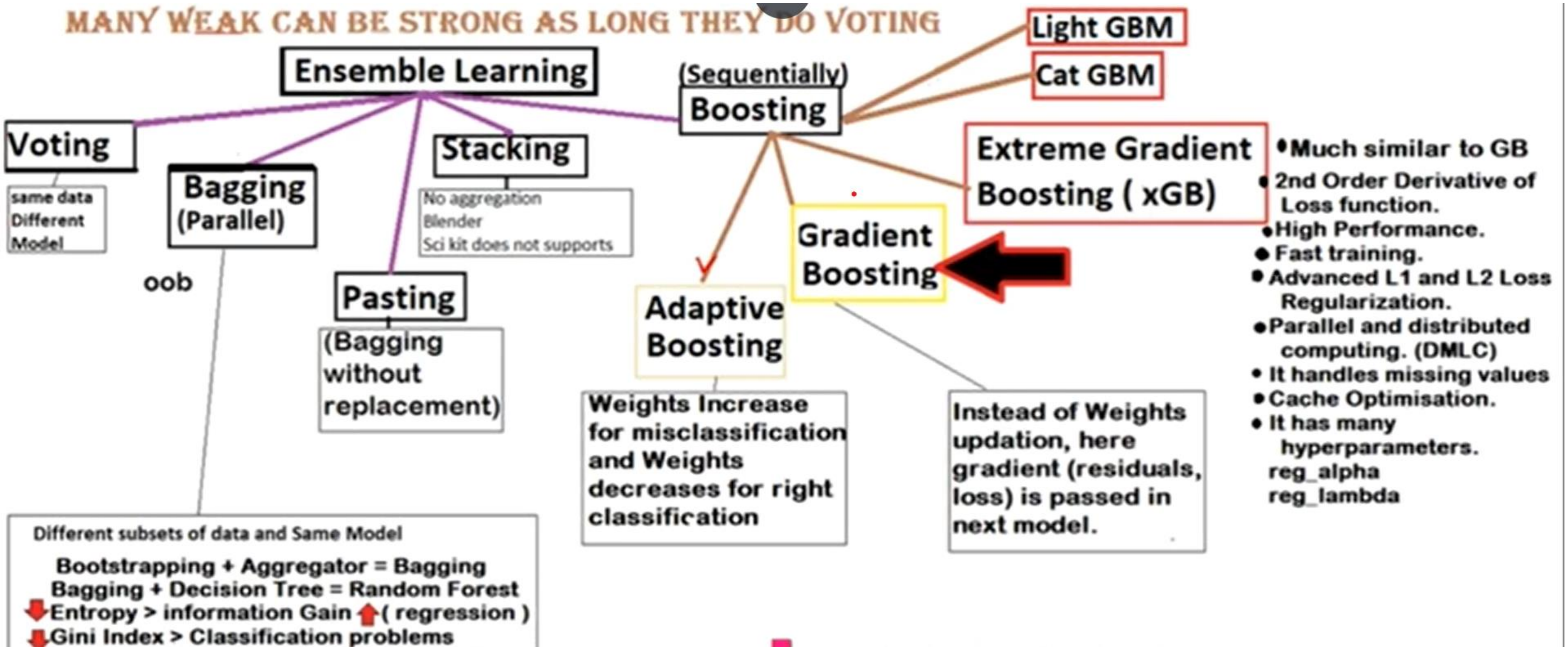
Gradient Boosting Machine (GBM) and XGBoost

Ensemble Learning



(Bagging without replacement)

Weights Increase for misclassification and Weights decreases for right classification

Instead of Weights updation, here gradient (residuals, loss) is passed in next model.

- Much similar to GB
- 2nd Order Derivative of Loss function.
- High Performance.
- Fast training.
- Advanced L1 and L2 Loss Regularization.
- Parallel and distributed computing. (DMLC)
- It handles missing values
- Cache Optimisation.
- It has many hyperparameters.
  reg_alpha
  reg_lambda

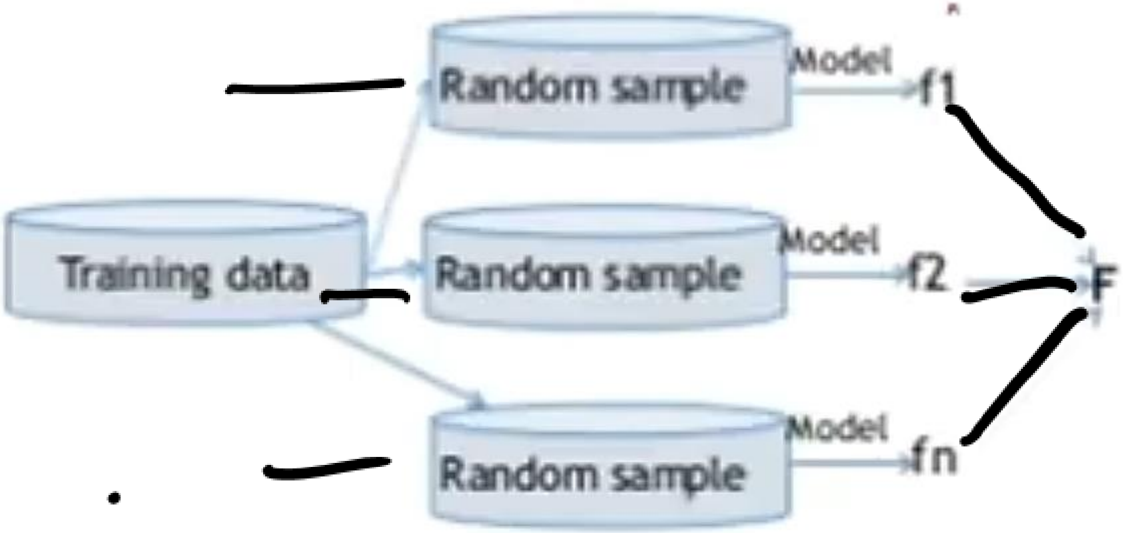
Different subsets of data and Same Model

Bootstrapping + Aggregator = Bagging
Bagging + Decision Tree = Random Forest
Entropy > information Gain (regression)
Gini Index > Classification problems

Types of Boosting

- Adaptive Boosting (AdaBoost)
- Gradient Boosting (GB)
- Extreme Gradient Boosting (xG Boost)
- Light GBM
- Cat GBM

Bagging



Training data

Random sample

Model

f1

Random sample

Model

f2

Random sample

Model

fn

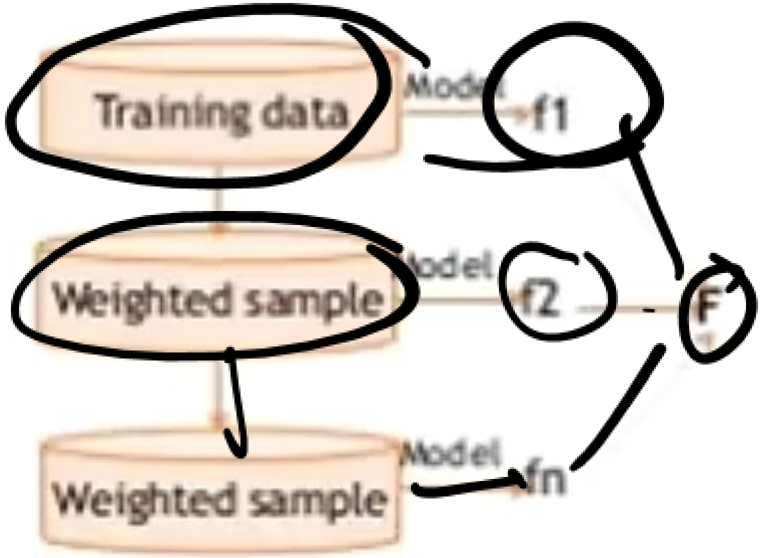
F

Resampling

Uniform distribution

Parallel Style

Boosting



Training data

Model

f1

Weighted sample

Model

f2

Weighted sample

Model

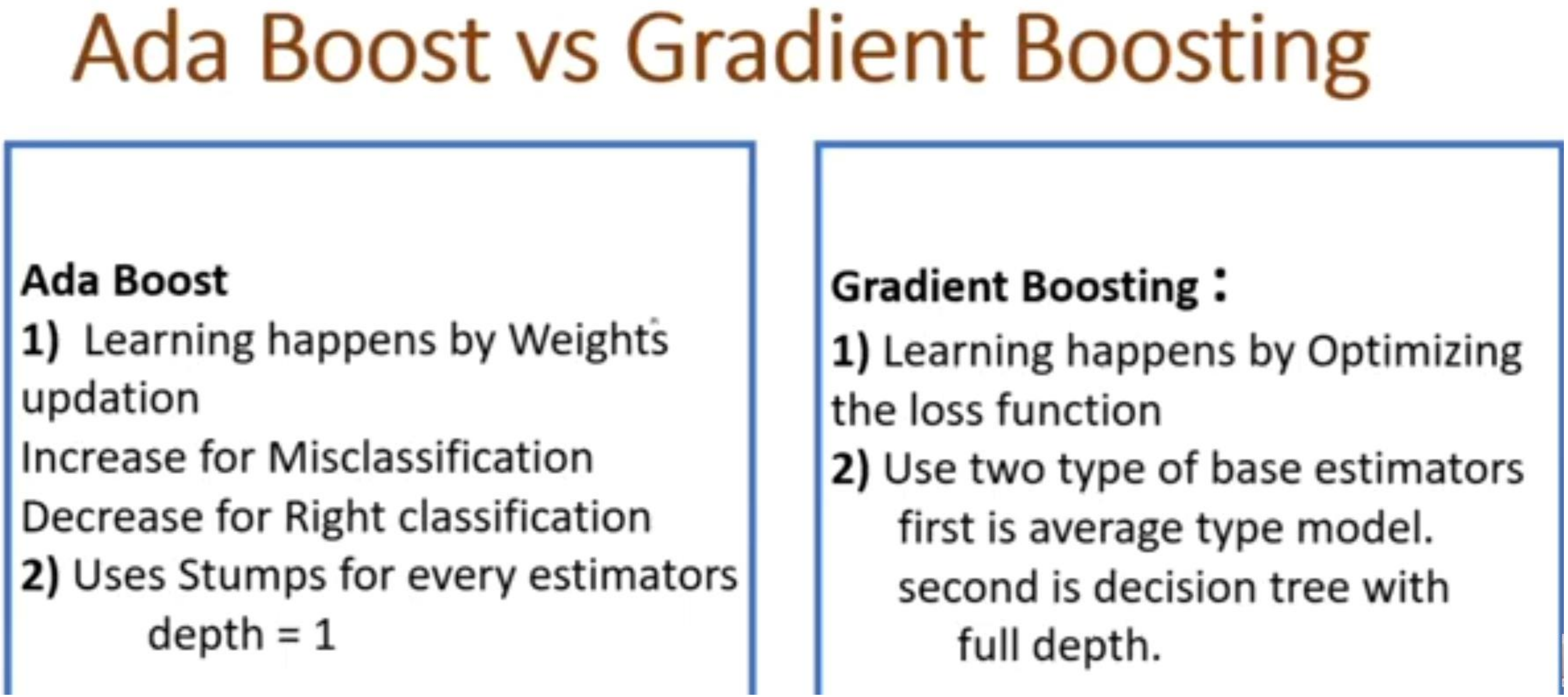
fn

F

Reweighting

Non-uniform distribution

Sequential Style



Ada Boost vs Gradient Boosting

Ada Boost

1) Learning happens by Weights updation

Increase for Misclassification

Decrease for Right classification

2) Uses Stumps for every estimators

depth = 1

Gradient Boosting :

1) Learning happens by Optimizing the loss function

2) Use two type of base estimators

first is average type model.

second is decision tree with full depth.

“Why bother with gradient boosting?”

