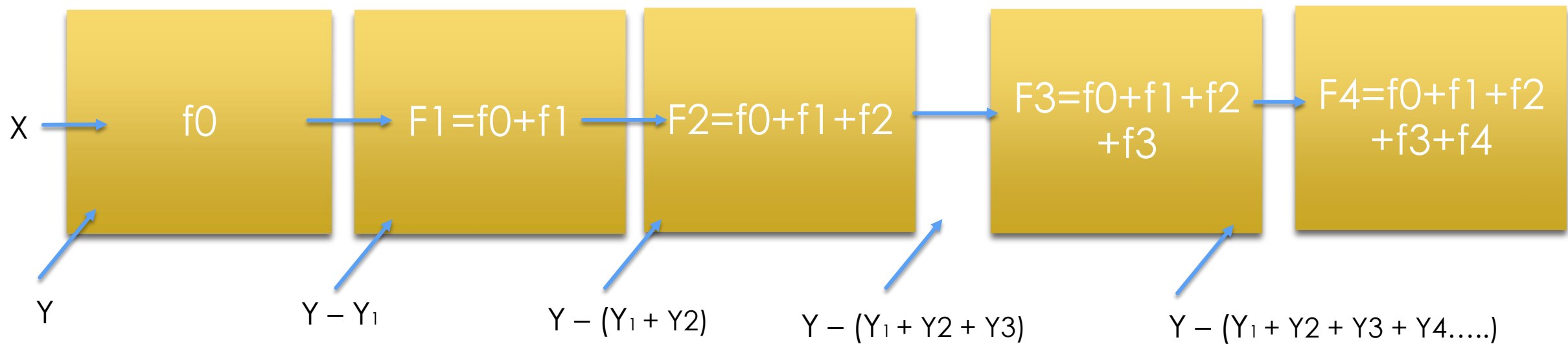




# Boosting Machines

# BOOSTING



# Boosting Machines

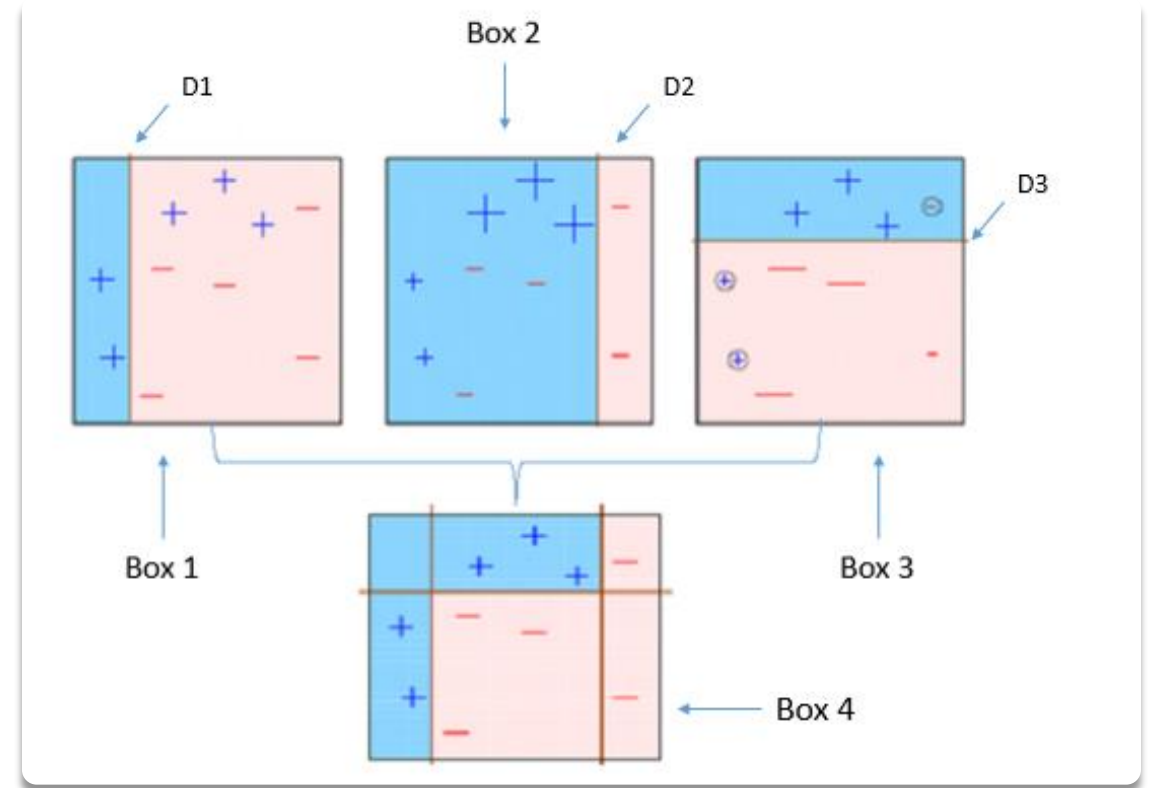
- ▶ Decision Tree Stump does not try to fit data to any specific pattern like linear model which would always try to fit data to linear equation.
- ▶ Input variables remain the same in all Weak Learners.
- ▶ Target variable is changed to the error value from the previous model.
- ▶ Target of first model is  $Y$
- ▶ Target for next models are  $Y - WL1 \dots$

