cdot m_1(x)$$

(where α is a learning rate, which can be added in next steps.)

Output of This Step

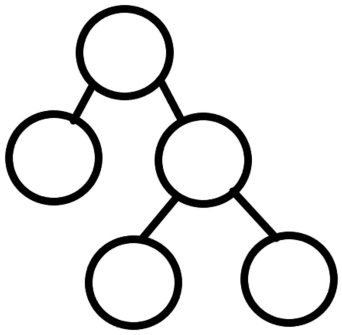
After training the regression tree on the input features and residuals, we obtain the second model $m_1(x)$, typically represented as a decision tree.
This completes Step 2(b) — building the second model using the errors of the first.

Terminal Region in Decision Trees

1. Introduction to Decision Trees

Decision trees are a type of supervised learning algorithm primarily used for classification and regression tasks. They work by recursively partitioning the data based on feature values, forming a tree-like structure.
A typical decision tree consists of:

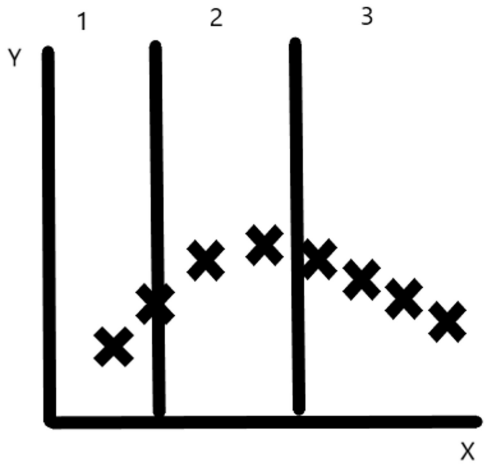
- Nodes: These represent decision points based on a particular feature.
- Leaf Nodes (or Terminal Nodes): These are the final nodes of the tree, representing the output or prediction. Data reaching a leaf node is assigned to that node's outcome.



2. Understanding Terminal Regions in Regression Trees

In the context of regression trees, the concept of a "terminal region" is crucial. When a decision tree is used for regression, its primary goal is to predict a continuous output variable (Y) based on input features (X). The tree achieves this by dividing the input space (the range of X values) into distinct, non-overlapping regions.
Each of these final, undivided regions corresponds directly to a leaf node of the decision tree. These are the "terminal regions." Within each terminal region, the regression tree predicts a constant value (typically the mean or median of the output variable for all training data points falling into that region).
Consider our regression data:

- The tree will apply a series of splits (vertical lines in a 1D feature space) to partition the data.
- Each split aims to minimize the error (e.g., sum of squared errors) within the resulting regions.
- The final, undivided segments of the input space are the terminal regions.



As shown in Image , our tree divides the regression data's input space (X) into three distinct segments, labeled 1, 2, and 3. These three segments are precisely what we refer to as terminal regions. For any new data point that falls into Region 1, Region 2, or Region 3, the regression tree will output a specific predicted value associated with that particular terminal region.

Step 2(c): Calculating Terminal Region Outputs (γ-values)

Objective:

For each terminal region R_{j1} of the regression tree m_1 , compute a value γ_{j1} that will minimize the loss in that region.

$$\gamma_{j1} = \arg\min_{\gamma} \sum_{x_i \in R_{j1}} L(y_i, f_0(x_i) + \gamma)$$

Here:

- $f_0(x_i)$ is the output from the first model m_0
- L is the loss function (typically squared error loss for regression)
- γ_{j1} is the correction (adjustment) the second model m_1 applies in region R_{j1}

If using Squared Loss Function:

$$L(y_i, \hat{y}_i) = (y_i - \hat{y}_i)^2$$

Then the loss expression becomes:

$$\gamma_{j1} = \arg\min_{\gamma} \sum_{x_i \in R_{j1}} (y_i - (f_0(x_i) + \gamma))^2$$

Which simplifies to:

$$\gamma_{j1} = \text{mean of residuals in region } R_{j1}$$

That is:

$$\gamma_{j1} = \frac{1}{|R_{j1}|} \sum_{x_i \in R_{j1}} (y_i - f_0(x_i))$$

So each region's gamma is just the average of the residuals in that region.

Intuition:

- You trained the first model m_0 on the data.
- You calculated residuals = true value − predicted value.
- The decision tree m_1 splits the data into 3 regions based on inputs.
- Now, you find the average residual in each region, and use that as γ — this tells how much correction is needed in that region.

Summary Workflow:

1. **For Region R_1**:

$$\gamma_{11} = \frac{1}{|R_1|} \sum_{i \in R_1} r_{i1}$$

2. **For Region R_2**:

$$\gamma_{12} = \frac{1}{|R_2|} \sum_{i \in R_2} r_{i1}$$

3. **For Region R_3**:

$$\gamma_{13} = \frac{1}{|R_3|} \sum_{i \in R_3} r_{i1}$$

Then these values γ_{11} , γ_{12} , γ_{13} become the predictions of the second model m_1 for each region.

Summary of γ Calculation for Terminal Node R_{j1}

Step-by-step:

- Region: You calculated γ_{11} for the left-most leaf node, where only x_3 (row index 2) falls.
- Initial Prediction $f_0(x_3)$: 142.333
- Actual Profit y_3 : 91

Applying squared loss:

$$L = (y_i - (f_0(x_i) + \gamma))^2 \Rightarrow \gamma_{11} = \arg\min_{\gamma}(y_3 - f_0(x_3) + \gamma)^2$$

Differentiate and solve:

$$\gamma = y_3 - f_0(x_3) = 91 - 142.333 = -51.333$$

So:

γ_{11} = −51.333

This becomes the prediction made by tree m_1 in that region.

Repeat for Other Regions

Use the same process for the other terminal regions:

For Region with x_2 (row index 1):

- y_2 = 144
- $f_0(x_2)$=142.333

  $\gamma_{21} = y_2 - f_0(x_2)$ = 144 − 142.333 = 1.667

For Region with x_1 (row index 0):

- y_1 = 192
- $f_0(x_1)$ = 142.333

  $\gamma_{31} = y_1 - f_0(x_1)$ = 192 − 142.333 = 49.667

Final Outputs for Step 2(c)

Region R_{j1}	Rows	γ Value
Left Leaf	x_3	-51.333
Middle Leaf	x_2	1.667
Right Leaf	x_1	49.667

These values are exactly what the decision tree in your diagram shows as the value at each terminal node.

Step 2 (d) : Update the Model Output

Current Stage (m = 1):

You have:

- $f_0(x)$: Initial model, which is just the mean of the target (Profit) → 142.33
- $m_1(x)$: First decision tree, trained on residuals (errors from $f_0(x)$)

Update Rule:

For m = 1, the updated model is:

$$f_1(x) = f_0(x) + m_1(x)$$

Where:

- $m_1(x)$ returns the corresponding γ_{jm} for the terminal region x belongs to
- In your case, for each row (training or test), it adds the output of the tree to the base model.

Example:

Using your earlier data:

Row	$f_0(x)$	Terminal Region	Tree Output (γ)	$f_1(x)$(= prediction)
x_1	142.33	Region 3	+49.67	192.00
x_2	142.33	Region 2	+1.667	144.00
x_3	142.33	Region 1	-51.33	91.00

This exactly reproduces the original y values, because this is only the first iteration and you're fitting residuals from the initial mean.

General Rule for Gradient Boosting :

For m trees:

$$F_m(x) = f_0(x) + \sum_{k=1}^{m} m_k(x)$$

If you also include a learning rate ν (usually used to avoid overfitting):

$$F_m(x) = f_0(x) + \nu \cdot \sum_{k=1}^{m} m_k(x)$$

Key Insight:

- Each tree tries to correct the residual (error) from the previous model.
- Over iterations, the model gets better at approximating the true function.
- Final output is initial prediction + sum of all tree outputs.

Gradient Boosting for Regression

If we are using Gradient Boosting for regression, we begin with input features and

an output column (target values). The core idea is to train a series of models sequentially, where each new model tries to correct the errors made by the previous ones.

1. **S**tep 1 – Initialize the Model
   The first model, m_0, is not a decision tree. It simply outputs the mean of the target values from the training data:
   $f_0(x)$ = mean(y)
2. Step 2 – Compute Residuals
   Next, we calculate the residuals (errors) between the actual target values y and the predictions from $f_0(x)$:
   $r_1 = y - f_0(x)$
3. Step 3 – Train the First Tree
   We use the input data and the residuals to train the first decision tree, m_1 .
   The output of this tree approximates the residuals.
4. Step 4 – Combine Models
   We now combine the base model and the first tree:
   $f_1(x) = f_0(x) + m_1(x)$
5. Step 5 – Repeat the Process
   We calculate new residuals:
   $r_2 = y - f_1(x)$
6. Then we train another decision tree m_2 on (x, r_2) and combine it again:
   $f_2(x) = f_1(x) + m_2(x)$
7. Step 6 – Continue Stage-wise Training
   Repeat the process for additional models. For example:
   - Compute $r_3 = y - f_2(x)$
   - Train $m_3(x)$ on r_3
   - Get final model:
     $f_3(x) = f_2(x) + m_3(x)$
8. Final Output
   After all stages, the final prediction is:
   $$\hat{y} = f_0(x) + m_1(x) + m_2(x) + m_3(x)$$
   This is our final boosted model prediction.