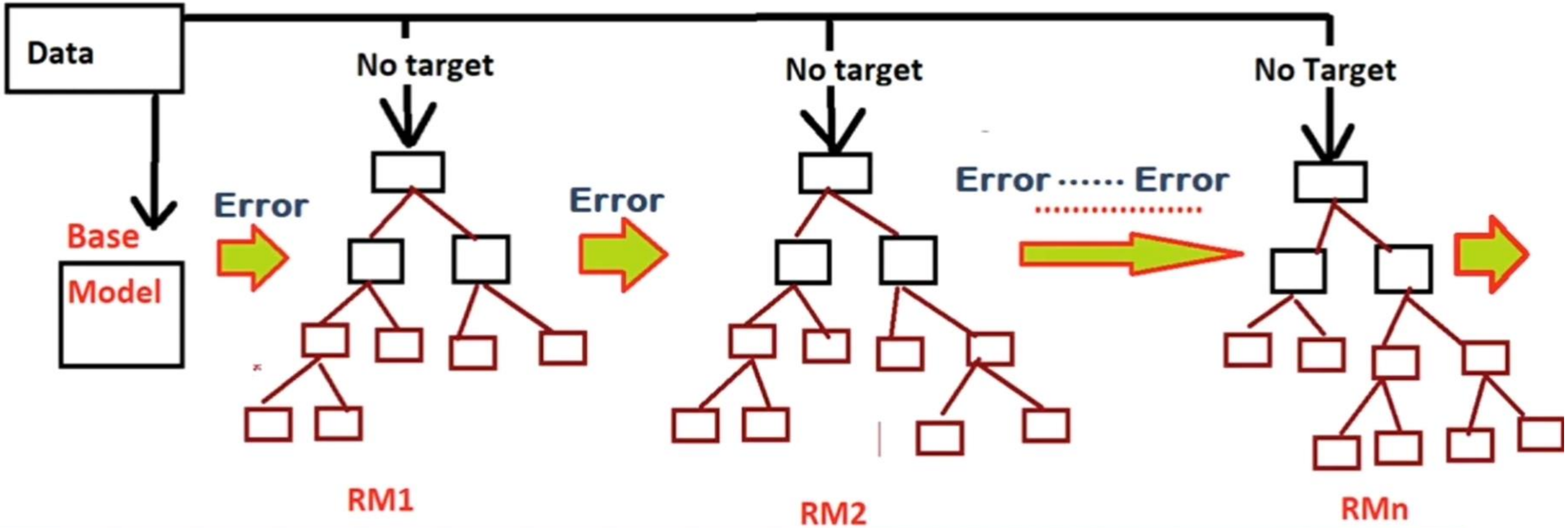
There are a number of excellent reasons:

- **Gradient boosting is the best**: its accuracy and performance are unmatched for tabular supervised learning tasks.
- **Gradient boosting is highly versatile**: it can be used in many important tasks such as regression, classification, ranking, and survival analysis.
- **Gradient boosting is interpretable**: unlike black-box algorithms like neural networks, gradient boosting does not sacrifice interpretability for performance. It works like a Swiss watch and yet, with patience, you can teach how it works to a school kid.
- **Gradient boosting is well-implemented**: it is not one of those algorithms that have little practical value. Various gradient boosting libraries like XGBoost and LightGBM in Python are used by hundreds of thousands of people.
- **Gradient boosting wins**: since 2015, professionals have used it to consistently win tabular competitions on platforms like Kaggle.

Real-World Applications of Gradient Boosting

- Gradient boosting has become such a dominant force in machine learning that its applications now span various industries, from predicting customer churn to detecting asteroids. Here's a glimpse into its success stories in Kaggle and real-world use cases:
- Dominating Kaggle competitions:
- **Otto Group Product Classification Challenge**: all top 10 positions used XGBoost implementation of gradient boosting.
- **Santander Customer Transaction Prediction**: XGBoost-based solutions again secured the top spots for predicting customer behavior and financial transactions.
- **Netflix Movie Recommendation Challenge**: Gradient boosting played a crucial role in building recommendation systems for multi-billion companies like Netflix.
- Transforming business and industry:
    - **Retail and e-commerce**: personalized recommendations, inventory management, fraud detection
    - **Finance and insurance**: credit risk assessment, churn prediction, algorithmic trading
    - **Healthcare and medicine**: disease diagnosis, drug discovery, personalized medicine
    - **Search and online advertising**: search ranking, ad targeting, click-through rate prediction

Gradient Boosting



Final Output = o/p of Base model $+\eta$RM1 $+\eta$RM2 $+\eta$RM3 + ...$+\eta$RMn

Steps in Gradient Boosting

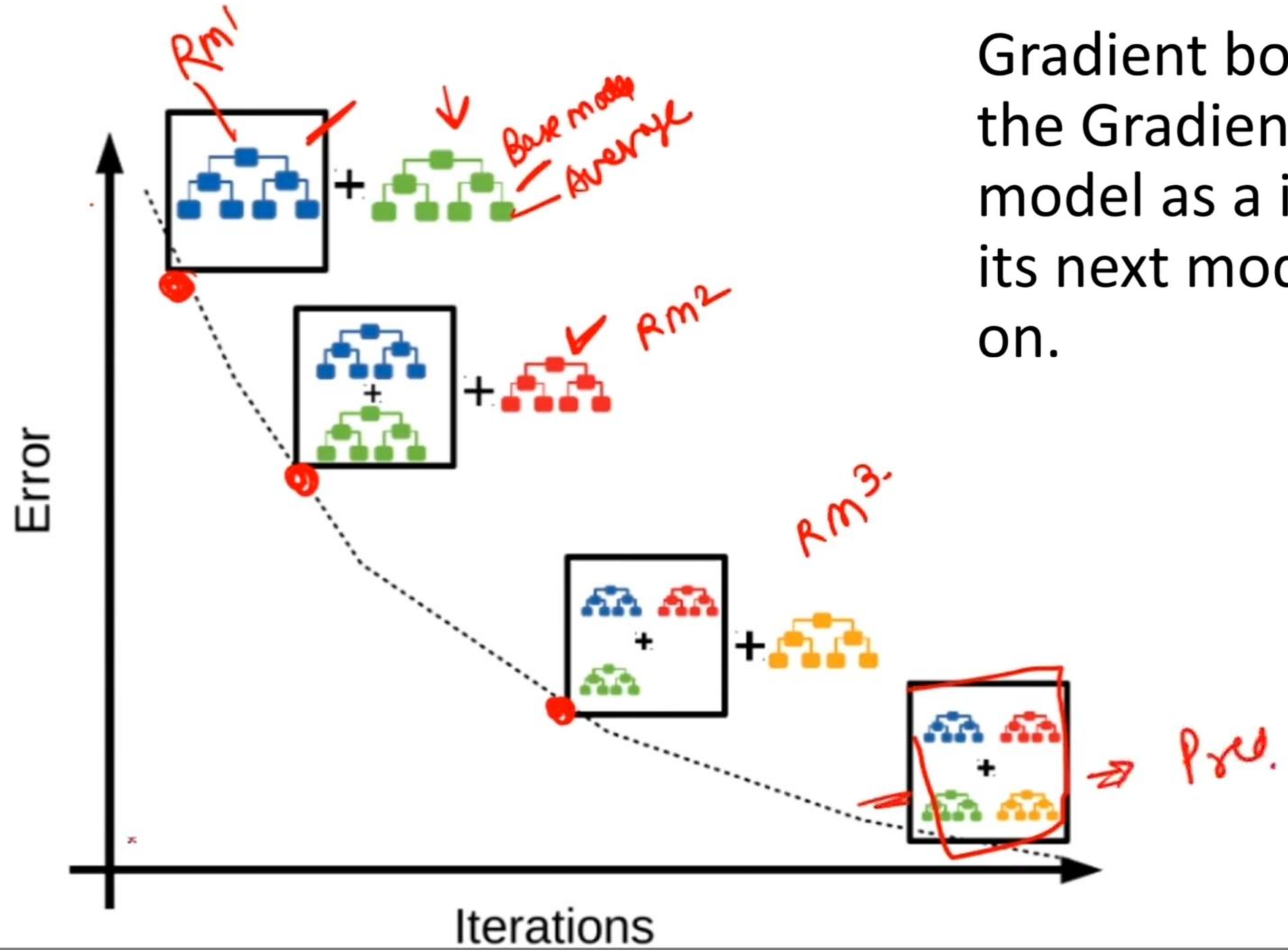
1) We will create a base model , **Average** model or most frequent category.

2) Calculate the **Residuals** from average prediction and actual values.

3) Now create another model **RM1** which will take residuals as target.

4) We have **New** predicted **residual** value, now we will calculate new Predicted target value.

5) Now we have residuals again (actual - Predicted) and New Model **RM 2** will fit again on the residues as target and will predict new residues.

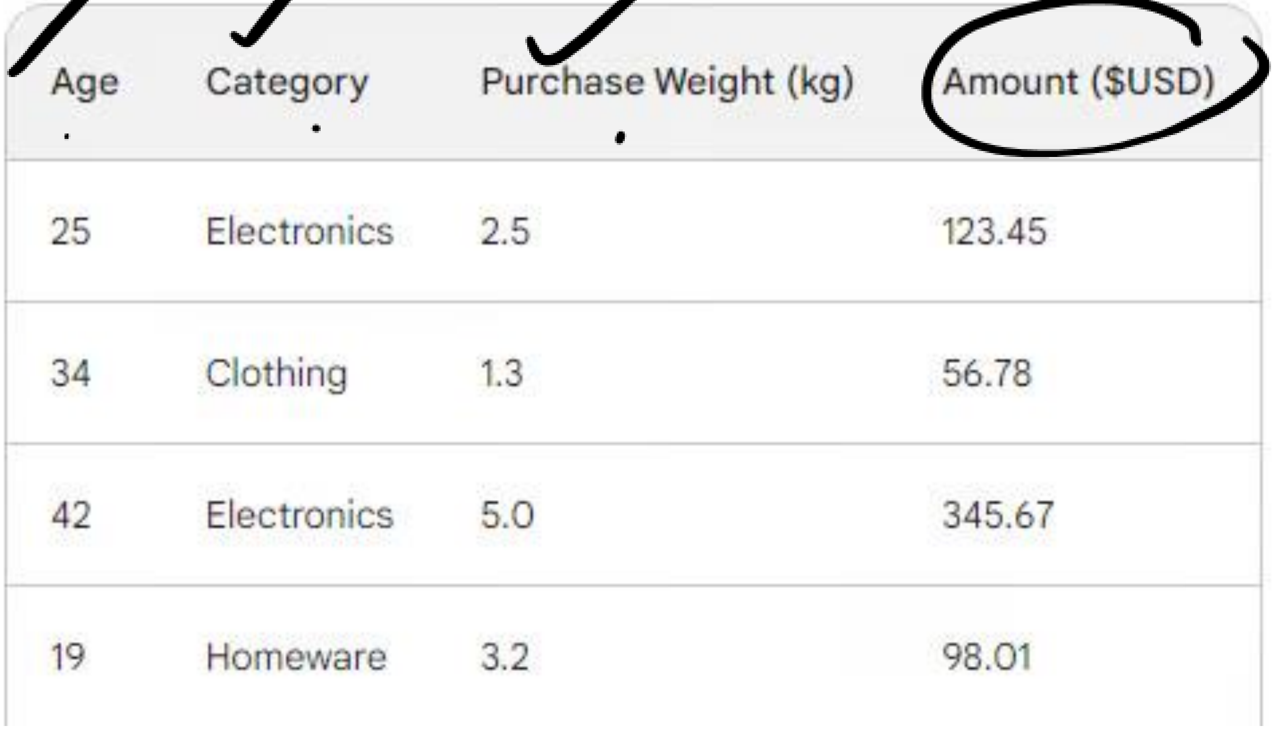


Gradient boosting uses the Gradient (loss) of model as a input to the its next model and it goes on.