# How Boosting Algorithms works?

- ▶ To find weak rule, we apply base learning algorithms with a different distribution. Each time base learning algorithm is applied, it generates a new weak prediction rule. This is an iterative process. After many iterations, the boosting algorithm combines these weak rules into a single strong prediction rule.
- ▶ For choosing the right distribution, here are the following steps:
  - ▶ Step 1: The base learner takes all the distributions and assign equal weight or attention to each observation.
  - ▶ Step 2: If there is any prediction error caused by first base learning algorithm, then we pay higher attention to observations having prediction error. Then, we apply the next base learning algorithm.
  - ▶ Step 3: Iterate Step 2 till the limit of base learning algorithm is reached or higher accuracy is achieved.
- ▶ Finally, it combines the outputs from weak learner and creates a strong learner which eventually improves the prediction power of the model. Boosting pays higher focus on examples which are misclassified or have higher errors by preceding weak rules.

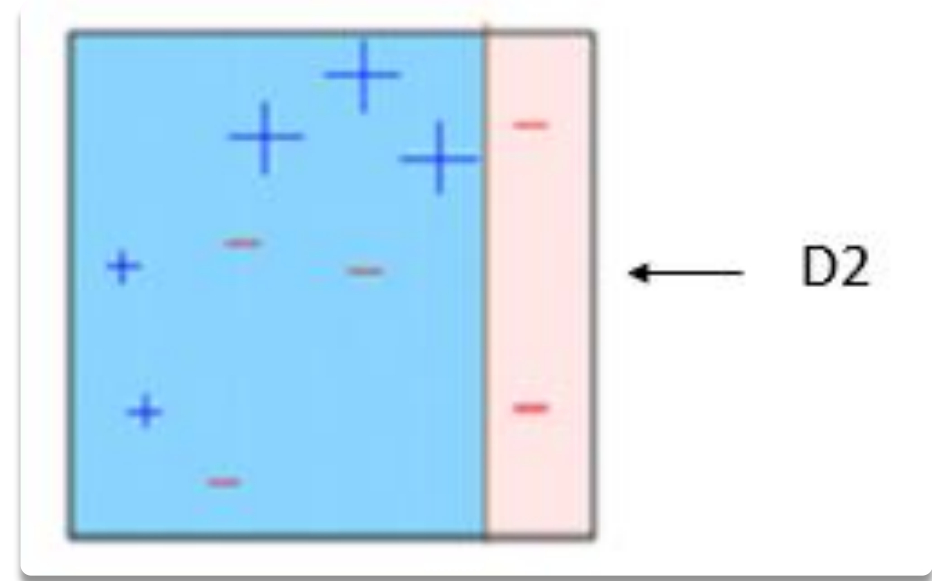
# AdaBoost

- Box 1: You can see that we have assigned equal weights to each data point and applied a decision stump (D1) to classify them as + (plus) or - (minus). The decision stump (D1) has generated vertical line at left side to classify the data points. We see that, this vertical line has incorrectly predicted three + (plus) as - (minus). In such case, we'll assign higher weights to these three + (plus) and apply another decision stump.