We keep adding new decision trees in a stage-wise manner, where each tree tries to fix the residual errors from the previous ensemble. This approach helps build a strong model from multiple weak learners, increasing accuracy and performance over time.

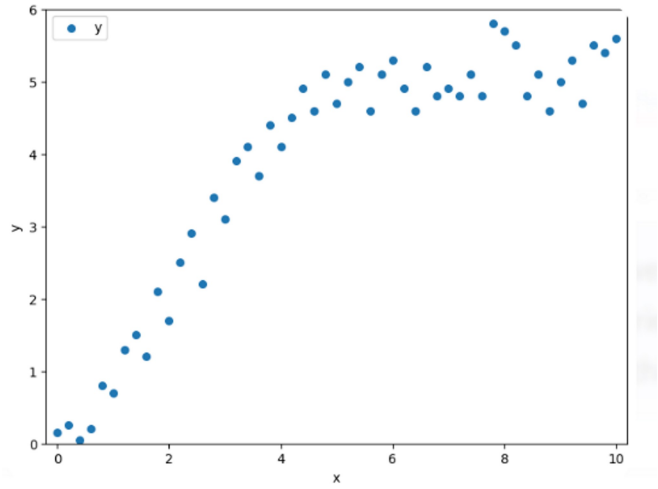
HOW GRADIENT BOOSTING WORKS ?

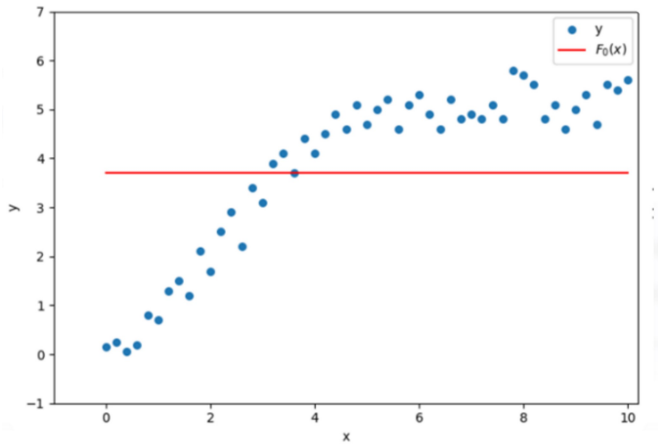
Suppose we have a dataset with one input column, "experience" (x), and one

output column, "salary" (y). When we plot this data, we observe a pattern: after a certain experience level, the salary increases linearly and then becomes stagnant. Our goal is to build a model, f(x), that captures this true relationship between 'x' and 'y'.

1. Initial Model ($F_0(x)$)

Initially, we assume no prior knowledge of the relationship. Our first model, $f_0(x)$, is simply the mean of the 'salary' (y) column. This means, regardless of the 'experience' value, our model always predicts the average salary.



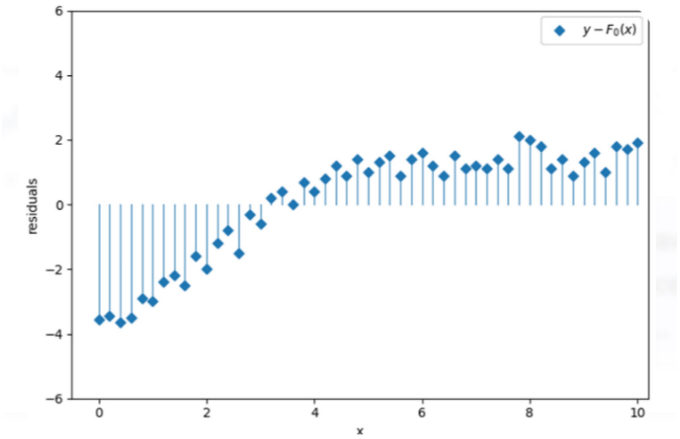


Naturally, this initial model is not perfect and makes significant errors at many data points.

2. Calculating Residuals

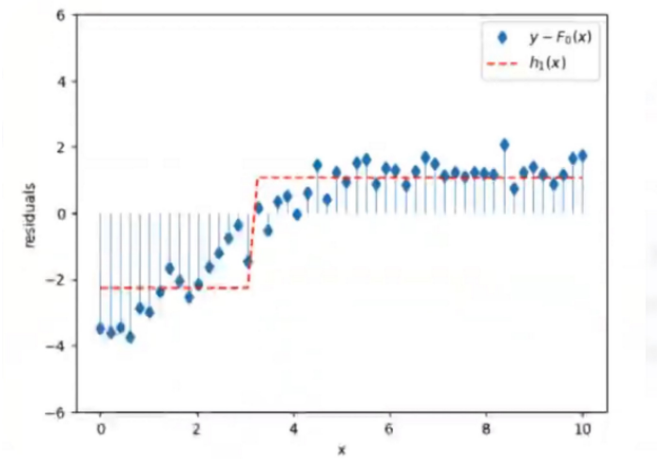
Next, we calculate the residuals, which are the differences between the actual 'salary' (y)
and our model's predictions ($f_0(x)$). We then plot these residuals against the 'experience' (x) values. The scale of the y-axis (now representing residuals)

changes to reflect these differences.



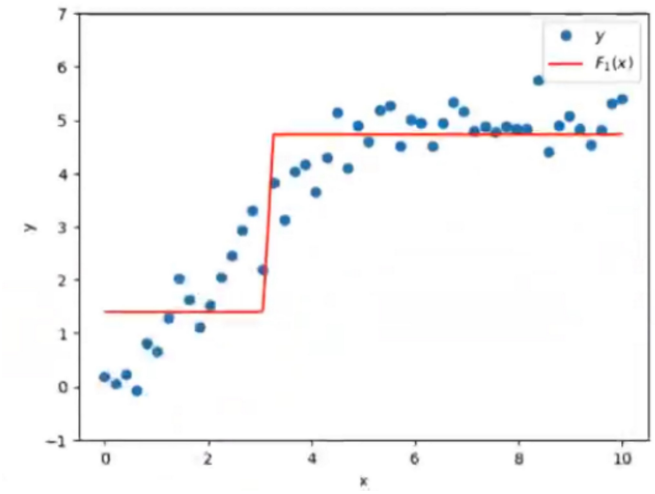
3. Training the First Decision Tree ($h_1(x)$)

In gradient boosting, the crucial next step is to train a new model on these residuals. We use a decision tree, denoted as $h_1(x)$, as our base learner. This decision tree is typically small, often containing only one node, focusing on capturing the patterns in the errors of our previous model.



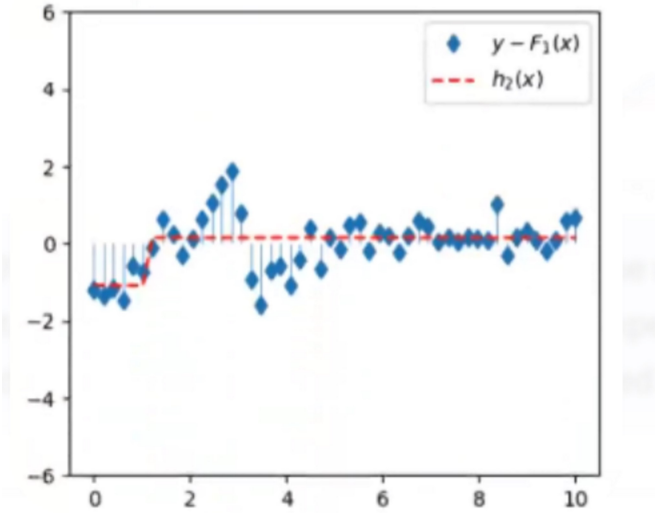
4. Updating the Model ($F_1(x)$)

We then combine our initial model $f_0(x)$ with the newly trained decision tree $h_1(x)$ to create an improved model, $f_1(x) = f_0(x) + h_1(x)$. When we plot $f_1(x)$against 'x' and 'y', we can see that it's a slightly better fit than $f_0(x)$, making fewer mistakes.