Step by step algorithm with example

- **Input**
- Gradient boosting algorithm works for tabular data with a set of features (X) and a target (y). Like other machine learning algorithms, the aim is to learn enough from the training data to generalize well to unseen data points.
- To understand the underlying process of gradient boosting, we will use a simple sales dataset with four rows. Using three features — customer age, purchase category, and purchase weight, we want to predict the purchase amount:

Example data



Age	Category	Purchase Weight (kg)	Amount ($USD)
25	Electronics	2.5	123.45
34	Clothing	1.3	56.78
42	Electronics	5.0	345.67
19	Homeware	3.2	98.01

The loss function in gradient boosting

- In machine learning, a **loss function** is a critical component that lets us quantify the difference between a model's predictions and the actual values. In essence, it measures how a model is performing.

Here is a breakdown of its role:

- **Calculates the error:** Takes the predicted output of the model and compares it to the ground truth (actual observed values). How it compares, i.e., calculates the difference, varies from function to function.
- **Guides model training**: a model's objective is to minimize the loss function. Throughout training, the model continually updates its internal architecture and configuration to make the loss as little as possible.
- **Evaluation metric**: By comparing the loss on training, validation, and test datasets, you can assess your model's ability to generalize and avoid overfitting.

Two most common loss functions

$\boxed{\frac{1}{2}(\underline{y}-\underline{\hat{y}})^2}$

$\frac{dL}{d\hat{y}}$ $\frac{1}{2}\times 2(y-\hat{y})(-1)$

- **Mean Squared Error (MSE)**: This popular loss function for regression measures the sum of the squared differences between predicted and actual values. Gradient boosting often uses this variation of it:

$$\boxed{\frac{1}{2}(\underline{Observed} - \underline{Predicted})^2}$$

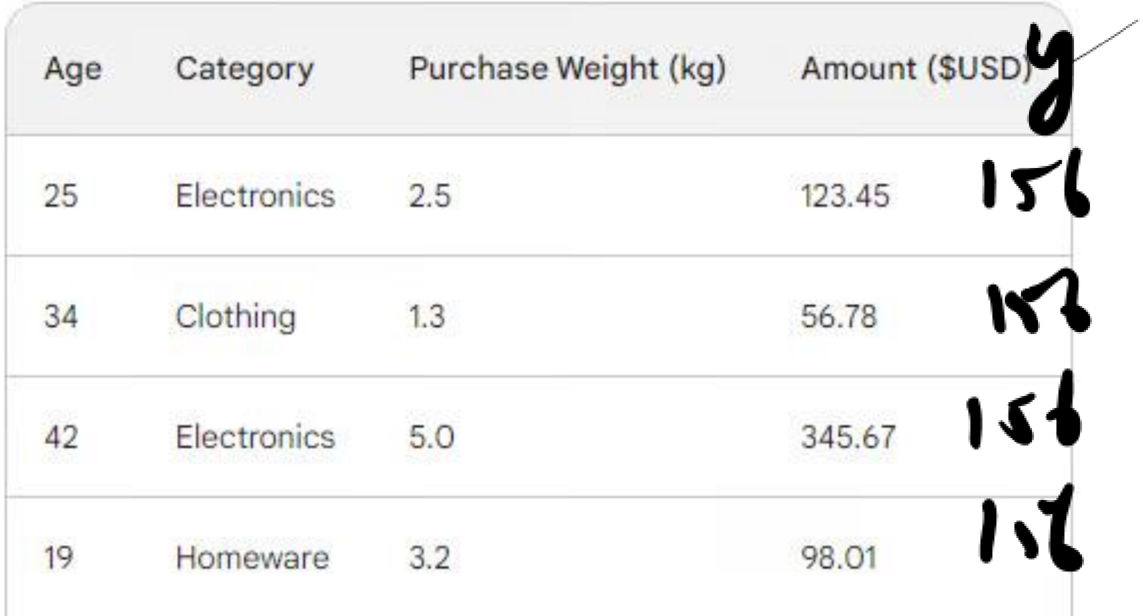
The reason the squared value is multiplied by one-half has got to do with differentiation. When we take the derivative of this loss function, one-half cancels out with the square because of the power rule. So, the final result would just be -(Observed - Predicted), making math much easier and less computationally expensive. **Therefore**, in Gradient Boosting for Regression, the final gradient (negative derivative of the loss) simplifies to $r_i = y_i - F(x_i)$

- **Cross-entropy:** This function measures the difference between two probability distributions. So, it is commonly used for classification tasks where the targets has discrete categories.

Step 1: Make an initial prediction

- Gradient boosting is an algorithm that gradually increases its accuracy. To start the process, we need an initial guess or prediction. The initial guess is always the average of the target. In other words, for the first round, our model predicts that all purchases were the same—156 dollars:
- So, our initial prediction is the average—156 dollars. Hold it in memory as we continue.

$$(123.45 + 56.78 + 345.67 + 98.01) / 4 = 156$$



Age	Category	Purchase Weight (kg)	Amount ($USD)	$\hat{y}$
25	Electronics	2.5	123.45	156
34	Clothing	1.3	56.78	156
42	Electronics	5.0	345.67	156
19	Homeware	3.2	98.01	156

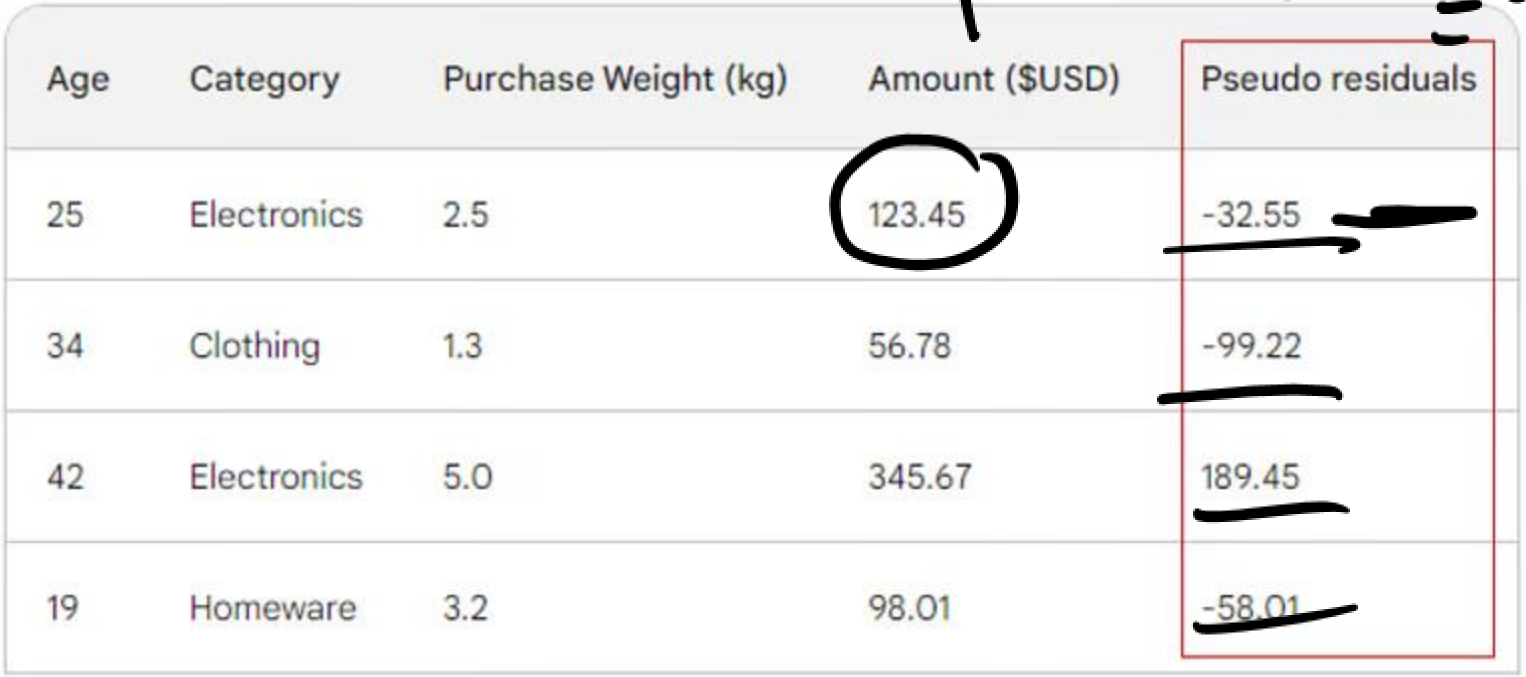
Step 2: Calculate the pseudo-residuals

- The next step is to find the differences between each observed value and our initial prediction: Observed-156. Remember that in linear regression, the difference between observed values and predicted values is called residuals.
- To differentiate linear regression and gradient boosting, we call them pseudo-residuals.
- For illustration, we will put those differences in a new column:

Step 2: Calculate the pseudo-residuals

$y - \hat{y}$

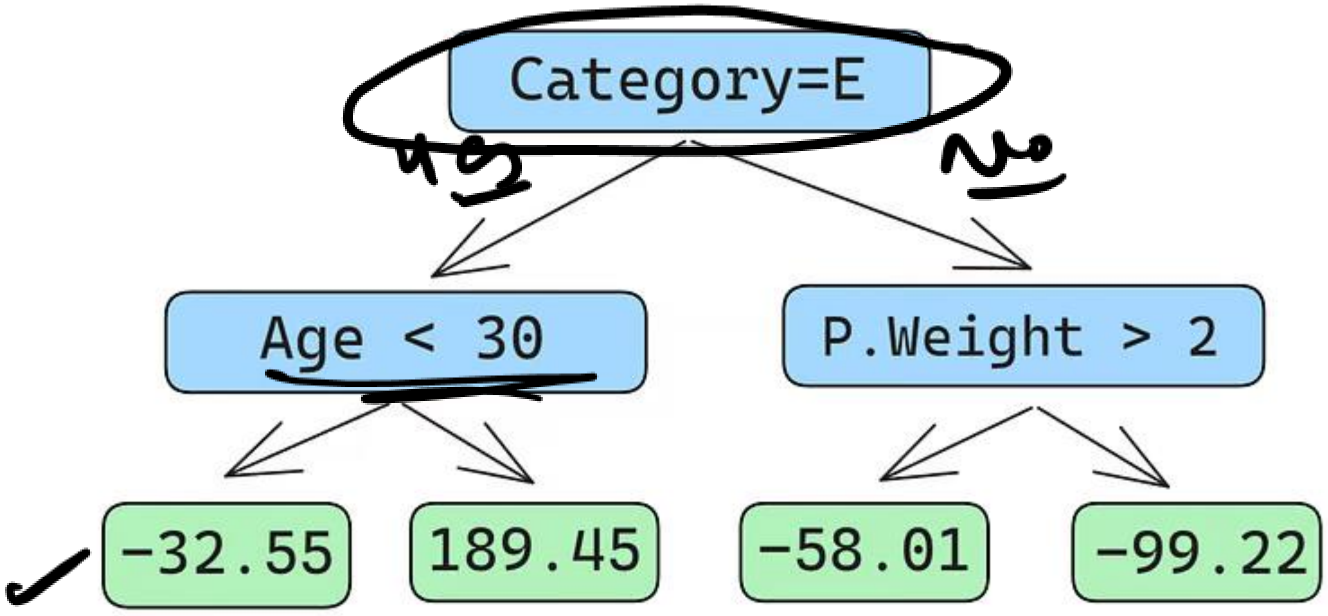
$= 123.45 - 156$

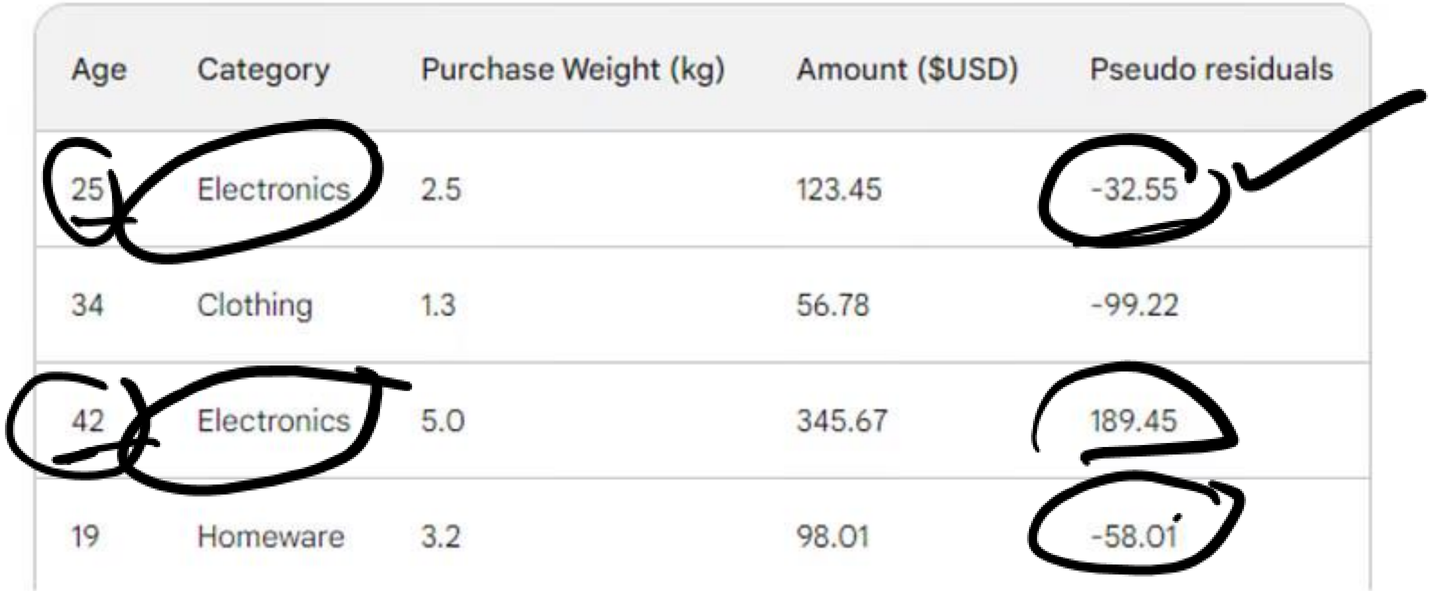


Age	Category	Purchase Weight (kg)	Amount ($USD)	Pseudo residuals
25	Electronics	2.5	123.45	-32.55
34	Clothing	1.3	56.78	-99.22
42	Electronics	5.0	345.67	189.45
19	Homeware	3.2	98.01	-58.01

Step 3: Build a weak learner

- Next, we will build a decision tree (weak learner) that predicts the residuals using the three features we have (age, category, purchase weight). For this problem, we will limit the decision tree to just four leaves (terminal nodes), but in practice, people usually choose leaves between 8 and 32.

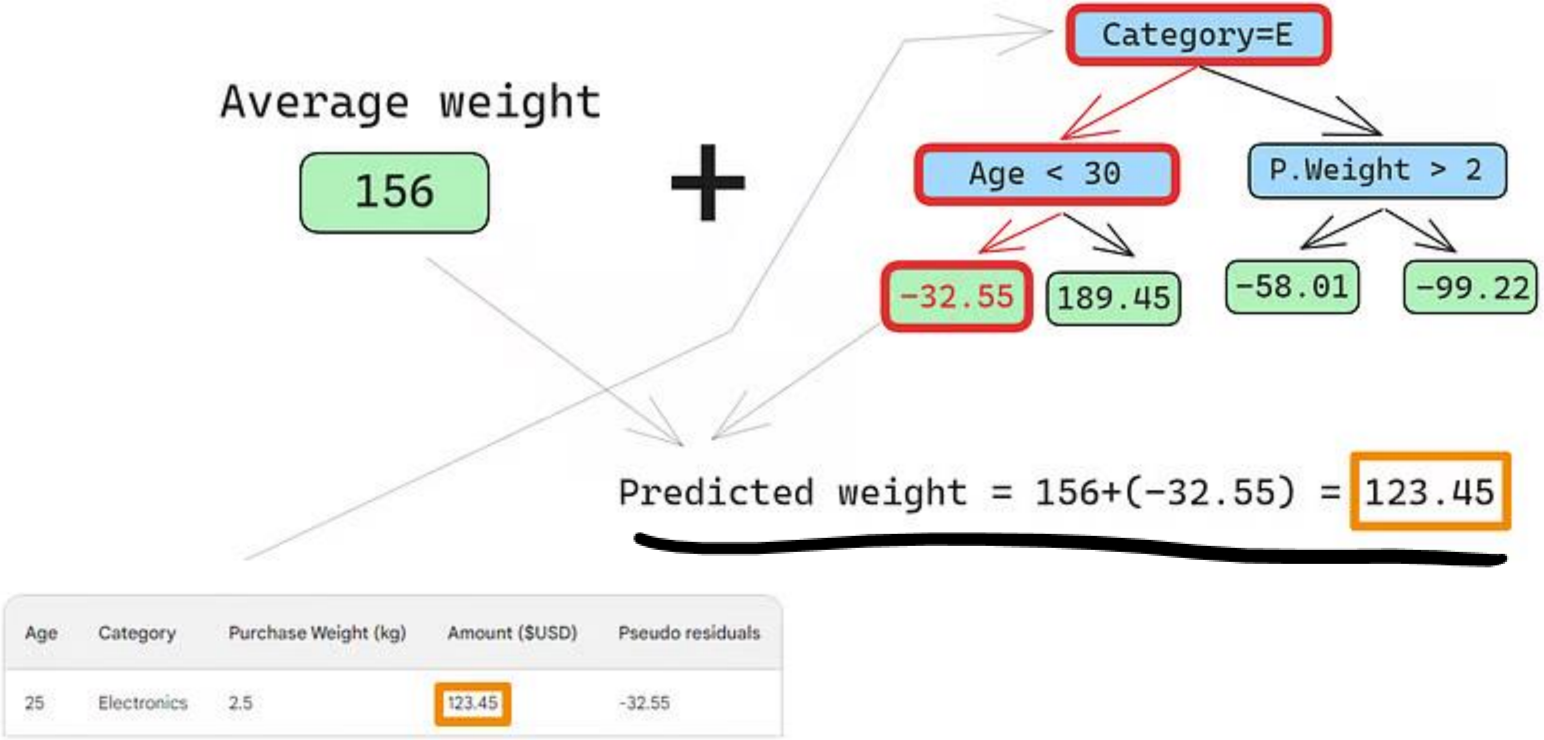




Age	Category	Purchase Weight (kg)	Amount ($USD)	Pseudo residuals
25	Electronics	2.5	123.45	-32.55
34	Clothing	1.3	56.78	-99.22
42	Electronics	5.0	345.67	189.45
19	Homeware	3.2	98.01	-58.01

Making predictions

- After the tree is fit to the data, we make a prediction for each row in the data. Here is how to do the first one: ***“Predicted purchase amount is denoted by “Predicted weight”***



Predicted weight = 156+(-32.55) = 123.45

Age	Category	Purchase Weight (kg)	Amount ($USD)	Pseudo residuals
25	Electronics	2.5	123.45	-32.55

Role of learning parameter

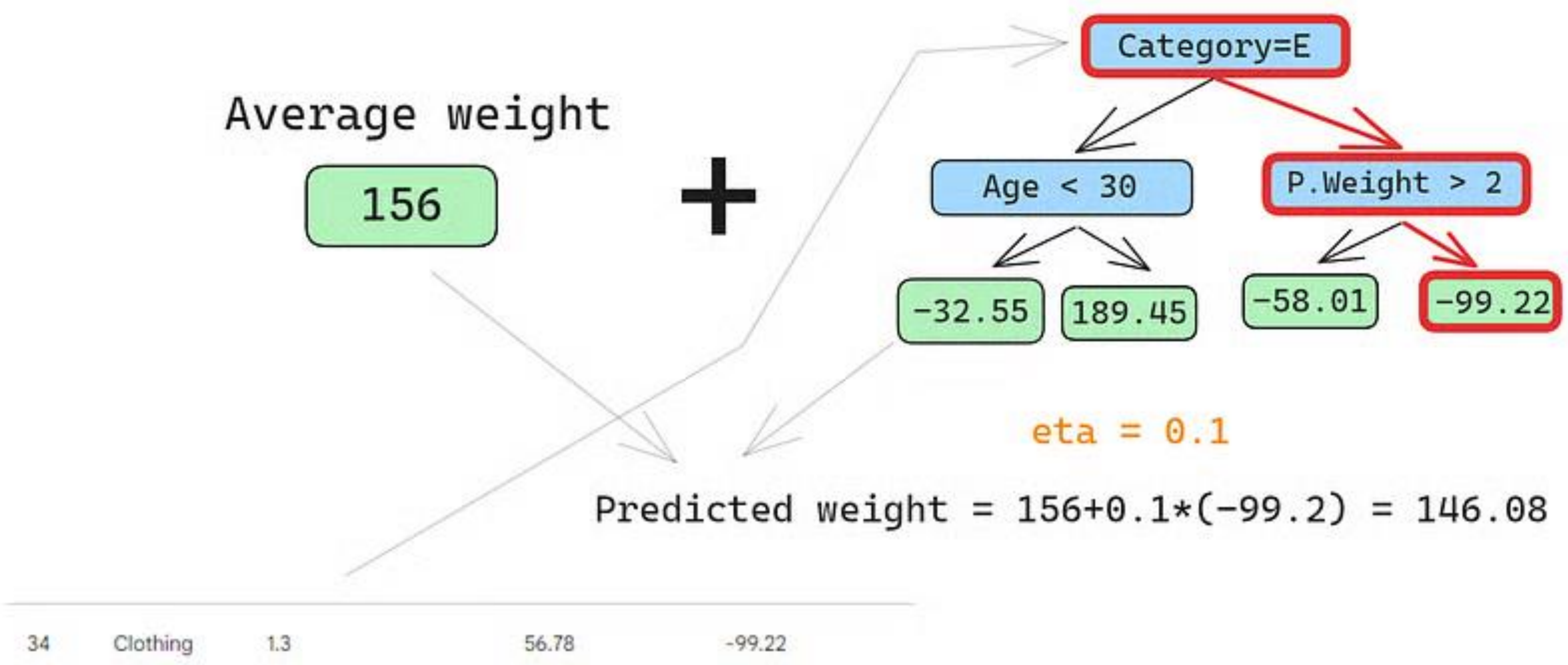
- The first row has the following features: a category of electronics (the left of the root node) and customer age below 30 (the left of the child node). This puts -32.55 into the leaf node. To make the final prediction, we add -32.55 to our first prediction, which is exactly the same as the observed value—123.45 dollars!
- We just made a perfect prediction, so why bother with building other trees? Well, right now, we are heavily overfitting to the training data. We want the model to generalize. So, to mitigate this problem, gradient boosting has a parameter called learning rate.
- The learning rate in gradient boosting is simply a multiplier between 0 and 1 that scales the prediction of each weak learner (see the section below for details on learning rate). When we add an arbitrary learning rate of 0.1 into the mix, our prediction becomes 152.75, not the perfect 123.45.