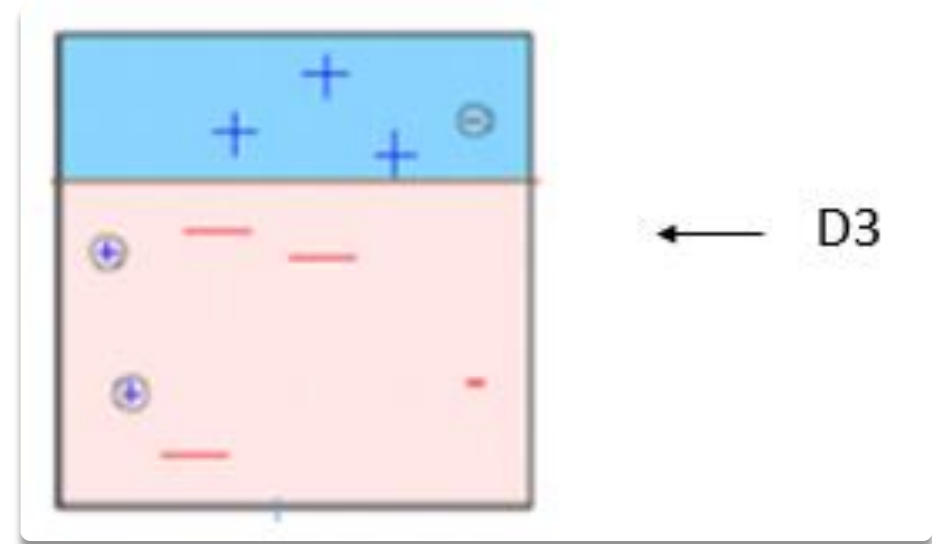
# AdaBoost

- Box 2: Here, you can see that the size of three incorrectly predicted + (plus) is bigger as compared to rest of the data points. In this case, the second decision stump (D2) will try to predict them correctly. Now, a vertical line (D2) at right side of this box has classified three misclassified + (plus) correctly. But again, it has caused misclassification errors. This time with three - (minus). Again, we will assign higher weight to three - (minus) and apply another decision stump.



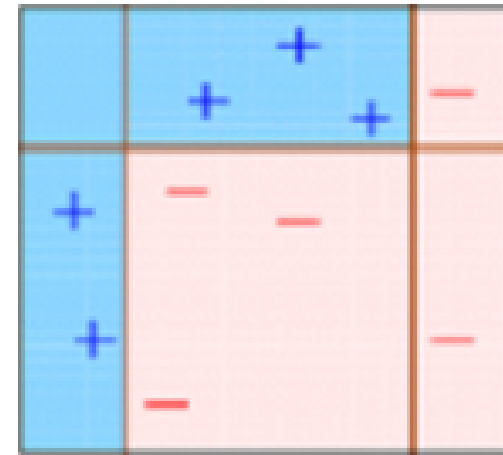
# Adaboost

- Box 3: Here, three – (minus) are given higher weights. A decision stump (D3) is applied to predict these misclassified observation correctly. This time a horizontal line is generated to classify + (plus) and – (minus) based on higher weight of misclassified observation.



# Adaboost

- Box 4: Here, we have combined D1, D2 and D3 to form a strong prediction having complex rule as compared to individual weak learner. You can see that this algorithm has classified these observation quite well as compared to any of individual weak learner.





