Boosting Methods

- **AdaBoosting** (Adaptive Boosting)
  - In AdaBoost, the successive learners are created with a focus on the ill fitteddata of the previous learner
  - Each successive learner focuses more and more on the harder to fit data i.e. their residuals in the previous tree
- **Gradient Boosting**
  - Each learner is fit on a modified version of original data. Original data is replaced with the x values and residuals from previous learner
  - By fitting new models to the residuals, the overall learner gradually improves in areas where residuals are initially high
- **XG Boost (Extreme Gradient Boosting)**
  - Upgraded implementation of Gradient Boosting. Developed for high computational speed, scalability, and better performance.
  - Parallel Implementation, Cross-Validation, Cache Optimization, Distributed Computation

What is the difference between bagging and boosting?

Bagging	Boosting
Parallel Training - All the weak learners are built in parallel i.e. independent of each other.	Sequential Training - Successive weak learners to improve the accuracy from the prior learners
Equal weightage - Each weak learner has equal weight in the final prediction.	Weighted average - More weight to those weak learners with better performance
Independent samples - Samples are drawn from the original dataset with replacement to train each individual weak learner	Dependent samples - Subsequent samples have more of those observations which had relatively higher errors in previous weak learners
Can help reduce variance of the model	Can help reduce bias of the model
Example: Bagging Classifier, Random Forest	Example: AdaBoost, Gradient Boosting Classifier

Summary - AdaBoost

1. Assign equal sample weights for each sample, that is, sample weight = 1 / number of samples

2. Bootstrap the samples as per the weights assigned and build a weak learner on that sample

3. Once the weak learner is built, AdaBoost chooses the alpha, which measures the importance of it based on the error made by that weak learner

$$\alpha_t = \frac{1}{2} log(\frac{1-\epsilon_t}{\epsilon_t})$$

4. Calculate the new sample weights for the next weak learner

   a. New sample weight for incorrect samples = sample weight * exp(alpha) / z_t

   b. New sample weight for correct samples = sample weight * exp(-alpha) / z_t

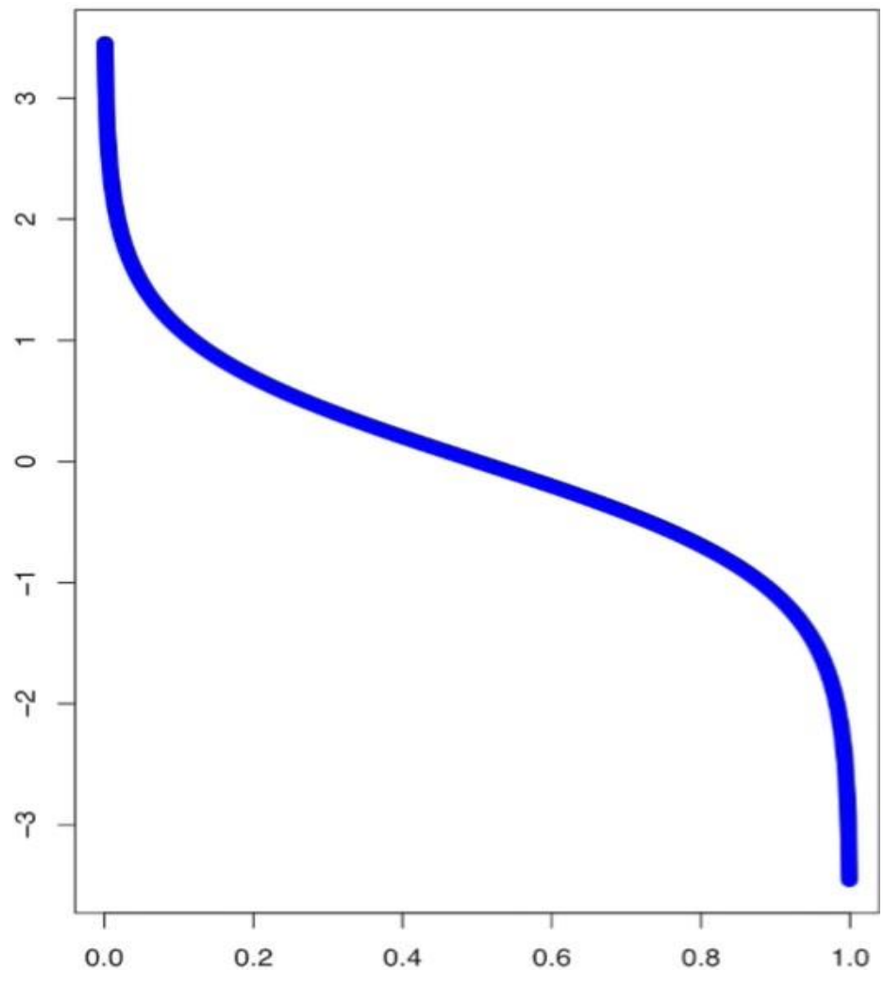
5. Create a bootstrapped dataset with the odds of each sample being chosen based on their new sample weights