$y_{old} + \alpha \times$ Residual error

eta = 0.1

Predicted weight = 156+0.1*(−32.55) = 152.75

Let’s predict on the second row as well:



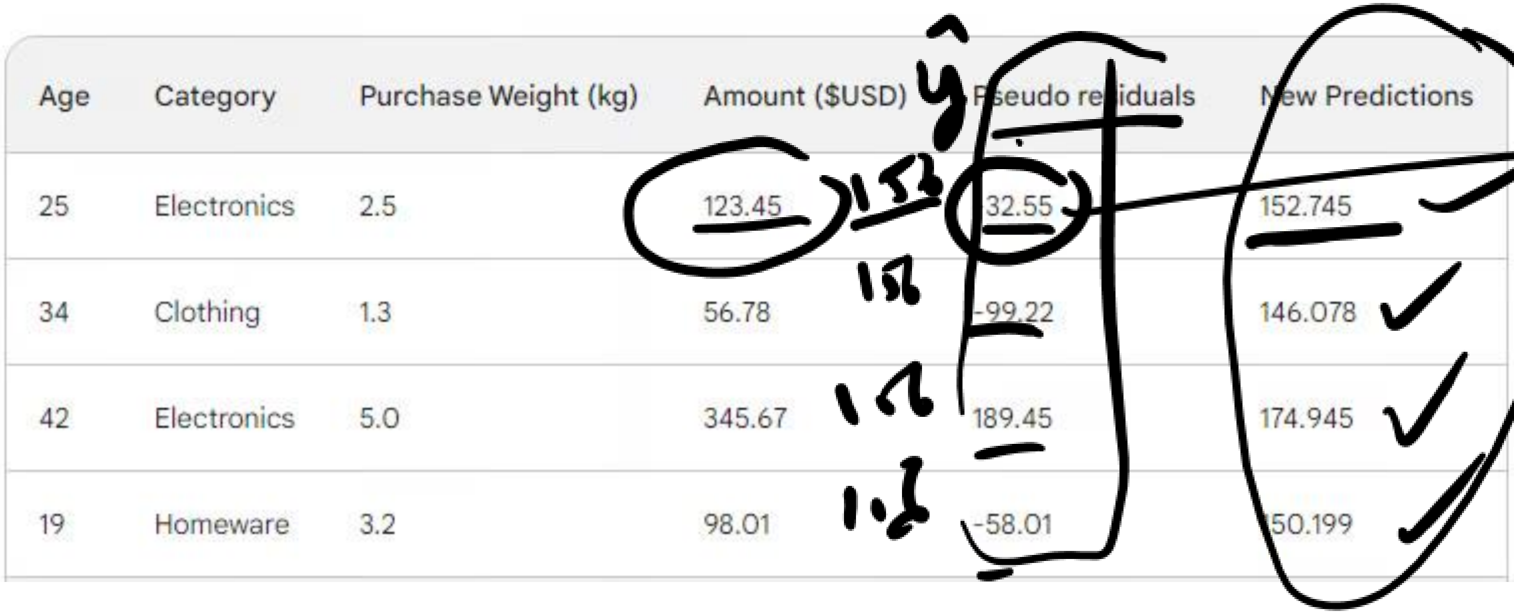
Predicted weight = 156+0.1*(−99.2) = 146.08

34	Clothing	1.3	56.78	-99.22

Iterating over the other rows

$\hat{y}$ = base + α × error
= 156 + 0.1 * (-32.55

- We run the row through the tree and get 146.08 as a prediction. We continue in this fashion for all rows until we have four predictions for four rows: 152.75, 146.08, 174.945, 150.2. Let's add them as a new column for now:
-



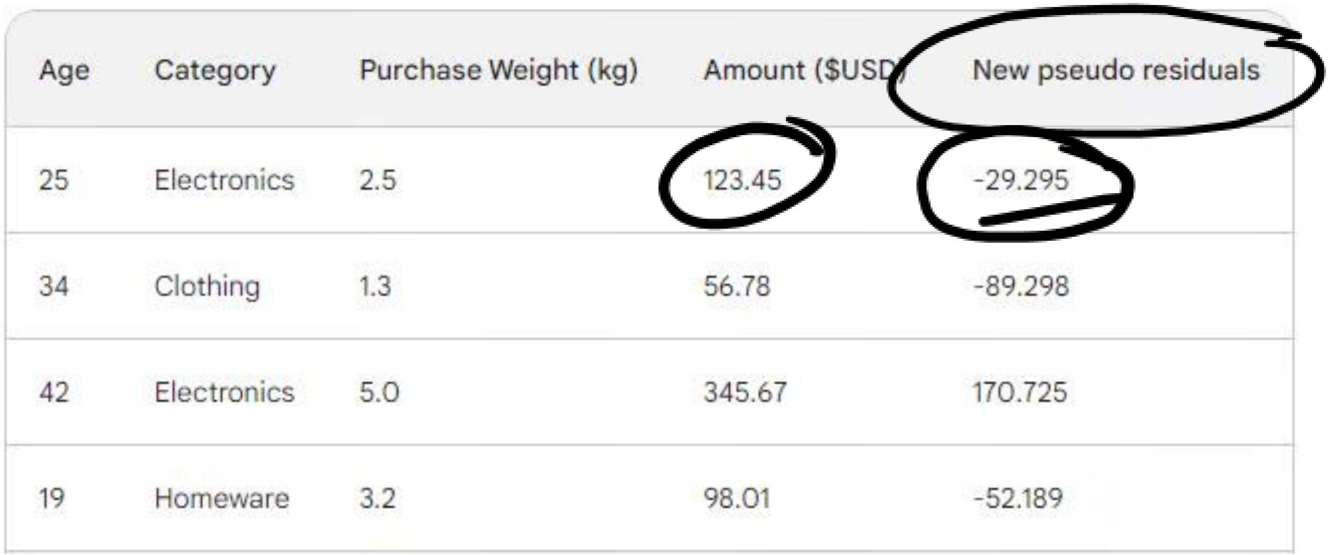
Age	Category	Purchase Weight (kg)	Amount ($USD)	$\hat{y}$	Pseudo residuals	New Predictions
25	Electronics	2.5	123.45	153	32.55	152.745
34	Clothing	1.3	56.78	156	-99.22	146.078 ✓
42	Electronics	5.0	345.67	156	189.45	174.945 ✓
19	Homeware	3.2	98.01	1.2	-58.01	150.199 ✓

-32.55

Calculating new residuals

$y_{pred} = \text{BaseModel} + \alpha RM_1 + \alpha RM_2$

- Next, we find the new pseudo-residuals by subtracting new predictions from the purchase amount. Let’s add them as a new column to the table and drop the last two:



Age	Category	Purchase Weight (kg)	Amount ($USD)	New pseudo residuals
25	Electronics	2.5	123.45	-29.295
34	Clothing	1.3	56.78	-89.298
42	Electronics	5.0	345.67	170.725
19	Homeware	3.2	98.01	-52.189

As you can see, our new pseudo residuals are smaller, which means our loss is going down.

Step 4: Iterate

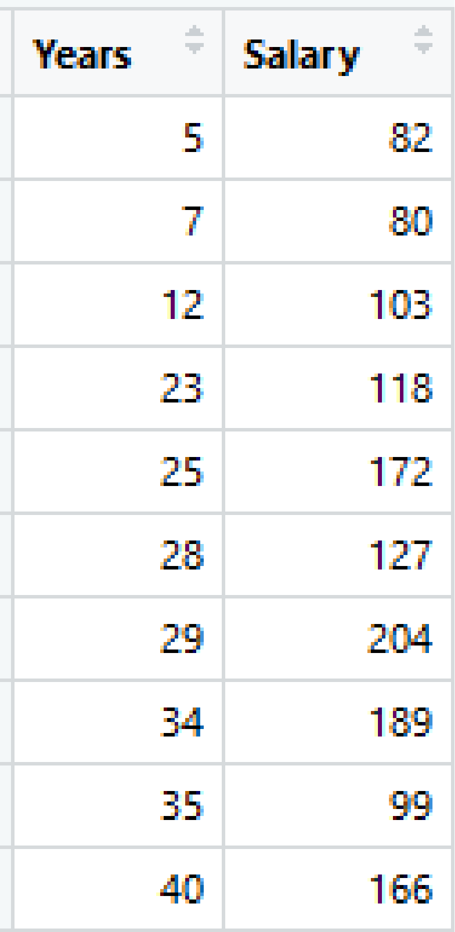
- In the next steps, we iterate on step 3, i.e. build more weak learners. The only thing to remember is that we have to keep adding the residuals of each tree to the initial prediction to generate the next.
- For example, if we build 10 trees and the residuals of each tree are denoted as r_i (1 <= i <= 10), the next prediction would become p_10 = 156 + eta * (r_1 + r_2 + ... + r_10) where p_10 denotes prediction in the tenth round.
- In practice, professionals often start with 100 trees, not just 10. In this case, the algorithm is said to train for 100 **boosting rounds**.

Early Stopping

- If you don’t know the exact number of trees you need for your specific problem, you can use a simple technique called early stopping.
- In **early stopping**, we choose a large number of trees, like 1000 or 10000. Then, instead of waiting for the algorithm to finish building all those trees, we monitor the loss. If the loss doesn’t improve for a certain number of boosting rounds, for example, 50, we stop training earlier, saving time and computation resources.

Another Example

- Consider the following data where the years of experience is predictor variable and salary (in thousand dollars) is the target. Using regression trees as base learners, we can create an **ensemble model** to predict the salary. For the sake of simplicity, we can choose square loss as our loss function and our objective would be to minimize the square error. Assume Learning rate=1.



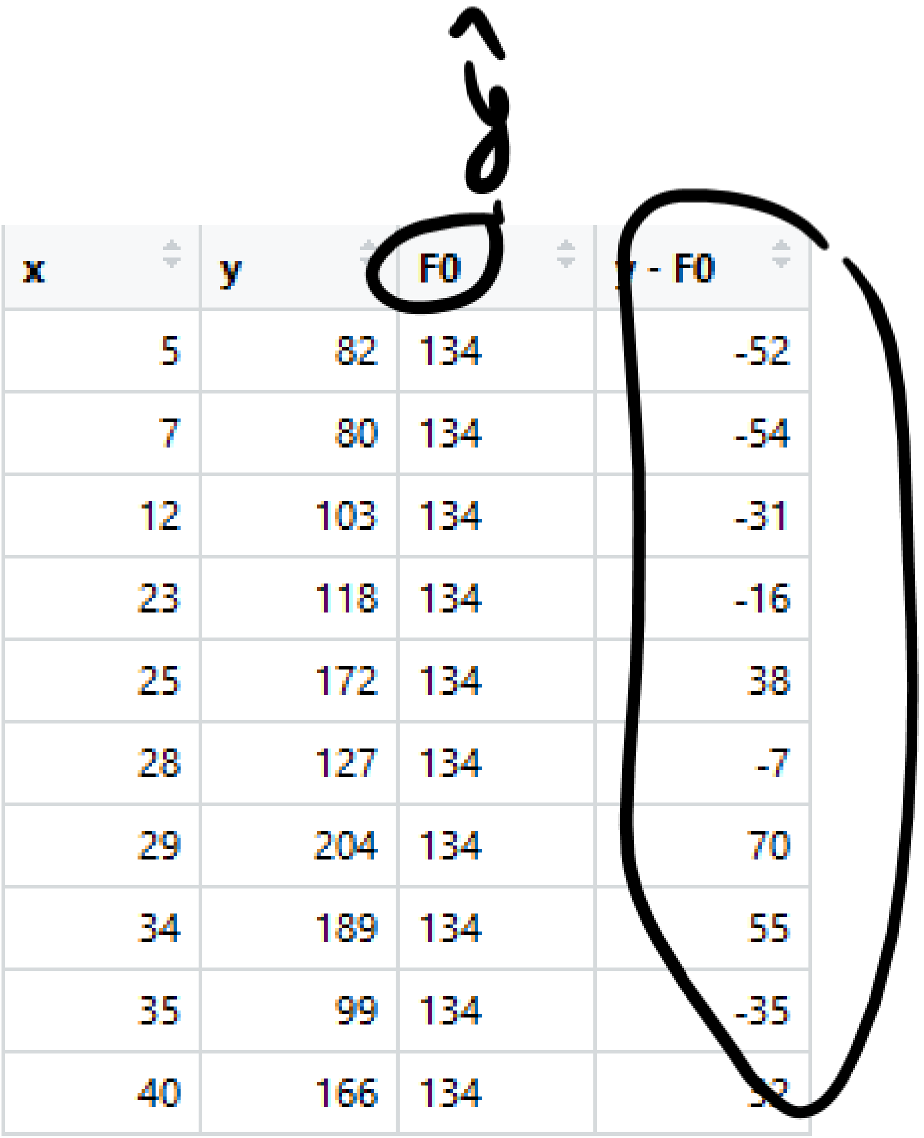
Years	Salary
5	82
7	80
12	103
23	118
25	172
28	127
29	204
34	189
35	99
40	166

Steps

- F0(x) gives the predictions from the first stage of our model. Now, the residual error for each instance is (yi – F0(x)).