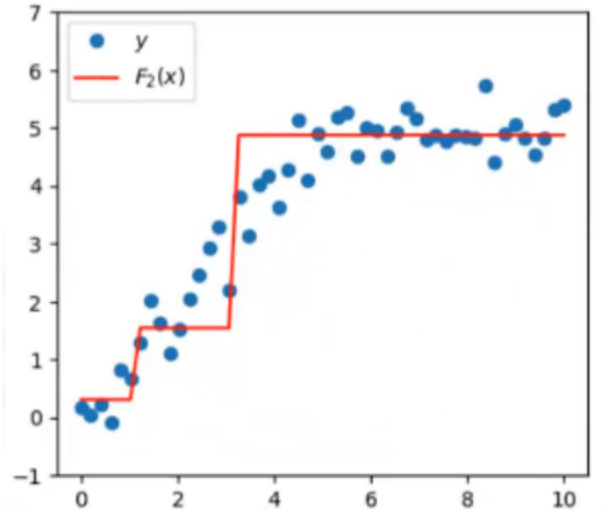


5. Iterative Refinement

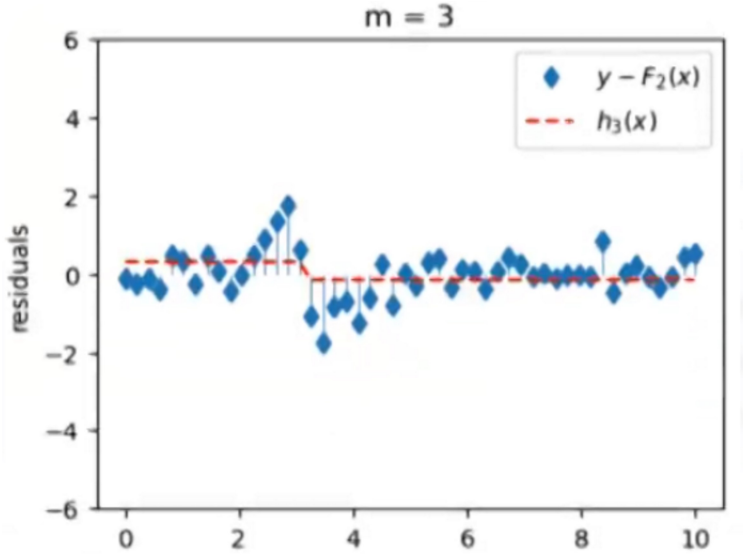
The process continues iteratively. We calculate new residuals, $y - f_1(x)$, and train another decision tree, $h_2(x)$, on these new residuals. These residuals are generally smaller than the previous ones, indicating that our model is gradually improving.



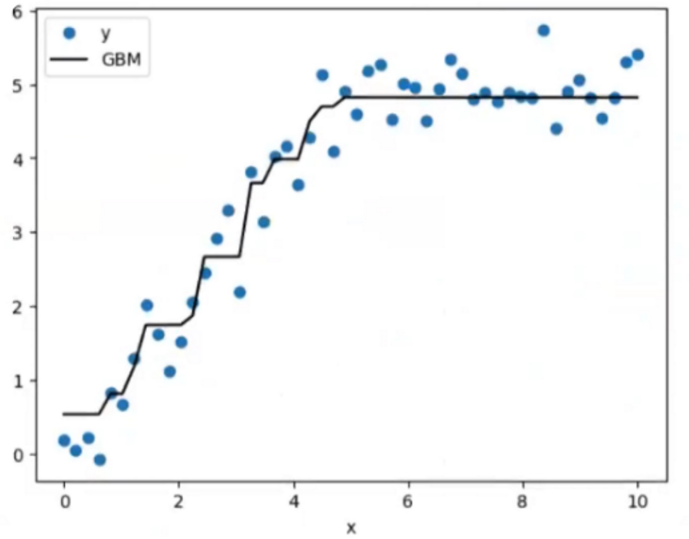
Our model is then updated to $f_2(x) = f_0(x) + h_1(x) + h_2(x)$. As observed, this model is superior to its predecessors, making even fewer errors.



This iterative process of calculating residuals, training a new decision tree on those residuals, and adding it to the ensemble continues. With each iteration, the residuals become smaller, and the model progressively captures more of the underlying data pattern. For instance, after training $h_3(x)$ on $y - f_2(x)$ residuals:



We keep repeating this process. After, for example, 10 iterations, the final model, which is the sum of the initial prediction and all subsequent decision trees, effectively captures the data's pattern and significantly reduces prediction errors.



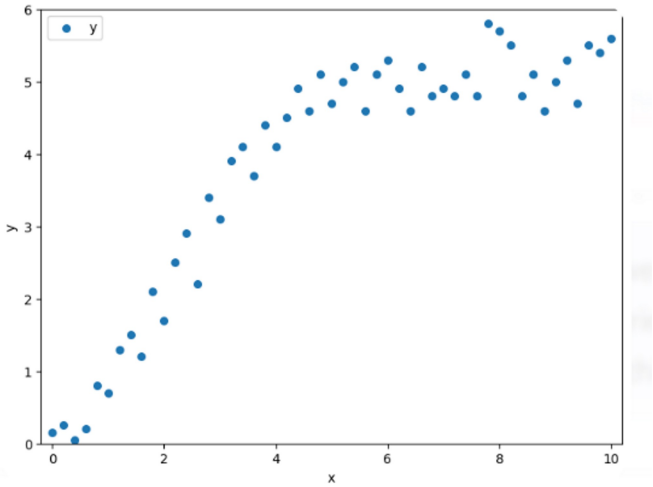
This is the core idea of how Gradient Boosting works: it sequentially builds an ensemble of weak learners (decision trees) by focusing on the errors (residuals) of the previous models, leading to a powerful predictive model.

FUNCTION SPACE VS PARAMETER SPACE

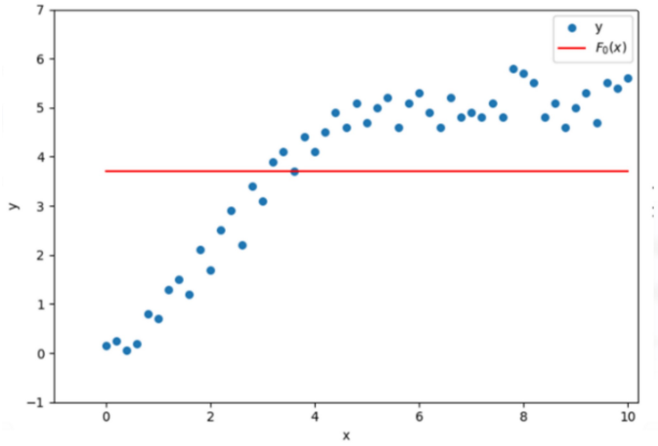
We have a dataset where the x-axis represents 'experience' and the y-axis represents 'salary'. Our objective is to build a machine learning model that can predict salary based on experience. To achieve this, we've decided to apply Gradient Boosting. Our goal is to discover the underlying function, y = f(x), that describes this relationship.

Step 1: Initial Model - $f_0(x)$

Initially, we begin with a simple, often "bad," model. For regression problems, this initial model, denoted as $f_0(x)$, is typically the average (mean) of the output column, 'salary' ($\bar{y}$).



So, our initial machine learning algorithm's prediction function is $y = f_0(x) = \bar{y}$.



This means that irrespective of the input value 'x' (experience), our initial model will always output the same constant value, the mean salary.

Improving the Initial Model

This initial model, $f_0(x)$, is clearly not a great fit as it makes mistakes on many data points. To improve it, our next crucial step is to understand *how much* our current model is wrong. By quantifying these errors, we can then strategically reduce them to move towards a progressively better model.

Quantifying Model Error with a Loss Function

The key to understanding how much error our model is making lies in the loss function. A loss function quantifies the discrepancy between the true output (actual values, y) and the model's predictions ($\hat{y}$). Examples of loss functions include Mean Squared Error (MSE), Log Loss, and Hinge Loss.

For regression tasks, we commonly use Mean Squared Error (MSE). The formula for MSE is:

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

Where:

- y_i is the true output for the i-th data point.
- $\hat{y}_i$ is the model's predicted output for the i-th data point.
- n is the total number of data points.

Let's illustrate this with an example. Suppose we have data for three employees:

Experience (exp)	Salary (y)	Prediction ($\hat{y}$)
10	10	20
20	20	20
30	30	20

Our initial model, F0(x), gives a constant prediction of 20 (the average salary) for

all employees. To calculate the Mean Squared Error for this model:

MSE = $\frac{1}{3}[(10-20)^2+(20-20)^2+(30-20)^2]$

MSE = $\frac{1}{3}[(-10)^2+(0)^2+(10)^2]$

MSE = $\frac{1}{3}[100+0+100]$

MSE = $\frac{1}{3}[200]$

MSE ≈ 66.67

Let's assume, for simplicity, the result was 100 as you initially stated. This calculated number (e.g., 100 or 66.67) represents the error of our initial model. We can indeed quantify the error of our initial model using a loss function.

