**XGBoost ensemble**

Everyone makes mistakes — even the simplest [decision trees](https://towardsdatascience.com/decision-tree-classifier-explained-a-visual-guide-with-code-examples-for-beginners-7c863f06a71e) in machine learning. Instead of ignoring them, AdaBoost (Adaptive Boosting) algorithm does something different: it learns (or *adapts*) from these mistakes to get better.

Unlike [Random Fores](https://towardsdatascience.com/random-forest-explained-a-visual-guide-with-code-examples-9f736a6e1b3c)t, which makes many trees at once, AdaBoost starts with a single, simple tree and identifies the instances it misclassifies. It then builds new trees to fix those errors, learning from its mistakes and getting better with each step.

Here, we’ll illustrate exactly how AdaBoost makes its predictions, building strength by combining targeted weak learners just like a workout routine that turns focused exercises into full-body power.

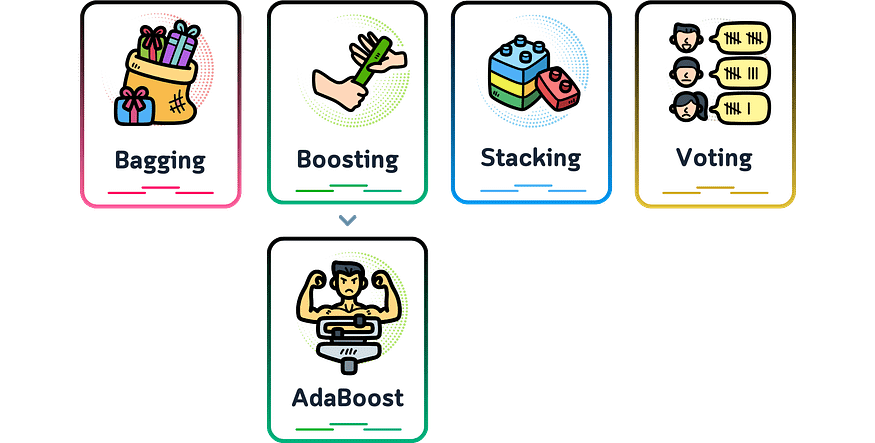
**Definition**

AdaBoost is an ensemble machine learning model that creates a sequence of weighted decision trees, typically using shallow trees (often just single-level “stumps”). Each tree is trained on the entire dataset, but with adaptive sample weights that give more importance to previously misclassified examples.

For classification tasks, AdaBoost combines the trees through a weighted voting system, where better-performing trees get more influence in the final decision.

The model’s strength comes from its adaptive learning process — while each simple tree might be a “weak learner” that performs only slightly better than random guessing, the weighted combination of trees creates a “strong learner” that **progressively focuses on and corrects mistakes**.

Press enter or click to view image in full size



AdaBoost is part of the boosting family of algorithms because it builds trees one at a time. Each new tree tries to fix the mistakes made by the previous trees. It then uses a weighted vote to combine their answers and make its final prediction.

**Main Mechanism**

Here’s how AdaBoost works:

1. **Initialize Weights:** Assign equal weight to each training example.
2. **Iterative Learning:** In each step, a simple decision tree is trained and its performance is checked. Misclassified examples get more weight, making them a priority for the next tree. Correctly classified examples stay the same, and all weights are adjusted to add up to 1.
3. **Build Weak Learners:** Each new, simple tree targets the mistakes of the previous ones, creating a sequence of specialized weak learners.
4. **Final Prediction:** Combine all trees through weighted voting, where each tree’s vote is based on its importance value, giving more influence to more accurate trees.

**How Does The AdaBoost Work?**

We can understand the working of the AdaBoost algorithm in step by step manner as going deep into the work, we can see there are multiple basic steps which this algorithm follows. Let’s take a look at these steps.

Step 1: When the algorithm is given data, it starts by Assigning equal weights to all training examples in the dataset. These weights represent the importance of each sample during the training process.