$$F_0(x) = \frac{\sum_{i=1}^{n} y_i}{n}$$

Steps



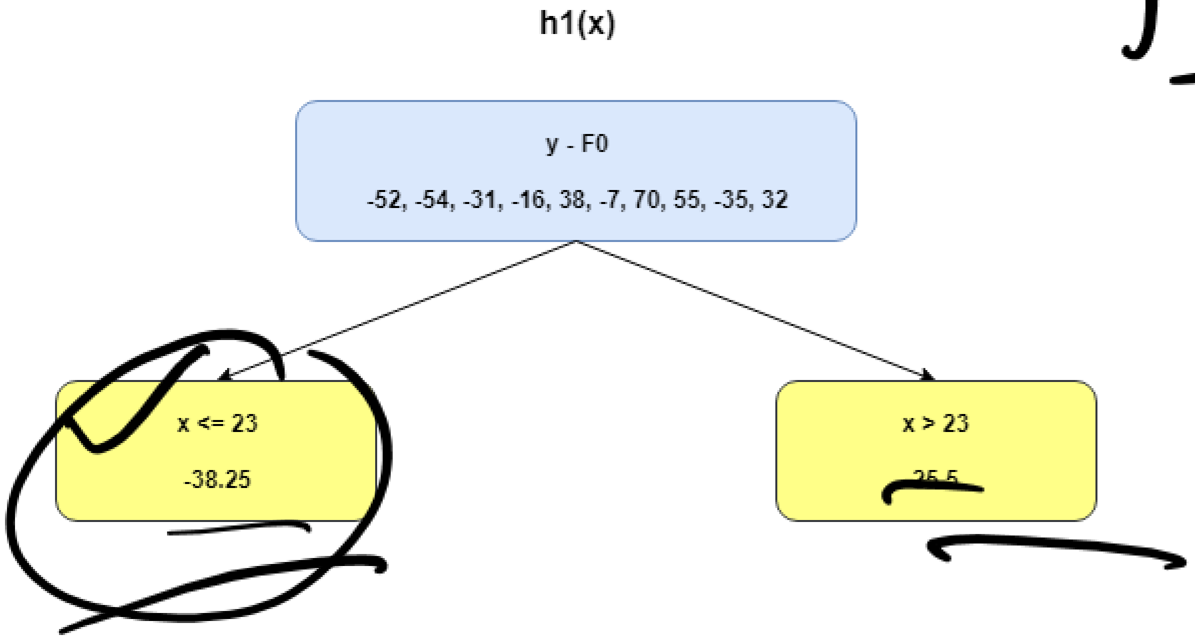
$\hat{y}$

x	y	F0	y - F0
5	82	134	-52
7	80	134	-54
12	103	134	-31
23	118	134	-16
25	172	134	38
28	127	134	-7
29	204	134	70
34	189	134	55
35	99	134	-35
40	166	134	32

Building Additive Learners

- We can use the residuals from F0(x) to create h1(x). h1(x) will be a regression tree which will try and reduce the residuals from the previous step. The output of h1(x) won’t be a prediction of y; instead, it will help in predicting the successive function F1(x) which will bring down the residuals.

$\boxed{x<=23}$



h1(x)

y - F0

-52, -54, -31, -16, 38, -7, 70, 55, -35, 32

x <= 23

-38.25

x > 23

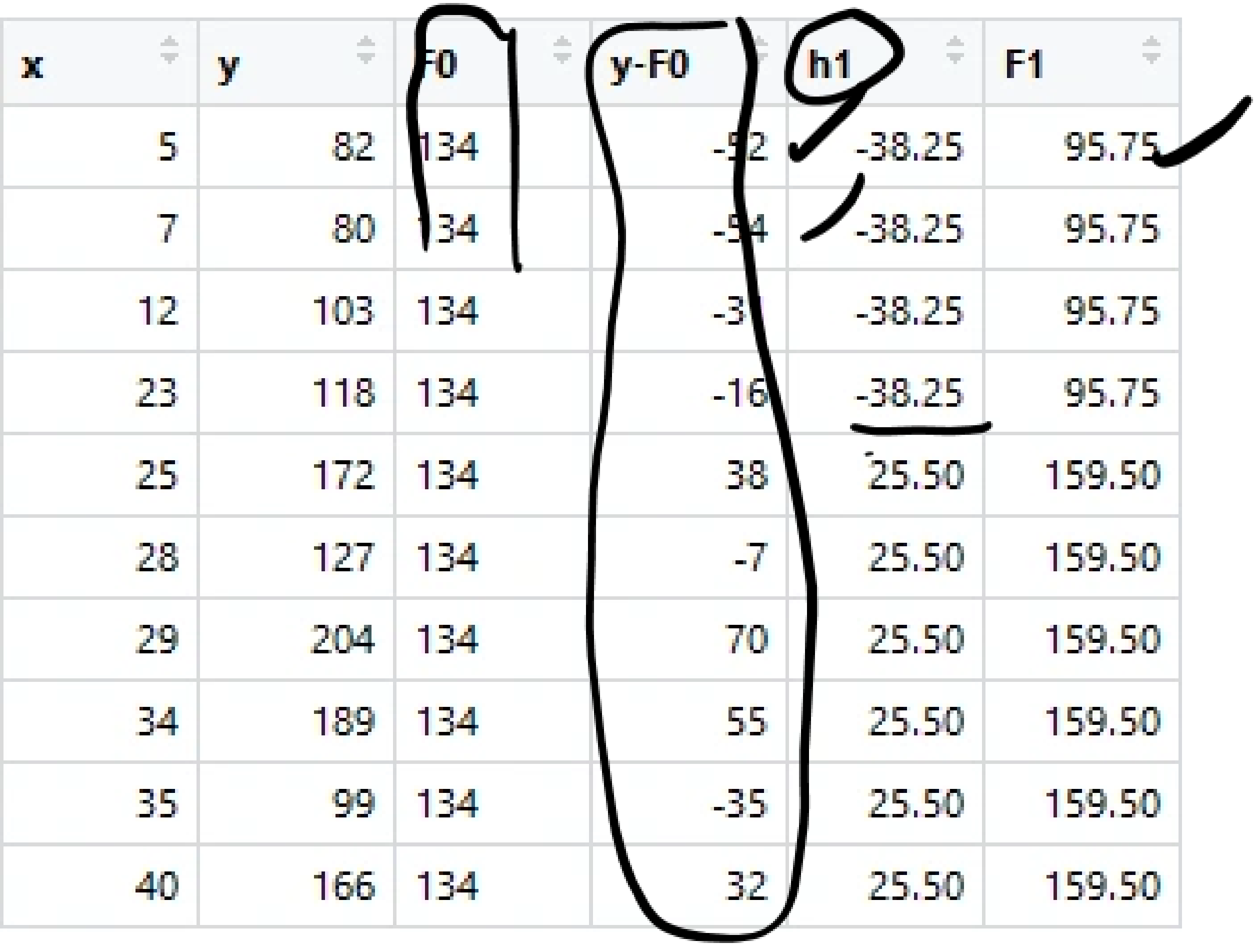
25.5

$$y_{pred} = \underline{\text{Base Model}} + \alpha\, \underline{\text{Rm}}$$

$$= \underline{134 + (1) * (-38.2}$$

Steps

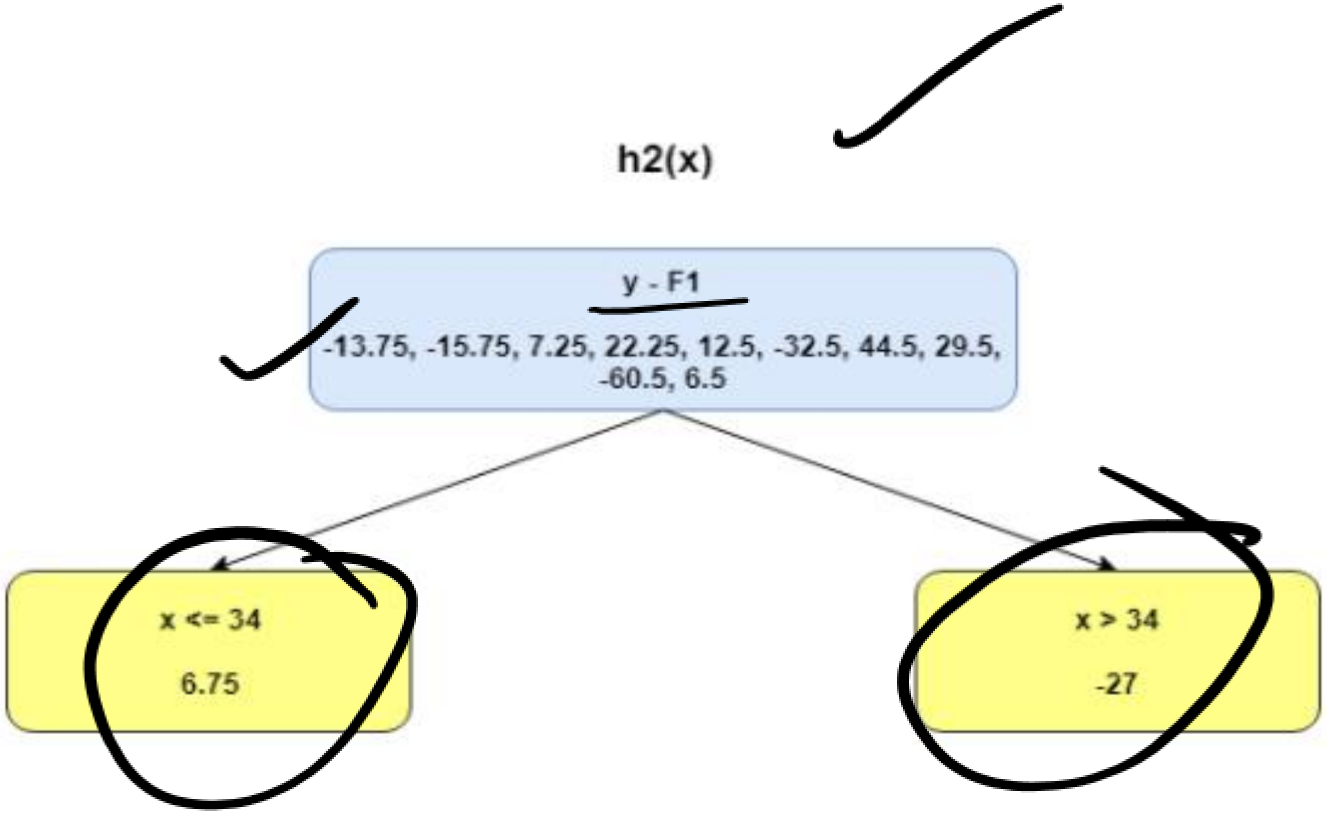
- The additive model h1(x) computes the mean of the residuals (y – F0) at each leaf of the tree. The boosted function F1(x) is obtained by summing F0(x) and h1(x). This way h1(x) learns from the residuals of F0(x) and suppresses it in F1(x).