How to Reduce This Error: Understanding Function Space

The crucial question now is: how do we reduce this error ? To answer this, we need to understand the concept of function space.

Our loss function L(y,$\hat{y}$)depends on two main components:

1. The true labels (y_i) from our dataset, which are constant.
2. The predicted outputs ($\hat{y}_i$) generated by our model.

Let's re-examine the squared error part of the loss function:

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

$$=(y_1-\hat{y}_1)^2+(y_2-\hat{y}_2)^2+(y_3-\hat{y}_3)^2+\cdots+(y_n-\hat{y}_n)^2$$

In this equation, the y_i values are fixed constants because they are the actual values from our dataset. However, the $\hat{y}_i$ values are variables. When we used $f_0(x)$ as our initial model, all our predictions $\hat{y}_i$ were 20. But if we chose a different initial model or an improved model, the predictions $\hat{y}_i$ would change. Therefore, essentially, this loss function is a function of all the individual predictions:

L($\hat{y}_1$,$\hat{y}_2$,$\hat{y}_3$,…,$\hat{y}_n$)

This relationship between the loss function and the predictions $\hat{y}_i$ can be thought of as existing in an (n+1)-dimensional space: n dimensions for the individual predicted values ($\hat{y}_1$,…,$\hat{y}_n$) and one dimension for the loss value itself. The loss will vary as these n prediction variables change.

If we modify our model, our predictions ($\hat{y}_i$) will change, which in turn causes the loss function's value (and thus the error) to change. Our goal in gradient boosting is to systematically adjust these predictions to minimize this loss.

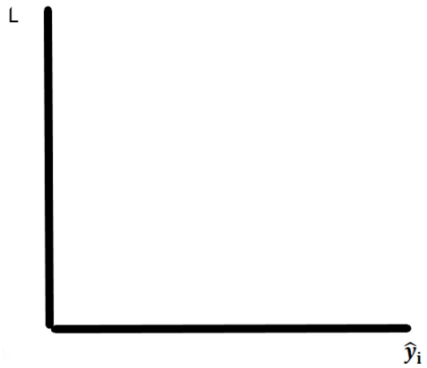
Understanding Model Optimization: Functional Space vs. Parameter Space

To understand how complex algorithms like Gradient Boosting work, we need to visualize how they find the best possible model. This involves understanding the concept of a functional space.

Visualizing the Functional Space

Let's simplify the complex, multi-dimensional problem of model fitting into a 2D graph.

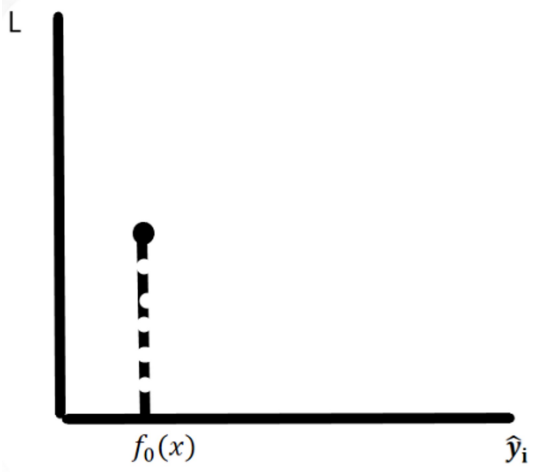
- The y-axis will represent the Loss (L), which tells us how bad our model's predictions are. A lower value is better.
- The x-axis will represent the different models (functions) we can use to make predictions.



Step 1: The Initial Model

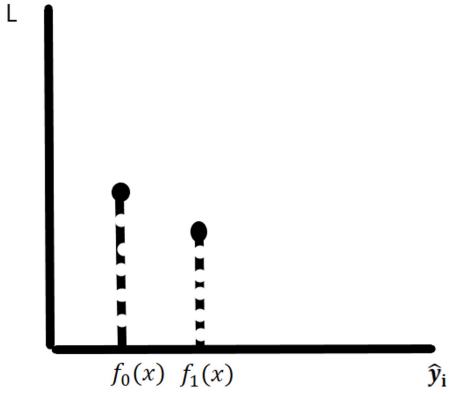
We start with a very simple, baseline model, which we'll call f_0(x). A common starting point is a model that predicts the average value of our target variable, y. Let's say the average is 20.

- Model: $f_0(x)$ = 20
- Prediction: For every input x_i, the prediction $\hat{y}_i$ is 20.
- Loss: We can now calculate the total loss (e.g., using Mean Squared Error - MSE) for this model. This gives us our first point on the graph.

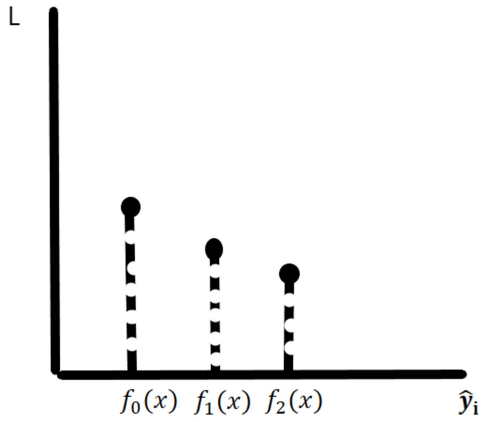


Step 2: Iterating and Finding Better Models

Now, we try to find a better model. We introduce a new function, $f_1(x)$ (e.g., $f_1(x)$ = 3x + 2). This new model will produce a new set of predictions ($\hat{y}_i$) and, consequently, a new loss value. We plot this new model and its loss.



We can repeat this process with another model, $f_2(x)$ (e.g., 2x + 3), which gives us yet another set of predictions and a new loss value.



The Goal: Minimizing Loss in Functional Space

The key idea is that we can try an infinite number of functions—linear, curved, or any other mathematical form. Each function is a unique point on the x-axis of our graph. This "space" of all possible functions is called the functional space.
Our goal is to find the function f(x) that results in the minimum possible loss. This function represents the best possible model because it captures the true relationship within our data most accurately.

Contrast with Parameter Space: The Case of Linear Regression

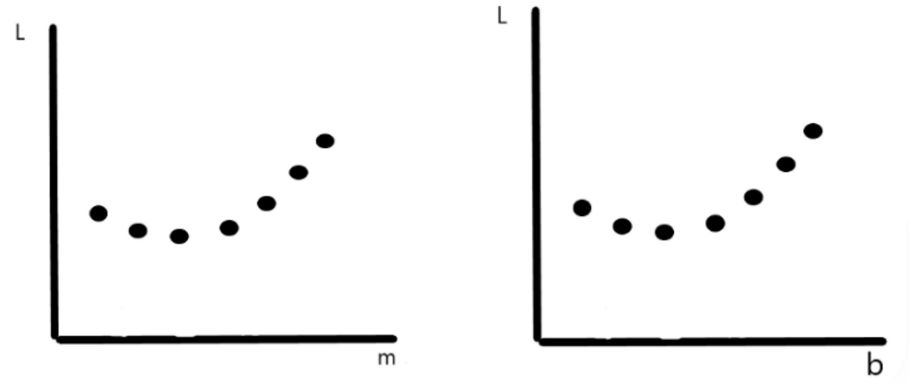
This approach is different from simpler, parametric machine learning algorithms like Linear Regression.

A parametric algorithm assumes the form of the function from the beginning. Linear regression, for instance, assumes the relationship between the variables is always a straight line:

$$y = f(x) = mx + b$$

Here, the goal is not to find the *type* of function (it's already assumed to be linear). The goal is to find the optimal parameters (m for the slope and b for the intercept) that create the best-fitting line.

The space we search for these optimal values is called the parameter space. In this space, the axes are not functions, but the parameters themselves (m and b). For each combination of m and b, we calculate a loss. Our objective is to find the specific values of m and b that correspond to the lowest point on the loss curve.