Step 2: Here, this algorithm iterates with a few algorithms for a specified number of iterations (or until a stopping criterion is met). The algorithm trains a weak classifier on the training data. Here the weak classifier can be considered a model that performs slightly better than random guessing, such as a decision stump (a one-level decision tree).

Step 3: During each iteration, the algorithm trains the weak classifier on given training data with the current sample weights. The weak classifier aims to minimize the classification error, weighted by the sample weights.

Step 4: After training the weak classifier, the algorithm calculates classifier weight based on the errors of the weak classifier. A weak classifier with a lower error receives a higher weight.

Step 4: Once the calculation of weight completes, the algorithm updates sample weights, and the algorithm gives assigns higher weights to misclassified examples so that more importance in subsequent iterations can be given.

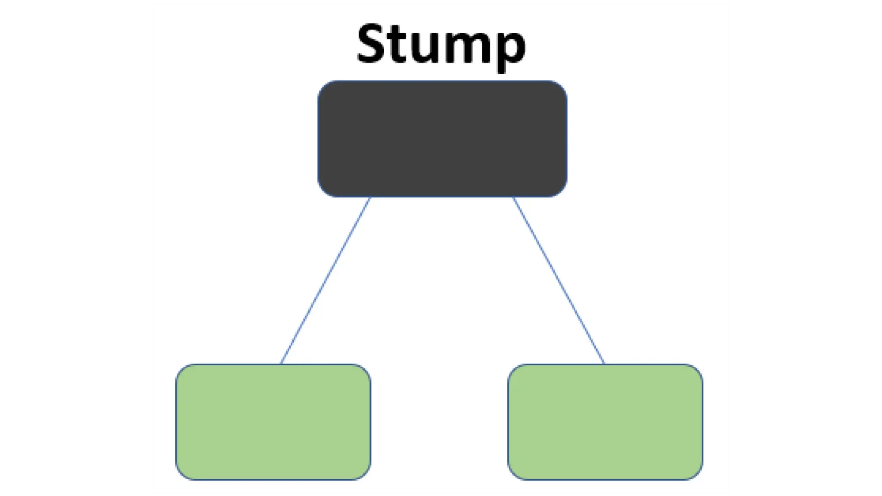
Step 5: After updating the sample weights, they are normalized so that they sum up to 1 and Combine the predictions of all weak classifiers using a weighted majority vote. The weights of the weak classifiers are considered when making the final prediction.

Step 6: Finally, Steps 2–5 are repeated for the specified number of iterations (or until the stopping criterion is met), with the sample weights updated at each iteration. The final prediction is obtained by aggregating the predictions of all weak classifiers based on their weights.