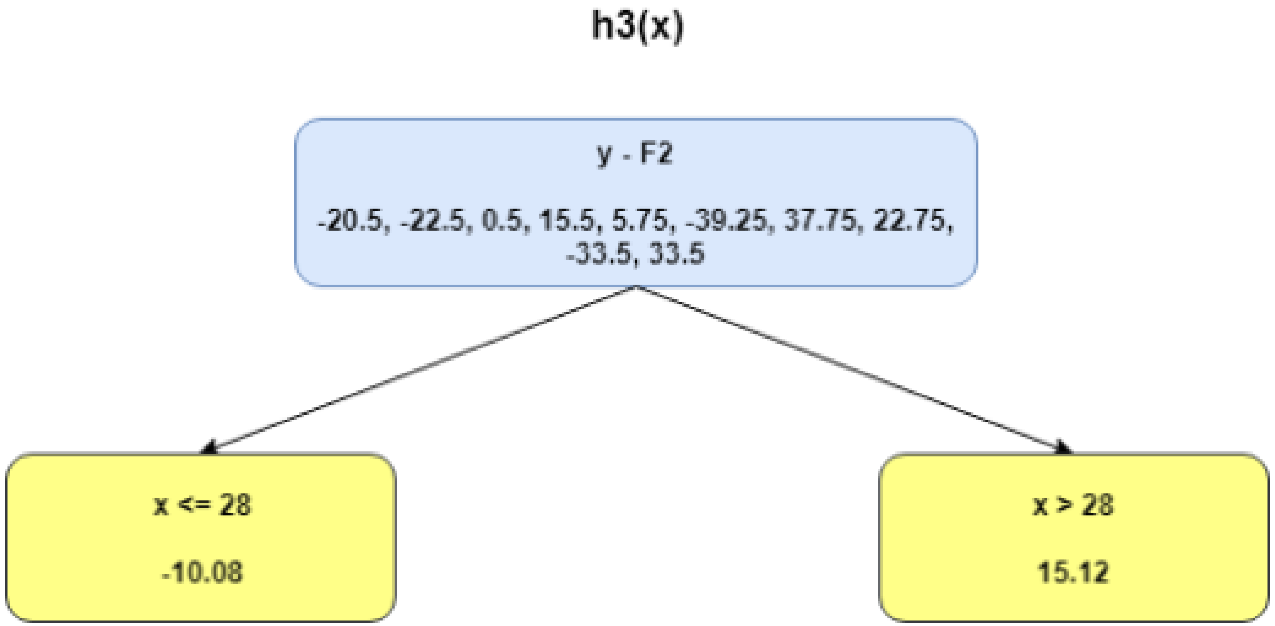
x	y	F0	y-F0	h1	F1
5	82	134	-52	-38.25	95.75
7	80	134	-54	-38.25	95.75
12	103	134	-31	-38.25	95.75
23	118	134	-16	-38.25	95.75
25	172	134	38	25.50	159.50
28	127	134	-7	25.50	159.50
29	204	134	70	25.50	159.50
34	189	134	55	25.50	159.50
35	99	134	-35	25.50	159.50
40	166	134	32	25.50	159.50

Steps

- This can be repeated for 2 more iterations to compute h2(x) and h3(x).

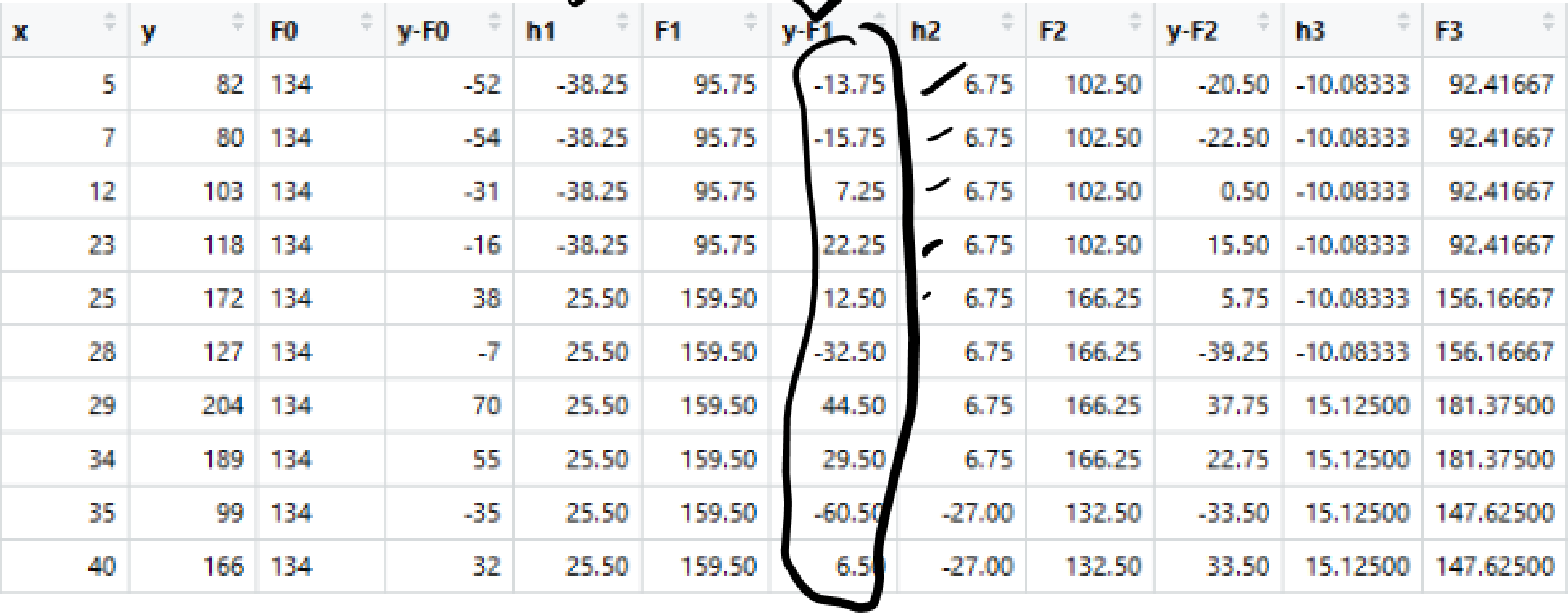


Steps


h3(x)
y - F2
-20.5, -22.5, 0.5, 15.5, 5.75, -39.25, 37.75, 22.75, -33.5, 33.5
x <= 28
-10.08
x > 28
15.12

F2 = 134 + α * h1 + α h2

Steps



x	y	F0	y-F0	h1	F1	y-F1	h2	F2	y-F2	h3	F3
5	82	134	-52	-38.25	95.75	-13.75	6.75	102.50	-20.50	-10.08333	92.41667
7	80	134	-54	-38.25	95.75	-15.75	6.75	102.50	-22.50	-10.08333	92.41667
12	103	134	-31	-38.25	95.75	7.25	6.75	102.50	0.50	-10.08333	92.41667
23	118	134	-16	-38.25	95.75	22.25	6.75	102.50	15.50	-10.08333	92.41667
25	172	134	38	25.50	159.50	12.50	6.75	166.25	5.75	-10.08333	156.16667
28	127	134	-7	25.50	159.50	-32.50	6.75	166.25	-39.25	-10.08333	156.16667
29	204	134	70	25.50	159.50	44.50	6.75	166.25	37.75	15.12500	181.37500
34	189	134	55	25.50	159.50	29.50	6.75	166.25	22.75	15.12500	181.37500
35	99	134	-35	25.50	159.50	-60.50	-27.00	132.50	-33.50	15.12500	147.62500
40	166	134	32	25.50	159.50	6.50	-27.00	132.50	33.50	15.12500	147.62500

The MSEs for F1(x), F1(x) and F2(x) are 875, 692 and 540. It's amazing how these simple weak learners can bring about a huge reduction in error!

Configuring Gradient Boosting Models

- In machine learning, choosing the settings for a model is known as "hyperparameter tuning." These settings, called "hyperparameters," are options that the machine learning engineer must choose themselves. Unlike other parameters, the model cannot learn the best values for hyperparameters simply by being trained on data.
- Gradient boosting models have many hyperparameters, some of which I will outline below.

Objective

- This parameter sets the direction and the loss function of the algorithm. If the objective is regression, MSE is chosen as a loss function, whereas for classification, Cross-Entropy is the one to go. Python libraries like XGBoost offer other objectives for other types of tasks, such as ranking with corresponding loss functions.

Configuring Gradient Boosting Models

- **Learning rate:** The most important hyperparameter of gradient boosting is perhaps the learning rate. It controls the contribution of each weak learner by adjusting the shrinkage factor. Smaller values (towards 0) decreases how much say each weak learner has in the ensemble. This requires building more trees and, thus, more time to finish training. But, the final strong learner will indeed be strong and impervious to overfitting.
- **The number of trees:** This parameter, also called the number of boosting rounds or n_es timators, controls the number of trees to build. The more trees you build, the stronger and more performant the ensemble becomes. It also becomes more complex as more trees allows the model to capture more patterns in the data. However, more trees significantly improve the chances of overfitting. To mitigate this, employ a combination of early stopping and low learning rate.
- **Max depth:** This parameter controls the number of levels in each weak learner (decision tree). A max depth of 3 means there are three levels in the tree, counting the leaf level. The deeper the tree, the more complex and computationally expensive the model becomes. Choose a value close to 3 to prevent overfitting. Your maximum should be a depth of 10.

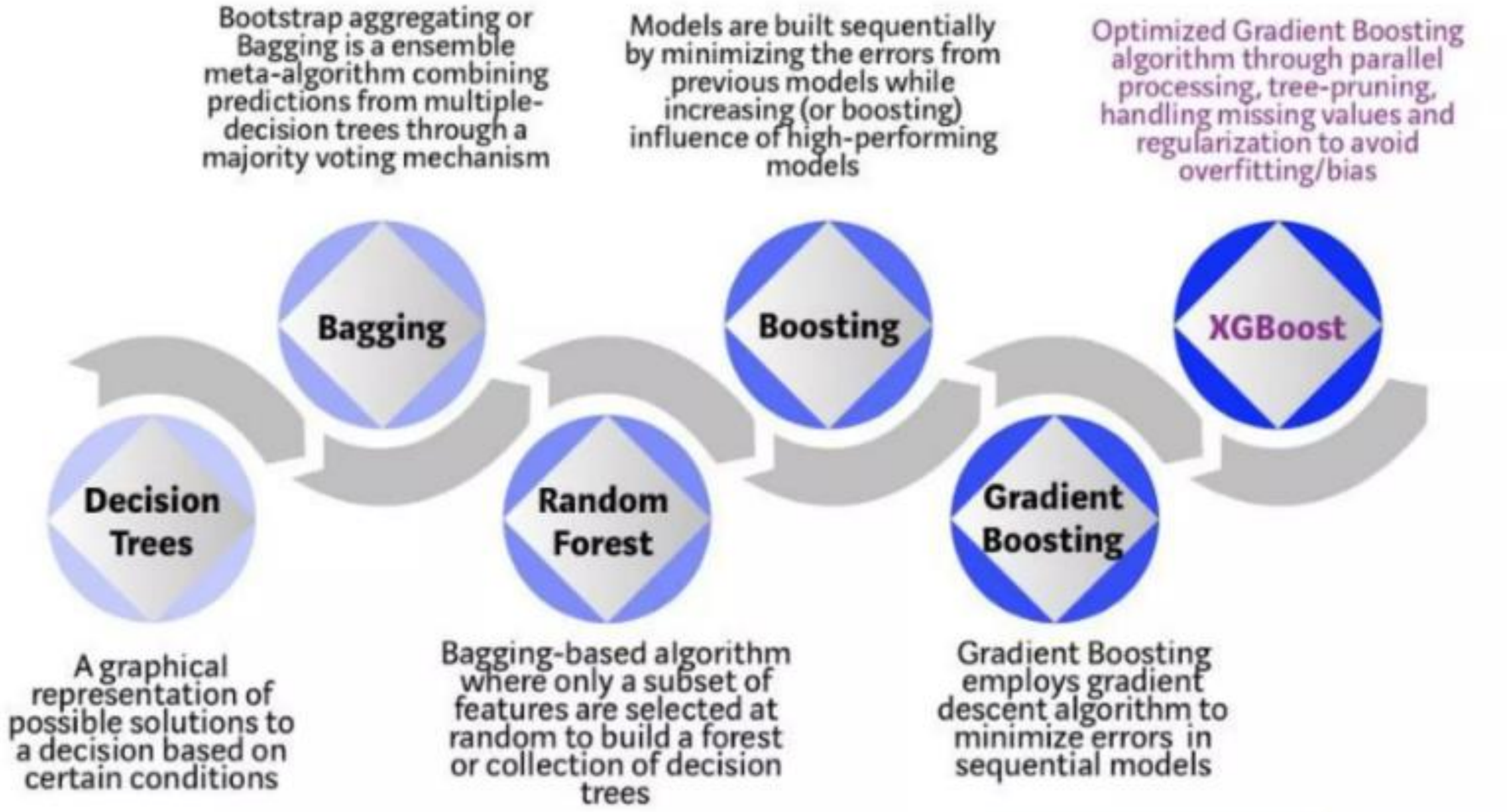
Configuring Gradient Boosting Models

- **Minimum number of samples per leaf:** This parameter controls how branches split in decision trees. Setting a low value for the number of samples in termination nodes (leaves) makes the overall algorithm sensitive to noise. A larger minimum number of samples helps to prevent overfitting by making it more difficult for the trees to create splits based on too few data points.
- **Subsampling rate:** This parameter controls the proportion of the data used to train each tree. In the examples above, we used 100% of rows as there were only four rows in our dataset. But, real-world datasets often have much more and require sampling. So, if you set the subsampling rate to a value below 1, such as 0.7, each weak learner trains on the randomly sampled 70% of the rows. A smaller subsample rate can lead to faster training but can also lead to overfitting.
- **Feature sampling rate:** This parameter is exactly like subsampling, but it samples rows. For datasets with hundreds of features, it is recommended to choose a feature sampling rate between 0.5 and 1 to lower the chance of overfitting.