Function Space vs Parameter Space

- In parametric algorithms (e.g., Linear Regression), the model assumes a specific function form:

  $$y = f(x) = mx + b$$

  - The algorithm doesn't find the function form—it only finds the best parameters m and b
  - So the graph has parameters on x-axis → it's called parameter space

- In non-parametric algorithms (e.g., Gradient Boosting):
  - We don't assume the function form in advance
  - We try many possible functions (additive models, trees, etc.)
  - So the x - axis holds entire functions → it's called function space

Summary of Key Differences

Feature	**Functional Space (e.g., Gradient Boosting)**	**Parameter Space (e.g., Linear Regression)**
What's on the x-axis?	Entire functions: $f_0(x), f_1(x)$, etc.	Model parameters: m, b, etc.
What is the	To find the optimal function that	To find the optimal parameters

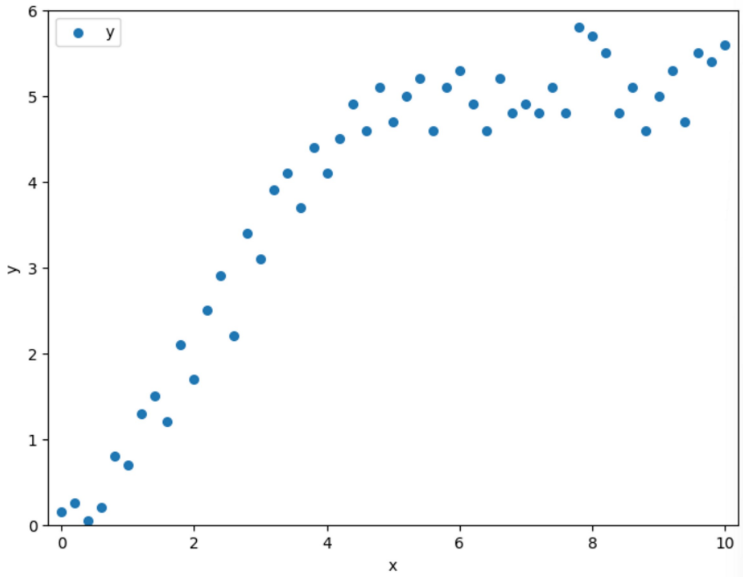
goal?	minimizes loss.	for a pre-defined function.
Model Type	Typically non-parametric. It does not assume the data follows a specific structure.	Parametric. It assumes the data follows a specific structure (e.g., a line).

DIRECTION OF LOSS MINIMIZATION

1. Our Dataset and Initial Scatter Plot

We begin with a set of observed points (x_i, y_i) . To visualize:

- Scatter plot of the data: points $\{(x_i, y_i)\}$ showing how y varies with x.



2. Fitting the First (Constant) Model $f_0(x)$

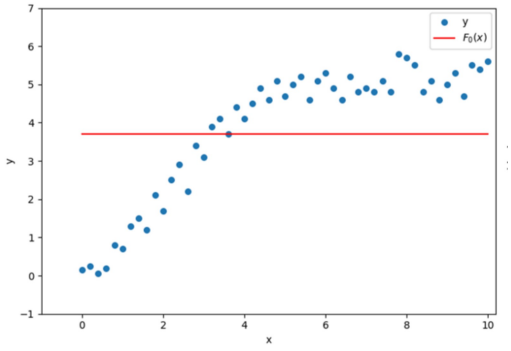
1. Choose an initial model
   In gradient boosting, a common choice is a constant function equal to the mean of all y_i :

$$f_0(x) = \bar{y} = \frac{1}{n}\sum_{i=1}^{n} y_i.$$

   In our case, $\bar{y} \approx 3.7$ (for example) .
2. Compute predictions
   Every data point's prediction under f_0 is

$$\hat{y}_i = f_0(x_i) = \bar{y} \quad \forall i.$$

3. Visualizing $f_0(x)$ on the data
   Draw a horizontal line at $y = \bar{y}$ on the scatter plot.



4. Compute the initial loss
   Using Mean Squared Error (MSE) as our loss:

$$L(f_0) = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 = \frac{1}{n}\sum_{i=1}^{n}(y_i - \bar{y})^2.$$

   On the graph below (in function-space), this loss corresponds to one point on a parabola (since MSE vs. $\hat{y}$ is quadratic).

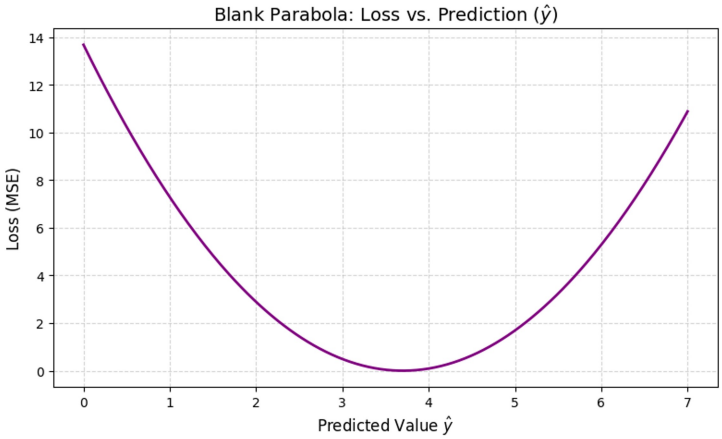
3. Visualizing Loss as a Function of Prediction ("Function Space")

We now switch perspective: instead of plotting y vs. x, we plot loss on the vertical axis L and predicted value $\hat{y}$ on the horizontal axis. Since

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

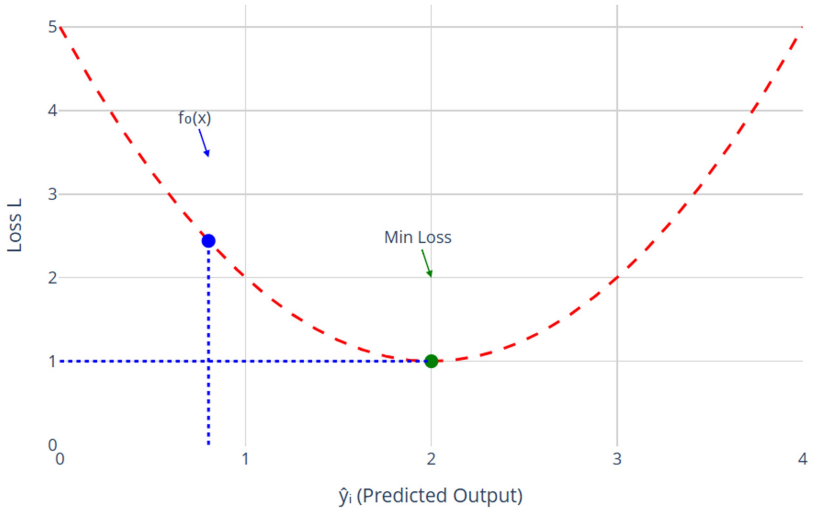
(with $\hat{y}$ being a constant, like in a simple model), this is a parabolic curve (a $(y - \hat{y})^2$ shape).

1. Blank parabola
   Imagine the parabola L($\hat{y}$) without any points on it.



2. "Our current location" on the parabola

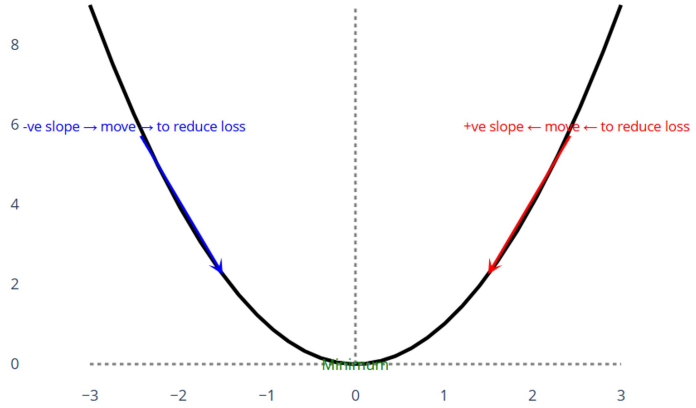
- Because $f_0(x) = \bar{y}$, each predicted value $\hat{y}_i$ equals $\bar{y}$.
- On the parabola L($\hat{y}$), we pick the point at $\hat{y}$=$\bar{y}$.
- That point's vertical coordinate is the current loss L(f_0).



4. How Do We Move Toward Minimum Loss ?

The parabola's minimum is where L($\hat{y}$) is smallest (the vertex). Since we are currently at $\hat{y} = f_0(x)$, we need to know which direction on the $\hat{y}$-axis to move in order to decrease L . That direction is given by the slope (derivative) of the parabola.

1. Interpreting slope on a parabola
   - If the slope $\frac{dL}{d\hat{y}}$ at our current $\hat{y}$ is positive, then moving to the left (decreasing $\hat{y}$) lowers the loss.
   - If $\frac{dL}{d\hat{y}}$ is negative, then moving to the right (increasing $\hat{y}$) lowers the loss.



2. Key idea
   Since we don't know in advance whether to increase or decrease our prediction for each observation, we calculate the slope of L with respect to

each individual prediction $\hat{y}_i$. In other words, we find:

$$(\text{negative})\ \text{gradient}_i \ = \ -\frac{\partial L}{\partial\, \hat{y}_i} \quad \text{evaluated at } \hat{y}_i = f_0(x_i).$$

For MSE,

$$L \ = \ \sum_{i=1}^{n} \left(y_i - \hat{y}_i\right)^2 \ \Longrightarrow \ \frac{\partial L}{\partial \hat{y}_i} \ = \ -2\left(y_i - \hat{y}_i\right).$$

Thus,

$$-\frac{\partial L}{\partial \hat{y}_i}\Big|_{\hat{y}_i = f_0(x_i)} \ = \ \ 2\left(y_i - f_0(x_i)\right).$$

(We usually drop the constant factor 2 and simply call the residual $r_i = y_i - f_0(x_i)$.)