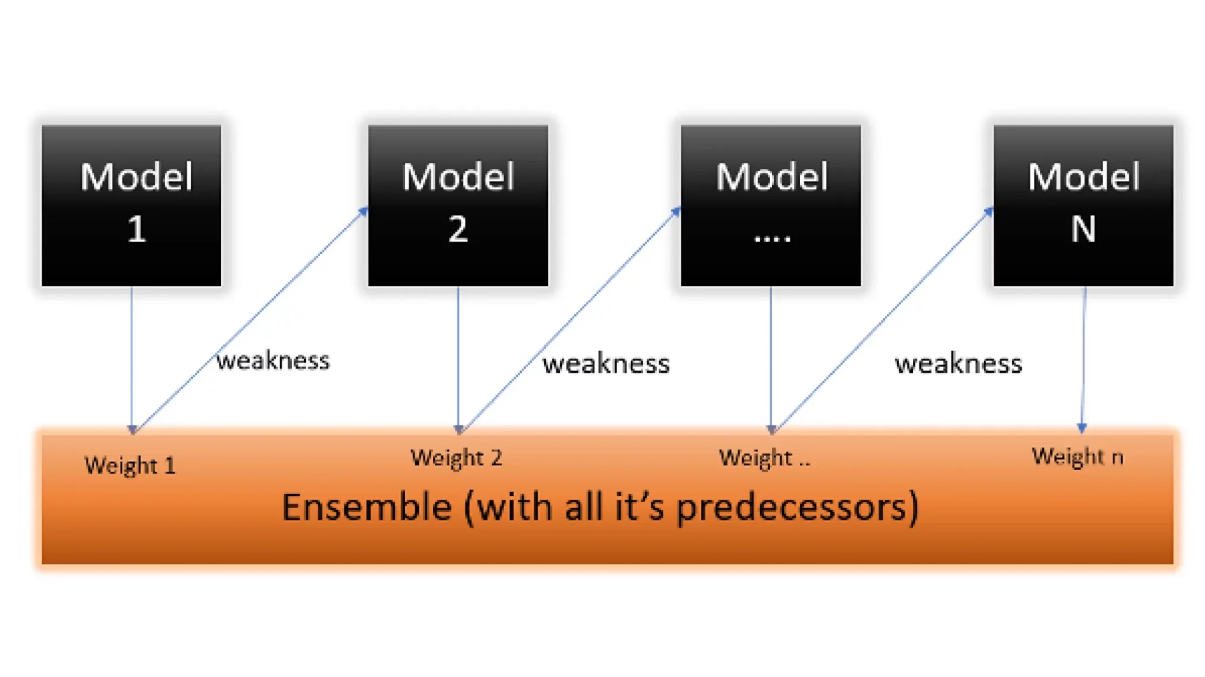
What Is the AdaBoost Algorithm?

There are many [**machine learning algorithms**](https://www.analyticsvidhya.com/blog/2022/01/machine-learning-algorithms/) to choose from for your problem statements. One of these algorithms for predictive modeling is called AdaBoost.

AdaBoost algorithm, short for Adaptive Boosting, is **a Boosting technique used as an**[**Ensemble Method in Machine Learning**](https://www.analyticsvidhya.com/blog/2018/06/comprehensive-guide-for-ensemble-models/). It is called Adaptive Boosting as the weights are re-assigned to each instance, with higher weights assigned to incorrectly classified instances.



What this algorithm does is that it builds a model and gives equal weights to all the data points. It then assigns higher weights to points that are wrongly classified. Now all the points with higher weights are given more importance in the next model. It will keep training models until and unless a lower error is received.



Let’s take an example to understand this, suppose you built a decision tree algorithm on the Titanic dataset, and from there, you get an accuracy of 80%. After this, you apply a different algorithm and check the accuracy, and it comes out to be 75% for KNN and 70% for Linear Regression.

When building different models on the same dataset, we observe variations in accuracy. However, leveraging the power of AdaBoost classifier, we can combine these algorithms to enhance the final predictions. By averaging the results from diverse models, Adaboost allows us to achieve higher accuracy and bolster predictive capabilities effectively.

Here we will be more focused on mathematics intuition.

There is another ensemble learning algorithm called the gradient ada boosting algorithm. In this algorithm, we try to reduce the error instead of wights, as in AdaBoost. But in this article, we will only be focussing on the mathematical intuition of Adaptive Boosting.

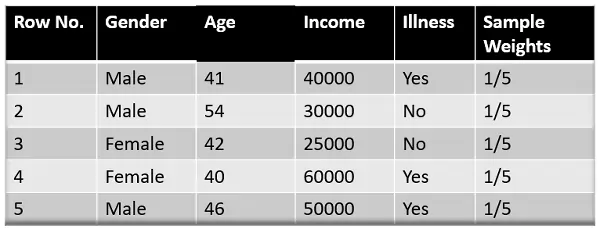
***Checkout this article about***[***AdaBoost Algorithm with Python***](https://www.analyticsvidhya.com/blog/2021/03/introduction-to-adaboost-algorithm-with-python/)

Understanding the Working of the AdaBoost Algorithm

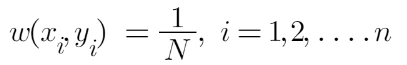
Let’s understand what and how this algorithm works under the hood with the following tutorial.