## **AdaBoost** **(Adaptive Boosting)**

- ▶ It works on similar method as discussed above. It fits a sequence of weak learners on different weighted training data. It starts by predicting original data set and gives equal weight to each observation. If prediction is incorrect using the first learner, then it gives higher weight to observation which have been predicted incorrectly. Being an iterative process, it continues to add learner(s) until a limit is reached in the number of models or accuracy.
- ▶ Mostly, we use decision stamps with AdaBoost. But, we can use any machine learning algorithms as base learner if it accepts weight on training data set. We can use AdaBoost algorithms for both classification and regression problem.

# Math Behind the Boosting Algorithms

- ▶ Boosting consists of three simple steps:
- ▶ An initial model  $F_0$  is defined to predict the target variable  $y$ . This model will be associated with a residual  $(y - F_0)$
- ▶ A new model  $h_1$  is fit to the residuals from the previous step
- ▶ Now,  $F_0$  and  $h_1$  are combined to give  $F_1$ , the boosted version of  $F_0$ . The mean squared error from  $F_1$  will be lower than that from  $F_0$ :

$$F_1(x) \leftarrow F_0(x) + h_1(x)$$

- ▶ To improve the performance of  $F_1$ , we could model after the residuals of  $F_1$  and create a new model  $F_2$ :

$$F_2(x) \leftarrow F_1(x) + h_2(x)$$

- ▶ This can be done for ' $m$ ' iterations, until residuals have been minimized as much as possible:

$$F_m(x) \leftarrow F_{m-1}(x) + h_m(x)$$

- ▶ Here, the additive learners do not disturb the functions created in the previous steps. Instead, they impart information of their own to bring down the errors.

Prediction at iteration  $t = F_t(X) = \sum_{i=0}^t f_t(X)$

where  $f_t(X)$  is  $t^{th}$  weak learner

$$J = \sum L(y_i, F_t(X_i))$$

$$\frac{\delta J}{\delta F_t(X)} = \sum \frac{\delta L(y_i, F_t(X_i))}{\delta F_t(X)}$$

$$f_{t+1}(X) \rightarrow -\eta \frac{\delta J}{\delta F_t(X)}$$

$$F_{t+1}(X) = F_t(X) + f_{t+1}(X)$$

# BOOSTING Machines

# REGRESSION

$$L(y_i, F_t(x_i)) = (y_i - F_t(x_i))^2$$

$$\frac{\delta L}{\delta F_t(x)} \sim -(y_i - F_t(x_i))$$

$$f_{t+1}(x) \rightarrow -\eta \frac{\delta L}{\delta F_t(x)} \rightarrow \eta(y_i - F_t(x_i))$$

Model



$$p_i^{(t)} = \frac{1}{1 + e^{-F_t(x_i)}}$$

Loss



$$\begin{aligned} L(y_i, F_t(x_i)) &= -(y_i * \log(p_i^{(t)}) + (1 - y_i) * \log(1 - p_i^{(t)})) \\ &= \log(1 + e^{F_t(x_i)}) - y_i * F_t(x_i) \end{aligned}$$

gradient



$$\begin{aligned} \frac{\delta J}{\delta F_t(x_i)} &= -(y_i - \frac{1}{1 + e^{-F_t(x_i)}}) \\ &= -(y_i - p_i^{(t)}) \end{aligned}$$