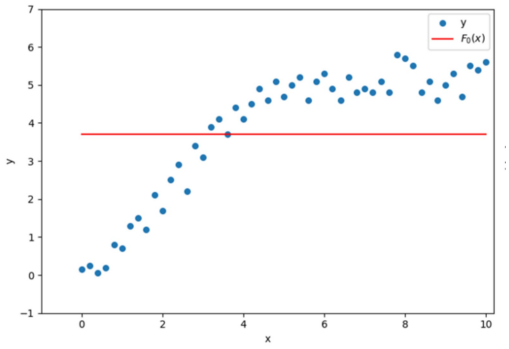
5. Computing Residuals for Each Data Point

1. Residuals

$$r_i = y_i - f_0(x_i) \quad for\ i \ = \ 1, \ldots, n.$$

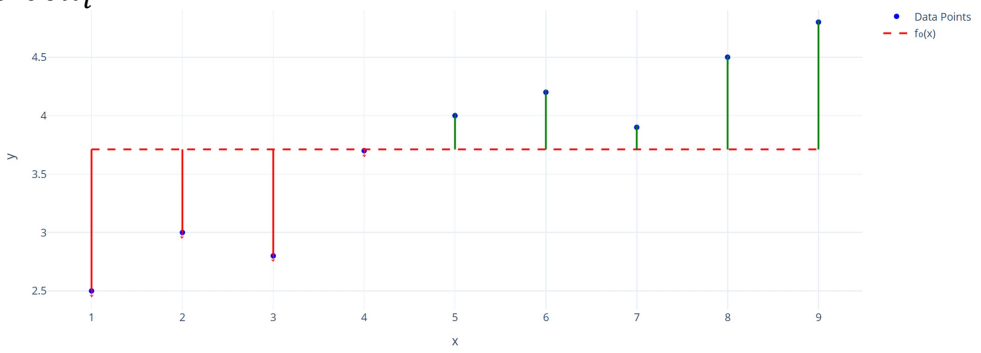
   - If $y_i > f_0(x_i)$, then $r_i > \ 0$. That means:
     Point i lies above the constant line, so the loss-vs-prediction slope is negative (we need to move $\hat{y}_i$ upward to reduce loss) .
   - If $y_i < f_0(x_i)$, then $r_i < \ 0$. That point lies below the constant line, so the slope is positive (we need to move $\hat{y}_i$ downward) .

2. Visualize residuals on the original scatter
Re-draw the scatter of (x_i, y_i) with the constant line f_0 . For each point, indicate its vertical distance (residual) from the line.



3. Interpretation
   - Points above the line: residual $r_i > \ 0$. Want to increase prediction at that x_i .

- Points below the line: residual $r_i < 0$. Want to decrease prediction at that x_i.



6. Fitting the First Decision Tree to Residuals

In gradient boosting, we now treat the residuals r_i as the "target values" for our first weak learner, typically a regression tree $h_1(x)$. Concretely:

1. Compute each residual
   $r_i = y_i - f_0(x_i)$
2. Train a regression tree
   - Let the tree $h_1(x)$ approximate r_i for each x_i .
   - In other words, we set

$$h_1(x_i) \approx r_i \implies h_1(x) \text{ is trained to minimize } \sum_{i=1}^{n} (r_i - h_1(x_i))$$

3. Why this works
   By fitting $h_1(x)$ to the negative gradient $\{r_i\}$, we capture, for each region of x, whether we need to push predictions up or down. At the end of this tree-fitting step, $h_1(x)$ provides a rule like: "At x in a certain region, add this amount to reduce loss."

7. Updating the Model and Proceeding to $f_1(x)$

1. **Model update**

   Once h_1 is trained, we form

$$f_1(x) = f_0(x) + \nu h_1(x),$$

   where $0 < \nu \leq 1$ is a small learning rate (e.g., $\nu = 0.1$).

   - This update moves each prediction a bit in the direction that decreases the loss.

2. **Iterate**
    - Recompute residuals $r_i^{(1)} = y_i - f_1(x_i)$.
    - Fit a second tree $h_2(x)$ to these new residuals.
    - Update $f_2(x) = f_1(x) + \nu\, h_2(x)$.
    - Continue for M rounds or until performance stops improving.

8. Recap of Key Concepts

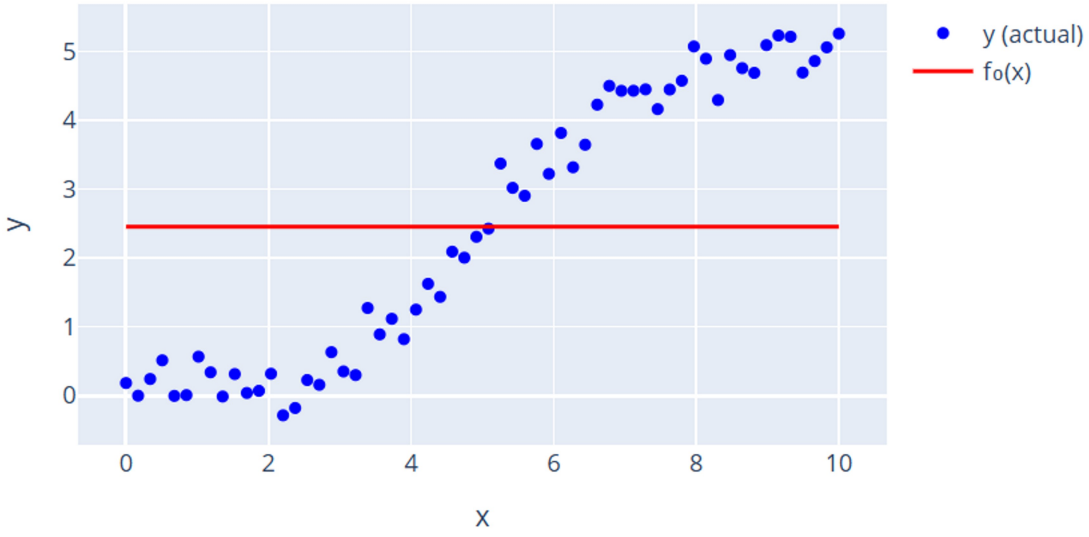
1. **Function Space vs. Parameter Space**
    - In *function space* (non-parametric), the x-axis represents *entire candidate functions* (e.g., constant lines, straight lines, curves, trees, etc.). We measure each function's loss and move toward the minimum by following the loss-vs-prediction slope (the gradient).
    - In *parameter space* (parametric methods like linear regression), the x-axis represents parameters (e.g., m and b). The algorithm simply finds the best (m, b) to minimize loss.

2. **Gradient as Direction**
    - At each step, computing $r_i = y_i - f_{m-1}(x_i)$ (the negative gradient of MSE) tells us, for each data point, whether we should push the model's prediction up or down.
    - Fitting a regression tree to $\{(x_i,\, r_i)\}$ builds a function $h_m(x)$ that approximates these residuals. Adding $h_m(x)$ to the existing model moves us closer to the loss-minimum in function space.

3. **Visual Summary**
    - **Scatter plot** shows raw data and current model $f_{m-1}(x)$.
    - **Function-space plot** (parabola) illustrates the loss for a given prediction value $\hat{y}$. The slope of that parabola at $\hat{y} = f_{m-1}(x_i)$ is exactly proportional to $(y_i - f_{m-1}(x_i))$.
    - By computing these slopes for each point and fitting a tree to them, we take a "step" in function space toward the minimum loss.

Update the Function in Gradient Boosting

Now we have data like this (see below). We drew a constant function $f_0(x)$, which simply outputs the mean of the target values y . This places us somewhere in the function space.
We want to improve this model, and we can do that using the slope, or in this case, the residuals $y_i - f_0(x_i)$. These residuals give us a direction — telling us where and how far we need to move to minimize error.



Gradient Boosting Update Intuition

Let's say we want to update this function and create a new function $f_1(x)$. From gradient descent, we know the update rule :
$m_{new} = m_{old} - \eta \cdot slope$
In our case:

- $m_{old} \rightarrow f_0(x)$
- $slope \rightarrow y_i - f_0(x_i)$
- So ,

$$f_1(x_i) = f_0(x_i) - (-(y_i - f_0(x_i))) = f_0(x_i) + (y_i - f_0(x_i)) = y_i$$

This means our model perfectly predicts the true values y_i. This would lead to zero loss, but also massive overfitting. The model memorizes the training data, which is not desirable.

Why Not Directly Add the Residual ?