6. Repeat the process n number of times

7. The final prediction is a weighted majority vote/average of all the weak learners

Summary - AdaBoost

- As evident from the graph, importance of each weak learner decreases with the increase in error made by that learner, that is when error is zero, importance of that weak learner is the highest.
- If the total error is greater than 0.5 then negative importance flips the class prediction.



Importance VS Error of each learner

Gradient Boosting for Classification

- It relies on the intuition that the next model, when combined with previous models, minimizes the overall prediction error
- In Gradient Boosting, instead of predicting the actual labels of the data at each iteration, it tries to predict the residual errors made by the previous predictor.

Steps involved in the gradient boosting algorithm for classification are

1. Initialize the model with initial prediction for all observations
2. Calculate the residual for each observation.
3. Build a tree and calculate the output value for each leaf node
4. Update all predictions using previous probabilities and new output value
5. Repeat steps 2-4 until maximum number of estimators reached

Let’s understand this with an example

Let’s consider an example dataset, where X is assumed to be the age of people and Y is whether they like a particular movie or not.

X	Y
10	0
20	1
30	1
40	1
50	0
60	1

- Let’s find log odds for Y =1
- **log(4/2) = 0.69**
- Let’s find out the probability using the below formula :

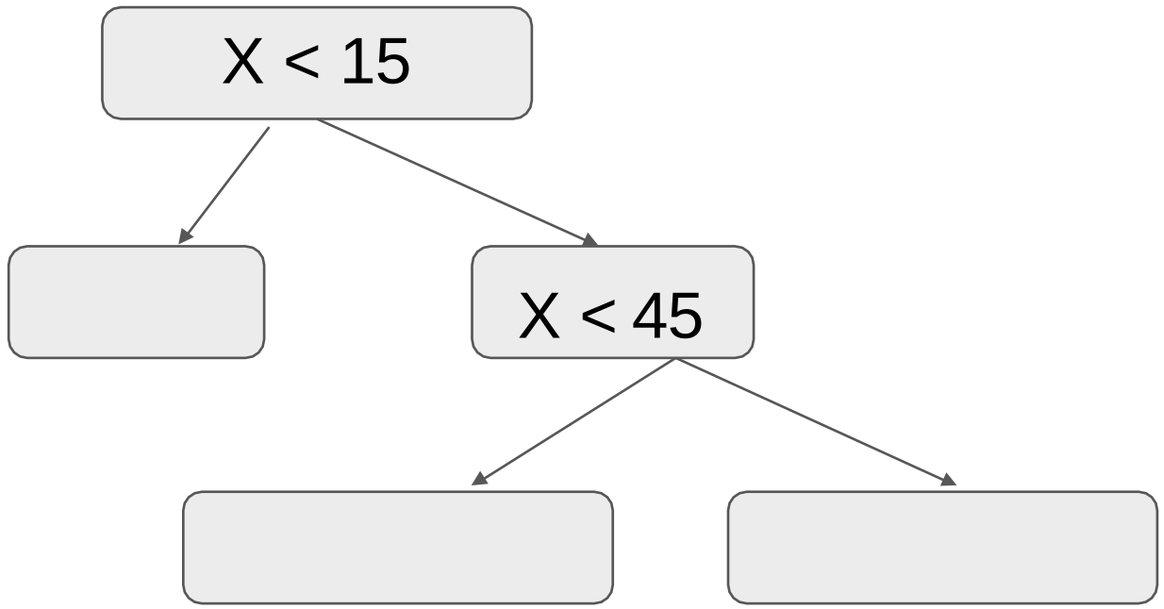
$$P = \frac{e^{log(odds)}}{1+e^{log(odds)}}$$

- **P(Y = 1) = (e^(0.69))/(1 + e^(0.69)) = 0.67**

Let’s calculate residuals for our predictions

We will build a tree with max_depth = 2 to predict residuals.