Next WL


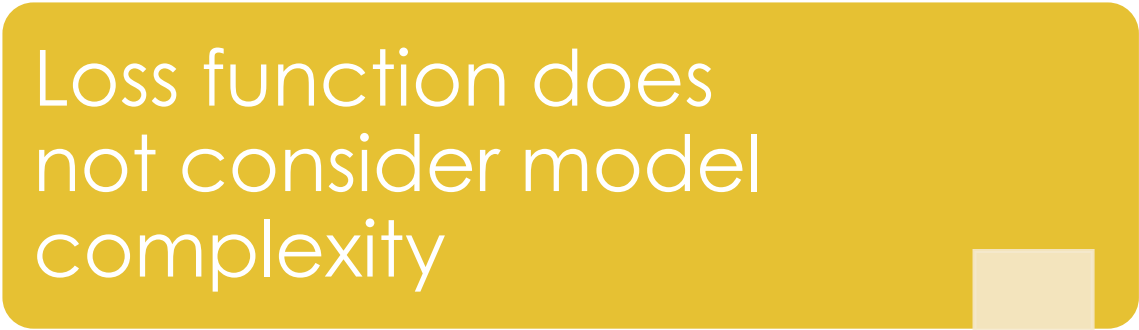


$$f_{t+1}(x) \rightarrow \eta(y_i - p_i^{(t)})$$

# CLASSIFICATION

## Issue with GBM

Loss function does  
not consider model  
complexity



Tends to overfit as it  
does not know  
where to stop.



Not generalised in  
nature



- ▶ Ever since its introduction in 2014, XGBoost has been lauded as the holy grail of machine learning hackathons and competitions. From predicting ad click-through rates to classifying high energy physics events, XGBoost has proved its mettle in terms of performance – and speed.
- ▶ Execution Speed: Generally, XGBoost is fast. Really fast when compared to other implementations of gradient boosting. But newly introduced LightGBM is faster than XGBoosting.
- ▶ Model Performance: XGBoost dominates structured or tabular datasets on classification and regression predictive modeling problems. The evidence is that it is the go-to algorithm for competition winners on the Kaggle competitive data science platform.