If we add the full residual $y_i - f_0(x_i)$, we cancel out $f_0(x_i)$, and the model outputs y_i directly:

$f_1(x_i) = f_0(x_i) + (y_i - f_0(x_i)) = y_i$

This leads to overfitting, because the model fits the training data exactly and performs poorly on new data.

So What's the Solution?

We treat this residual as an approximation target. Let's denote the residual as:

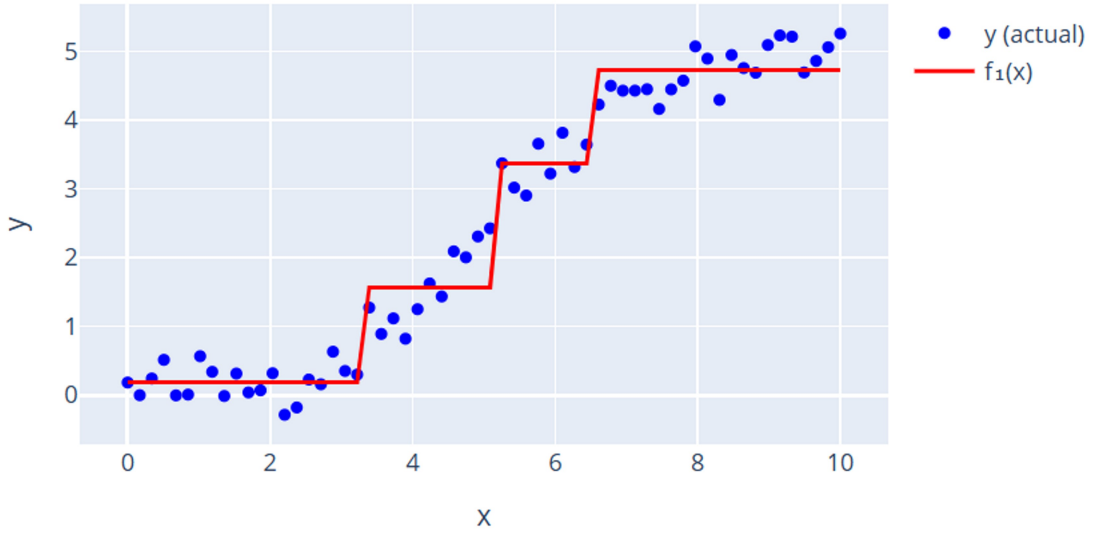
$a_i = y_i - f_0(x_i)$

Now, instead of adding a_i directly, we approximate it using a weak learner — typically a shallow decision tree $h_1(x)$. This tree is trained to predict a_i , not perfectly, but closely.

This gives us the new function:

$f_1(x) = f_0(x) + h_1(x)$

This prevents overfitting and moves us closer to the optimal model step-by-step.

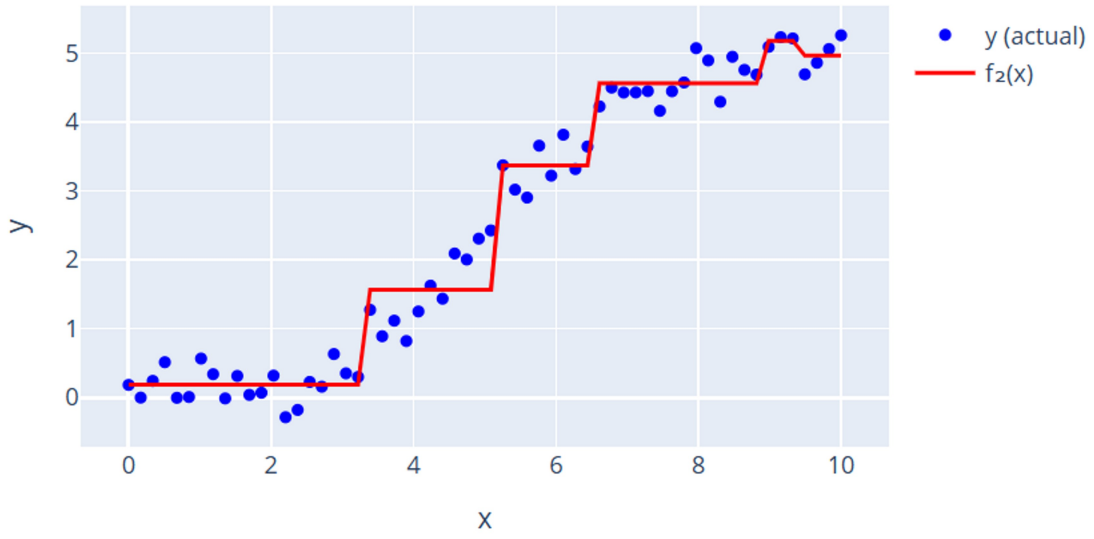


Iterate Again: Second Update

Now, we again calculate residuals:

$Residual_2 = y - f_1(x)$

And train another weak learner $h_2(x)$, and then:
$f_2(x) = f_1(x) + h_2(x)$
This process is repeated, each time correcting the previous model's mistakes slightly using weak learners.

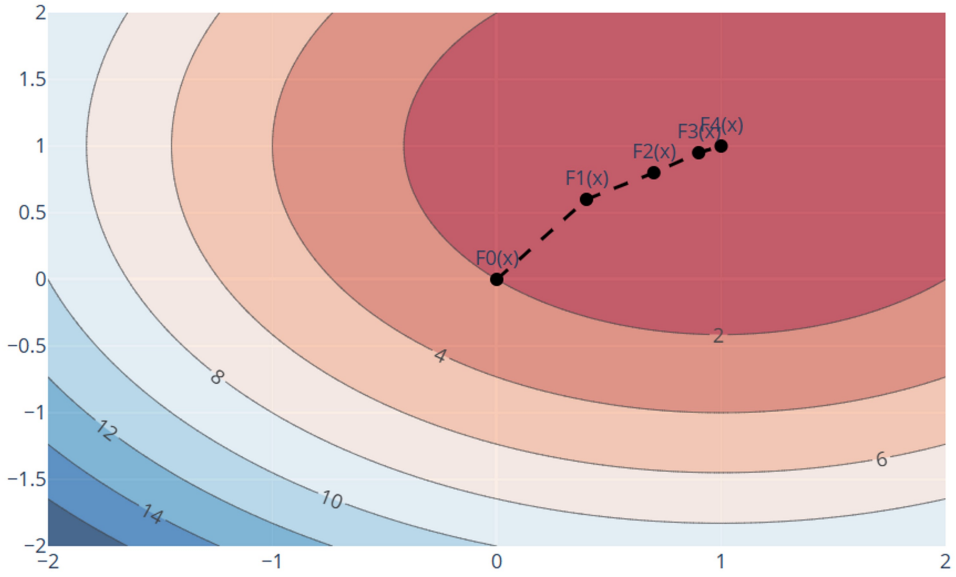


Moving in Function Space

As we continue training our model using weak learners, we are not just updating predictions — we are navigating the loss surface in function space.
Each weak learner points in the direction of steepest descent, and we take a step along that direction to reduce the loss.
You can imagine this process like moving on a 2D terrain where the height represents loss (error), and each move with a weak learner is a small step downhill toward the minimum.

- We start at an initial point $F_0(x)$ (e.g., mean).
- Each weak learner (like h_1, h_2, ...) pushes us a little closer to the best possible model.
- After n steps, $F_n(x)$ becomes a good approximation of the true function.

The figure below shows how each updated model moves us closer to the optimal prediction (minimum loss):



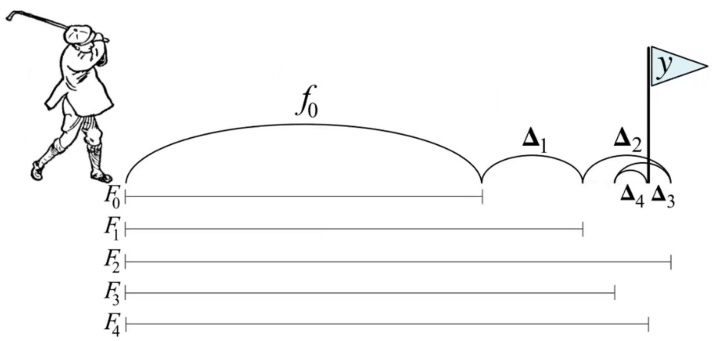
What This Shows:

- A contour plot of a convex loss surface (like MSE).
- Each F_0, F_1, F_2 ... point is a stage in boosting.
- The steps visually represent how gradient boosting minimizes loss by moving in the direction of the negative gradient in function space.

Final Summary

- At each step, we:
  1. Compute the residual.
  2. Train a weak learner to predict it.
  3. Add the weak learner to the previous model.
- This gradually builds a strong predictive model.
- It avoids overfitting by using weak learners and only partially correcting errors.