X	Y	Residual (observed-predicted probability)
10	0	-0.67
20	1	0.33
30	1	0.33
40	1	0.33
50	0	-0.67
60	1	0.33

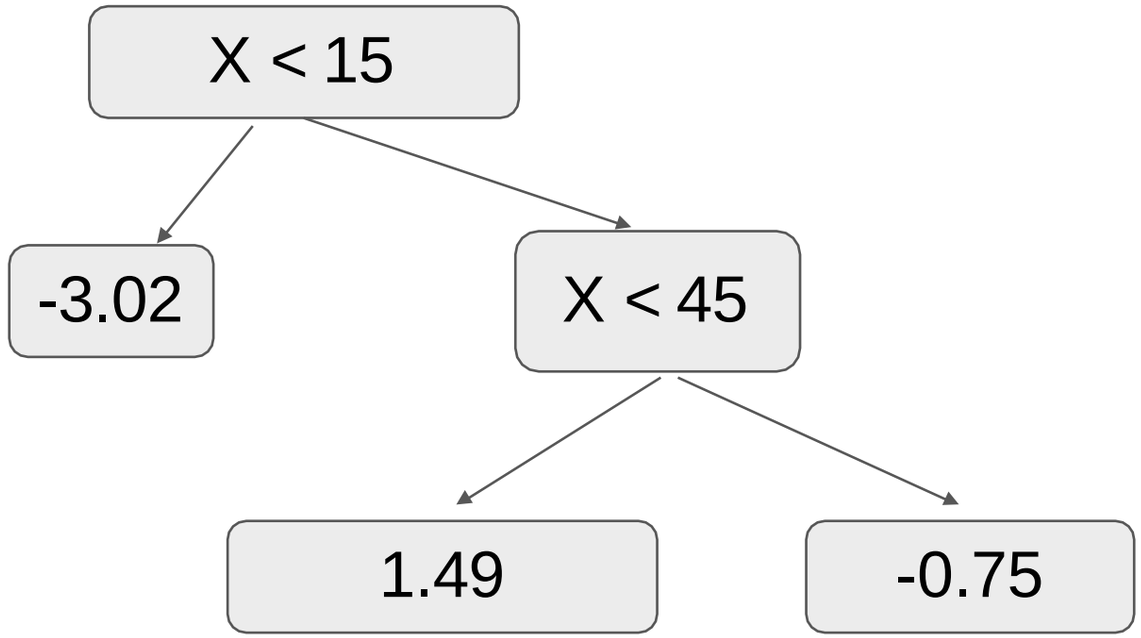


Transformation formula

We can calculate the output value of each leaf using the following formula:

$$\frac{\sum Residual}{\sum[PreviousProb * (1-PreviousProb)]}$$

X	Y	Residual
10	0	-0.67
20	1	0.33
30	1	0.33
40	1	0.33
50	0	-0.67
60	1	0.33



Update Predictions

We'll update our predictions using below formula with a learning rate of 0.8

$$OldTree + LearningRate * NewTree$$

X	Y	Residual	Y1 = Previous(log odds) + (LR * New values)	Output1 = New predictions (e^Y1) / (1 + e^Y1)
10	0	-0.67	-1.72	0.15
20	1	0.33	1.88	0.87
30	1	0.33	1.88	0.87
40	1	0.33	1.88	0.87
50	0	-0.67	0.09	0.52
60	1	0.33	0.09	0.52

Calculate the Residuals again

We'll calculate the residuals again and repeat the same steps until we get good prediction probabilities

X	Y	Residual	Y1 = Previous (log odds) + (LR * New values)	Output1 = predicted probability (e^Y1) / (1 + e^Y1)	Residuals
10	0	-0.67	-1.72	0.15	-0.15
20	1	0.33	1.88	0.87	0.13
30	1	0.33	1.88	0.87	0.13
40	1	0.33	1.88	0.87	0.13
50	0	-0.67	0.09	0.52	-0.52
60	1	0.33	0.09	0.52	0.48

Probabilities are closer to actual labels

Residuals are getting smaller

Loss function

- Loss function is same as is generally use in case of classification models with some variations

•The loss function used in gradient boosting is log-likelihood. Our goal is to maximize the log likelihood function (or minimize negative log likelihood).

$$log(likelihood\ of\ the\ observed\ data\ given\ the\ prediction) = [y_i * log(p) + (1 - y_i * log(1-p)]$$

Where, y_i is the observed value (0 or 1) and p is the predicted probability.