## XGBoosting (Extreme Gradient Boosting)

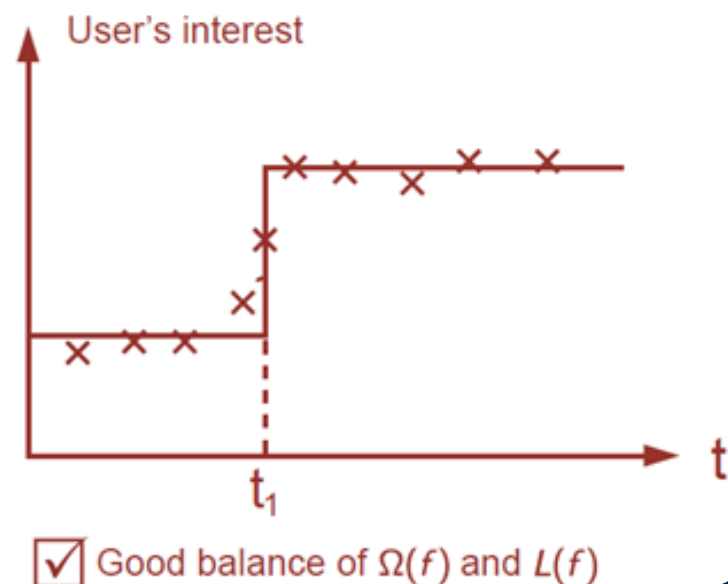
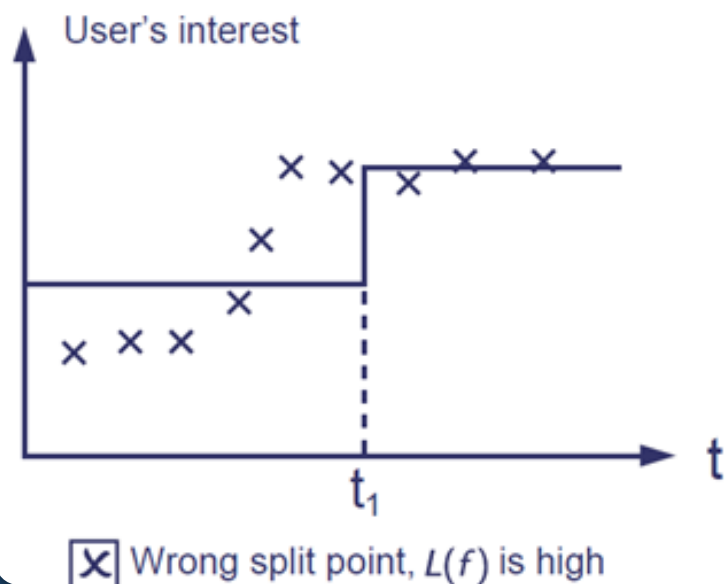
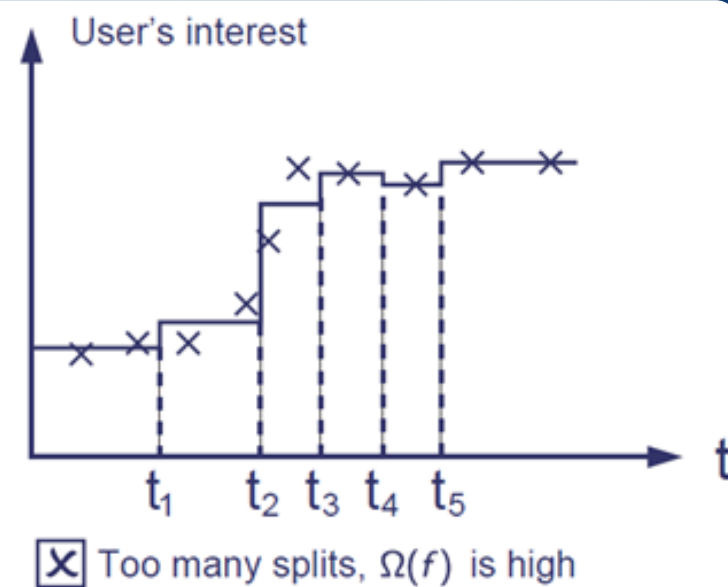
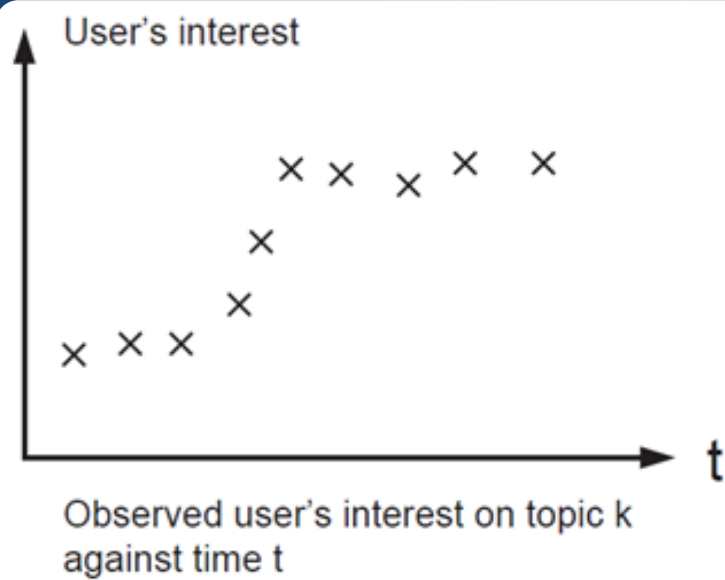
# What Algorithm Does XGBoost Use?

- ▶ The XGBoost library implements the gradient boosting decision tree algorithm.
- ▶ This algorithm goes by lots of different names such as gradient boosting, multiple additive regression trees, stochastic gradient boosting or gradient boosting machines.
- ▶ Boosting is an **ensemble technique** where new models are added to correct the errors made by existing models. Models are added sequentially until no further improvements can be made. A popular example is the AdaBoost algorithm that weights data points that are hard to predict.
- ▶ Gradient boosting is an approach where new models are created that predict the residuals or errors of prior models and then added together to make the final prediction. It is called gradient boosting because it uses a gradient descent algorithm to minimize the loss when adding new models.
- ▶ This approach supports both regression and classification predictive modeling problems.