Difference between Adaboost and Gradient Boosting

Features	AdaBoost	Gradient Boosting
Weight Update Strategy	Increase weights of misclassified sample so that the next learner focuses more on them.	Updates predictions by minimizing a loss function using the negative gradient
Base learners	AdaBoost uses simple decision trees with one split known as the decision stumps of weak learners.	Gradient Boosting can use a wide range of base learners such as decision trees and linear models.
Sensitivity to Noise	AdaBoost is more sensitive to noisy data and outliers due to aggressive weighting.	Gradient Boosting is less sensitive as it smooths updates using gradients.
Optimization Technique	No explicit loss function i.e it focuses on classification error.	Explicitly minimizes a differentiable loss function.
Boosting Mechanism	Learners are trained sequentially with sample reweighting.	Learners are trained sequentially with residual fitting (gradient descent).
Interpretability	Easier to interpret due to simple weak learners.	Harder to interpret if complex models are used.
Use case	Suitable for clean datasets with fewer outliers	Suitable for complex problems with varying loss function

Evolution of tree-based algorithms



Decision Trees

A graphical representation of possible solutions to a decision based on certain conditions

Bagging

Bootstrap aggregating or Bagging is a ensemble meta-algorithm combining predictions from multiple-decision trees through a majority voting mechanism

Random Forest

Bagging-based algorithm where only a subset of features are selected at random to build a forest or collection of decision trees

Boosting

Models are built sequentially by minimizing the errors from previous models while increasing (or boosting) influence of high-performing models

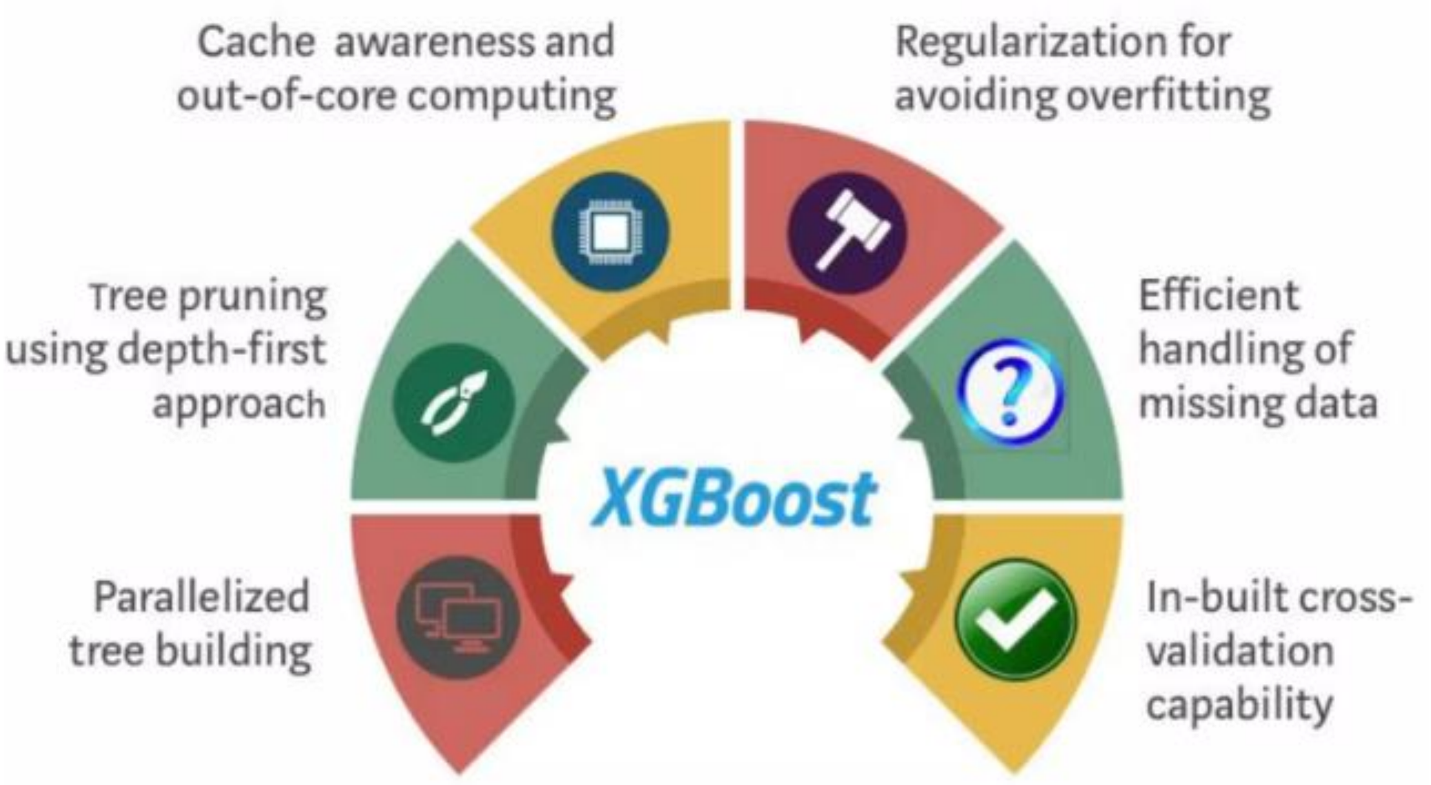
Gradient Boosting

Gradient Boosting employs gradient descent algorithm to minimize errors in sequential models

XGBoost

Optimized Gradient Boosting algorithm through parallel processing, tree-pruning, handling missing values and regularization to avoid overfitting/bias

Unique features of XGBoost


Cache awareness and out-of-core computing
Regularization for avoiding overfitting
Tree pruning using depth-first approach
Efficient handling of missing data
XGBoost
Parallelized tree building
In-built cross-validation capability

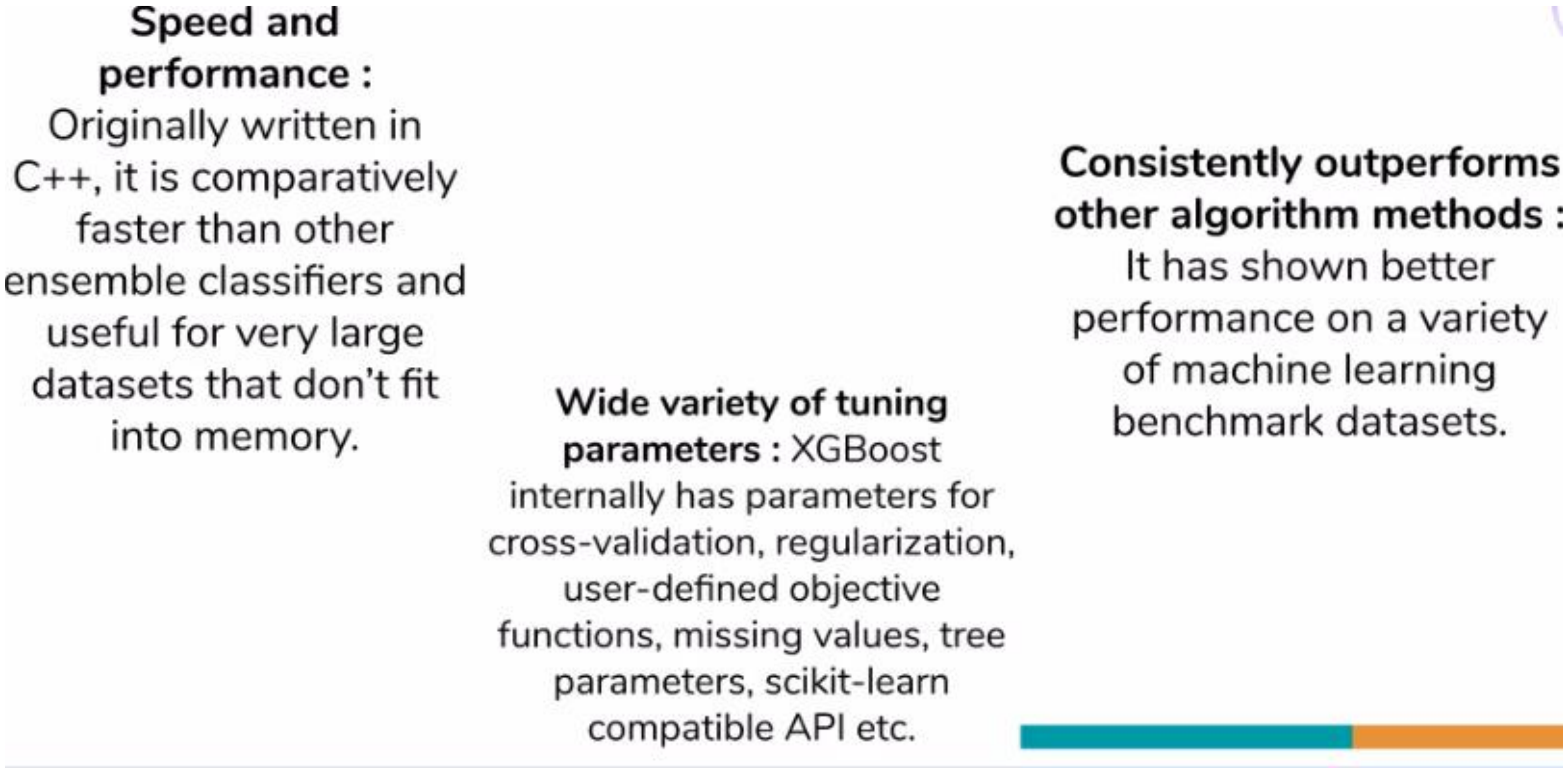
Unique Features of XGBoost Model

- XGBoost model is a popular implementation of **gradient boosting.** There are some features or metrics of XGBoost that make it so interesting:
- **Regularization:** XGBoost has an option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting
- **Handling sparse data:** Missing values or data processing steps like one-hot encoding make data sparse. XGBoost Classifier incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data
- **Weighted quantile sketch:** Most existing tree-based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data.

Unique Features of XGBoost Model

- **Block structure for parallel learning:** For faster computing, XGBoost Classifier can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sort and store data in in-memory units called blocks. Unlike other algorithms, this approach enables subsequent iterations to reuse the data layout instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling.
- **Cache awareness:** In XGBoost machine learning, Scala requires non-continuous memory access to obtain the gradient statistics by row index. Hence, Tianqi Chen designed XGBoost to optimize hardware usage. This optimization occurs by allocating internal buffers in each thread, where the workflow can store the gradient statistics. And these parallel tree make better XGboost algorithms with the help of julia and java lanuages.
- **Out-of-core computing:** This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

Why XGBoost?



Speed and performance : Originally written in C++, it is comparatively faster than other ensemble classifiers and useful for very large datasets that don't fit into memory.

Wide variety of tuning parameters : XGBoost internally has parameters for cross-validation, regularization, user-defined objective functions, missing values, tree parameters, scikit-learn compatible API etc.

Consistently outperforms other algorithm methods : It has shown better performance on a variety of machine learning benchmark datasets.