- Writing this in terms of log(odds) and differentiating, we get:

$$\frac{d}{dlog(odds)}(y_i log(odds) + log(1 + e^{log(odds)})) = -y_i + \frac{e^{log(odds)}}{1 + e^{log(odds)}}$$

XGBoost Overview

- XGBoost builds upon the idea of gradient boosting algorithm with some modifications. Gradient boosted trees are built in sequence because each estimator predicts residuals of the previous estimator, which makes it slow at the time of model training as compared to building estimators in parallel

- Thus, the main concentration in XGBoost is speed enhancement and model performance

- The speed and scalability of XGBoost is due to several important features that help in better computing using concepts like parallelization, cache optimization, out of core computing and distributed computing

XGBoost - Salient features

- **Parallelization** - In XGBoost, data is stored in in-memory units, called block, and the optimal split in each tree can be found in parallel using this block structure. This reduces the time of model training significantly.

- **Cache Optimization** - XGBoost optimizes the block size for efficient parallelization and make best use of hardware.

- **Out-of-Core Computing** - This helps to handle huge data sets which do not even fit into memory.

- **Distributed Computing** - It helps to train huge models using multiple similar machines.

- **Missing Values Imputation** - XGBoost is designed to handle missing values internally. It uses a default direction for the missing values. However, the default direction can be right-child or left-child, and it is learned in the tree construction process to choose the best direction that optimizes the training loss.

Steps for XGBoost

1.In XGBoost, initial prediction for all observations is taken as mean of the target values for regression and 0.5 for binary classification problems.

2. Calculate the residual for each observation.

3. Build a tree and calculate the output value for each leafnode

How a tree is built in XGBoost?

a. At each level, XGBoost calculates a **similarity score(ss)** for the node and the possible two leaves.

SS for Regression

$$\frac{(\sum Residuals)^2}{Number\ of\ Samples + \lambda}$$

SS for Classification

$$\frac{(\sum Residuals)^2}{[Probability * (1 - Probability)] + \lambda}$$

b. The formula for similarity score is derived directly from the loss function and helps to find the optimal point which minimizes the loss function.

c. Where λ is a regularization parameter in loss function of XGBoost

Steps for XGBoost

d. Gain is calculated for each possible split, where gain is calculated as -

Left leave ss + Right leave ss - Root node ss

e. The split which maximizes the gain is considered as best split and is chosen to grow a tree.

f. After building a tree, output value is calculated for each of the leaves.

Output Value - Regression

$$\frac{\sum Residuals}{Number\ of\ Samples + \lambda}$$

Output Value - Classification

$$\frac{\sum Residuals}{[Probability * (1 - Probability)] + \lambda}$$

4.Update all predictions using previous probabilities and new output value - same as Gradient Boosting.

5. Repeat steps 1-5 until maximum number of estimators reached.

Hyperparameter Tuning - Importance and Caution !

1. Needed to handle overfitting
2. Can help improve performance
3. But excessive use can bias the whole model not useful for ‘new’ data
4. Need to be careful and appropriate study of data should be done

Hyperparameters in XGBoost

- **Learning_rate / eta**: Gradient boosted decision trees are quick to learn and overfit training data. One effective way to slow down learning in the model is to use a learning rate, also called shrinkage.

- **gamma**: A node is split only when the resulting split gives a positive reduction in the loss function. Gamma specifies the minimum loss reduction required to make a split. Higher the gamma value, lesser the chances of overfitting

- **scale_pos_weight:** Control the balance of positive and negative weights. It can have any positive value as input. It helps in imbalanced classification problems.

Hyperparameters in XGBoost

A key one is to select a subset of columns while building an individual tree, level in a tree or node split in a level in a tree

- **colsample_bytree:** This denotes the the ratio of columns to be used when constructing each tree. It takes input as float values between (0,1] i.e. between greater than 0% of features and less than or equal to 100% of features.
- **colsample_bylevel:** This denotes the ratio of columns for each new level of a tree. If colsample_bytree is used, then columns are subsampled from the set of columns chosen for the current tree.
- **colsample_bynode:** This denotes the ratio of columns to be used for each node i.e. for each split. If colsample_bylevel is used, then columns are subsampled from the set of columns chosen for the current level.

Note: Three parameters defined above works cumulatively, for example, if we have total 32 columns and if {'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5}, then we leave only 4 columns to choose from at each split.