# XGBoosting (Extreme Gradient Boosting)

- ▶ **What is the difference between the gbm (gradient boosting machine) and XGBoost (extreme gradient boosting)?**
- ▶ Both XGBoost and gbm follows the principle of gradient boosting. There are however, the difference in modeling details. Specifically, XGBoost used a more regularized model formalization to control over-fitting, which gives it better performance.
- ▶ Objective Function : Training Loss + Regularization
- ▶ The regularization term controls the complexity of the model, which helps us to avoid overfitting. This sounds a bit abstract, so let us consider the following problem in the following picture. You are asked to *fit* visually a step function given the input data points on the upper left corner of the image. Which solution among the three do you think is the best fit?



$$obj^{(t)} = \sum_{i=1}^n L(y_i, F_t(x_i)) + \sum_{i=1}^t \Omega(f_i)$$

# XGBoosting

- ▶ In the XGBoost package, at the  $t^{\text{th}}$  step we are tasked with finding the tree  $F_t$  that will minimize the following objective function:

$$Obj(F_t) = L(F_{t-1} + F_t) + \Omega(F_t)$$

where  $L(F_t)$  is our loss function and  $\Omega(F_t)$  is our regularization function.

- ▶ Regularization is essential to prevent overfitting to the training set. Without any regularization, the tree will split until it can predict the training set perfectly. This will usually mean that the tree has lost generality and will not do well on new test data. In XGBoost, the regularization function shows the model complexity.

$$\Omega(F_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

where  $T$  is the number of leaves in the tree,  $w_j$  is the score of leaf  $j$ ,  $\gamma$  is the leaf weight penalty parameter, and  $\lambda$  is the tree size penalty parameter.

- Determining how to find the function to optimize the above objective function is not clear.

$$Obj(F_t) = -\frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} + \gamma T$$

$$G_j = \sum_{i \in I_j} \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}^{(t-1)})$$

$$H_j = \sum_{i \in I_j} \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}^{(t-1)})$$

$$I = \{i | q(x_i) = j\}$$

where  $q(x)$  maps input features to a leaf node in the tree and  $l(y_i; \hat{y})$  is our loss function. This objective function is much easier to work with because it is now gives a score that we can use to determine how good a tree structure is.

# Model Complexity

- ▶ In XGBoost, we define the complexity as

$$\Omega(f) = \lambda n + \frac{1}{2} \sum_{j=1}^n w_j^2$$

- Of course there is more than one way to define the complexity, but this specific one works well in practice.
- The regularization is one part most tree packages treat less carefully, or simply ignore. This was because the traditional treatment of tree learning only emphasized improving impurity, while the complexity control was left to heuristics. By defining it formally, we can get a better idea of what we are learning, and it works well in practice.
- To see more comparative analysis:  
<https://www.kaggle.com/nschneider/gbm-vs-xgboost-vs-lightgbm/code>

# Unique features of XGBoost

- ▶ XGBoost is a popular implementation of gradient boosting. Let's discuss some features 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 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

- ▶ **Block structure for parallel learning:** For faster computing, XGBoost can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted and stored in in-memory units called blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling
- ▶ **Cache awareness:** In XGBoost, non-continuous memory access is required to get the gradient statistics by row index. Hence, XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored
- ▶ **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



## Parameters

- ▶ XGBoost requires a number of parameters to be selected. The following is a list of all the parameters that can be specified:
- ▶ **(eta) Shrinkage term.** Each new tree that is added has its weight shrunk by this parameter, preventing overfitting, but at the cost of increasing the number of rounds needed for convergence.
- ▶ **(gamma)** Tree size penalty
- ▶ **max depth** The maximum depth of each tree
- ▶ **min child weight** The minimum weight that a node can have. If this minimum is not met, that particular split will not occur.
- ▶ **subsample** Gives us the opportunity to perform "bagging," which means randomly sampling with replacement a proportion specified by parameter of the training examples to train each round on. The value is between 0 and 1 and is a method that helps prevent overfitting.
- ▶ **colsample bytree** Allows us to perform "feature bagging," which picks a proportion of the features to build each tree with. This is another way of preventing overfitting.
- ▶ **(lambda)** This is the L2 leaf node weight penalty