Intuition Behind Gradient Boosting (Golf Player Analogy)



Imagine a golf player aiming to putt a ball into a hole—the hole represents the true value y, and the player's goal is to reach it with a series of shots.

1. First Shot – The Initial Guess
   The player takes the first shot, and the ball lands somewhere far from the hole. This is our initial model:
   $f_0(x)$ = mean(y)
   This is just a naive guess—like the average of all outcomes.
2. Second Shot – Correcting the Mistake
   The player measures the distance to the hole from where the ball landed. This distance is called the residual (the error we made):
   $$r_1 = y - f_0(x)$$
   The player now trains a decision tree to predict this residual. This correction is:
   $h_1(x)$
   After applying this correction, the ball moves closer to the hole:
   $$f_1(x) = f_0(x) + h_1(x)$$
3. Third Shot – More Fine-tuning
   Now the player again checks how far the ball is from the hole:
   $$r_2 = y - f_1(x)$$
   Another decision tree is trained on this new residual, giving:
   $$h_2(x)$$
   Apply it:
   $$f_2(x) = f_1(x) + h_2(x)$$
4. More Shots – Approaching the Goal
   The player may overshoot or still be a bit off, so they keep training more trees on the remaining error:
   $$f_3(x), f_4(x), \ldots, f_m(x)$$
   until the predicted value is very close to the actual target y.
1. Final Output
   After multiple attempts (shots), the player finally lands the ball in the hole:
   $$\hat{y} = f_0(x) + h_1(x) + h_2(x) + \cdots + h_m(x)$$

Gradient Boosting works like a golfer learning from each shot, adjusting each new attempt to move closer to the target. Each decision tree focuses only on the mistakes (residuals) of the previous steps, and collectively they form a powerful predictive model.

Iterative Process in Gradient Boosting

Gradient Boosting builds the model step-by-step. Let’s walk through the process using the following images:

Step 1: Initial Model $f_0(x)$

In the beginning, we initialize our model $f_0(x)$ with a constant value — typically the mean of all target values y. As seen in the plot, this is just a flat horizontal line. Clearly, it doesn't fit the data well.

Step 2: Compute Residuals and Train $h_1(x)$

At this stage, we compute the residuals — the difference between actual values y_i and predicted values $f_0(x_i)$. These residuals indicate how far off we are and in which direction we need to move. This is our "gradient" in function space.

Step 3: Fit Weak Learner $h_1(x)$ and Update Model

Now, we train a weak learner (typically a shallow decision tree $h_1(x)$) on the residuals. This learner learns the direction in which the model should adjust. We then update our model:

$$f_1(x) = f_0(x) + h_1(x)$$

The new function $f_1(x)$ fits the data a bit better — especially in places where the residuals were large.

Step 4: Repeat for Further Improvement

The process is repeated:

- Compute residuals for $f_1(x)$
- Train a new weak learner $h_2(x)$ on these residuals
- Update the model:

$$f_2(x) = f_1(x) + h_2(x) = f_0(x) + h_1(x) + h_2(x)$$

This iterative process continues — each new model improving upon the last — until the final boosted model minimizes error.

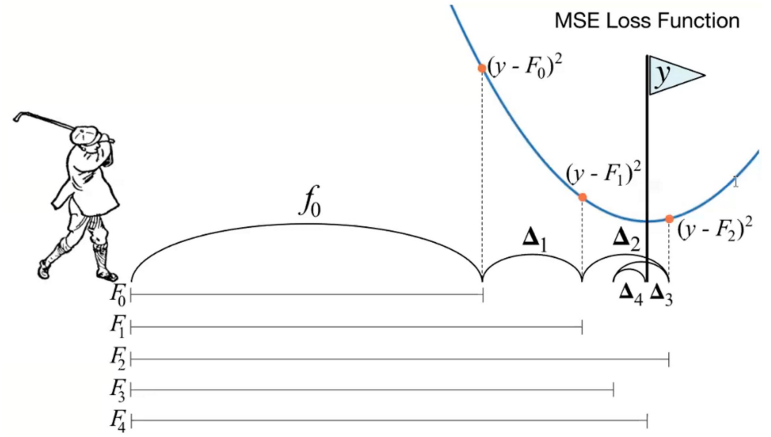
Final Outcome

After several such iterations (say up to $f_3(x), f_4(x)$, ….), we get a model that closely fits the training data and learns complex non-linear patterns.

Key Point:

The power of Gradient Boosting lies in combining many weak learners to form a strong model, where each learner tries to correct the mistakes of the previous one.

ANOTHER PERSPECTIVE OF GRADIENT BOOSTING



Let's understand Gradient Boosting from a more intuitive, visual perspective using the golfer analogy.
Imagine a golfer (our model) trying to reach a target flag (the correct prediction y) on a loss landscape (such as the MSE loss function curve).

1. First Shot – $f_0(x)$
2. The golfer takes the first shot and lands somewhere off target in function space (initial model f_0). We realize that this is not the correct prediction because the loss $(y - f_0)^2$ is high.
3. Compute Residual & Train Tree
   To improve, we compute the residual (difference between the actual y and current prediction $f_0(x)$) and train a decision tree to predict that residual. This gives us $f_1(x)$, a model closer to the target.
4. Repeat the Process
   We repeat this process multiple times:
   - Compute residuals again,
   - Train new weak learners (trees) on those residuals,
   - Update the function:

$$f_m(x) = f_{m-1}(x) + \eta \cdot h_m(x)$$

5. Convergence
   Eventually, after a few iterations (say by f_4), we reach close to the actual target value y. This is represented in the diagram by the function moving step-by-step closer to the flag on the curve.

Conceptual Insight

This is very similar to gradient descent, but with a key difference:

- Gradient Descent:
  Moves in parameter space by updating weights using gradients.
- Gradient Boosting:

Moves in function space by sequentially adding weak learners that correct the residuals (gradient of loss) from the previous model.

Summary

What gradient descent does in parameter space, gradient boosting does in function space.
Each step is like a controlled shot in the right direction using weak learners to gradually reach the goal.

Difference Between Gradient Descent and Gradient Boosting

Gradient Descent and Gradient Boosting are both optimization strategies, but they work in different spaces and follow different update rules.

1. Working Space

- Gradient Descent:
  Operates in the parameter space.
  It updates the model by adjusting parameters (like weights and biases) to minimize the loss.
  Example:
  $$m_{\text{new}} = m_{\text{old}} - \eta \cdot \text{slope}$$
- Gradient Boosting:
  Operates in the function space.
  It builds a sequence of models (typically decision trees) to correct the errors of the previous model.
  Example:
  $$f_m(x) = f_{m-1}(x) + \eta \cdot h_m(x)$$
- where $h_m(x)$ is a weak learner (e.g., decision tree) trained to predict residuals.

2. Update Rule

- Gradient Descent:
  The update rule is:
  $$\theta_m = \theta_{m-1} - \eta \cdot \nabla_\theta L(\theta)$$
  It directly updates the parameters of the model.
- Gradient Boosting:
  The update rule is:
  $$f_m(x) = f_{m-1}(x) + \eta \cdot h_m(x)$$
- It adds a new function h_m(x) to correct the residuals from the previous function.

Feature	Gradient Descent	Gradient Boosting
Space	Parameter space	Function space
Update Rule	$m_n = m_0 - \eta \cdot \text{slope}$	$f_m(x) = f_{m-1}(x) + \eta \cdot h_m(x)$
Learner Used	Usually linear (e.g., weight updates)	Usually decision trees (weak learners)
Focus	Minimizing loss by tuning parameters	Minimizing residuals by combining learners

Advantages of Gradient Boosting

Gradient Boosting is one of the most powerful machine learning techniques. Below are its key advantages:

1. High Predictive Accuracy

- Gradient Boosting often delivers state-of-the-art performance in both regression and classification tasks.
- It minimizes errors stage-by-stage, which leads to highly accurate models.

2. Handles Non-Linear Data Well

- Since it builds an ensemble of decision trees, it can capture complex, non-linear relationships in the data.

3. Feature Importance

- It naturally provides feature importance scores, helping you understand which features are most influential.

4. Robust to Overfitting (with Regularization)

- Techniques like:
  - shrinkage (learning rate),
  - limiting tree depth,
  - subsampling

  help reduce overfitting, making it more generalizable than simple ensembles.

5. Flexible Framework

- It can optimize different loss functions (e.g., MSE, MAE, log-loss).
- It can also be adapted to ranking, classification, and other tasks.

6. Works with Missing Data

- Some implementations like XGBoost and LightGBM can handle missing values internally.

7. No Need for Feature Scaling

- Unlike models like SVM or Logistic Regression, Gradient Boosting doesn't require normalization or standardization of features.

8. Custom Loss Functions

- You can define and plug in custom loss functions depending on the problem, which gives high flexibility.