Step 1: Assigning Weights

The Image shown below is the actual representation of our dataset. Since the target column is binary, it is a classification problem. First of all, these data points will be assigned some weights. Initially, all the weights will be equal.



The formula to calculate the sample weights is:



Where N is the total number of data points

Here since we have 5 data points, the sample weights assigned will be 1/5.

Step 2: Classify the Samples

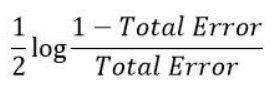
We start by seeing how well “*Gender*” classifies the samples and will see how the variables (Age, Income) classify the samples.

We’ll create a decision stump for each of the features and then calculate the ***Gini Index***of each tree. The tree with the lowest Gini Index will be our first stump.

Here in our dataset, let’s say ***Gender*** has the lowest gini index, so it will be our first stump.

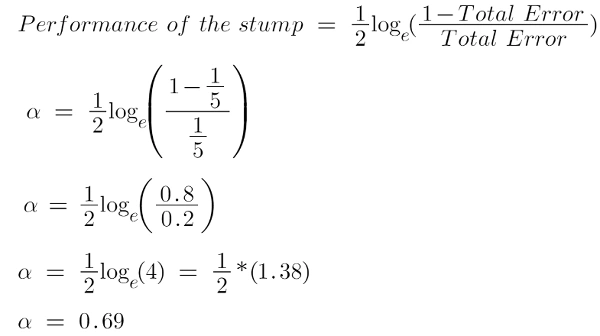
Step 3: Calculate the Influence

We’ll now calculate the **“Amount of Say”**or**“Importance”**or **“Influence”**for this classifier in classifying the data points using this formula:



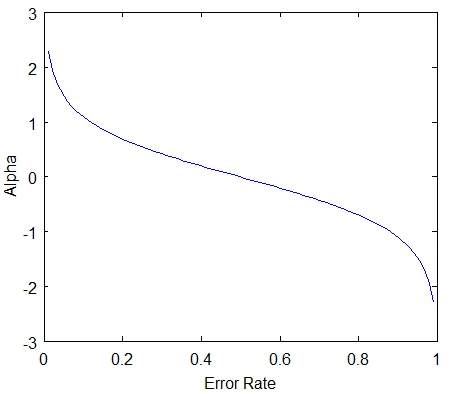
The total error is nothing but the summation of all the sample weights of misclassified data points.

Here in our dataset, let’s assume there is 1 wrong output, so our total error will be 1/5, and the alpha (performance of the stump) will be:



**Note**: Total error will always be between 0 and 1.

0 Indicates perfect stump, and 1 indicates horrible stump.



From the graph above, we can see that when there is no misclassification, then we have no error (Total Error = 0), so the “amount of say (alpha)” will be a large number.

When the classifier predicts half right and half wrong, then the Total Error = 0.5, and the importance (amount of say) of the classifier will be 0.

If all the samples have been incorrectly classified, then the error will be very high (approx. to 1), and hence our alpha value will be a negative integer.

***If you are Beginner and want to know***[***AdaBoost you can go with this article***](https://www.analyticsvidhya.com/blog/2022/01/introduction-to-adaboost-for-absolute-beginners/)

Step 4: Calculate TE and Performance

You might be wondering about the significance of calculating the Total Error (TE) and performance of an Adaboost stump. The reason is straightforward – updating the weights is crucial. If identical weights are maintained for the subsequent model, the output will mirror what was obtained in the initial model.

The wrong predictions will be given more weight, whereas the correct predictions weights will be decreased. Now when we build our next model after updating the weights, more preference will be given to the points with higher weights.

After finding the importance of the classifier and total error, we need to finally update the weights, and for this, we use the following formula:

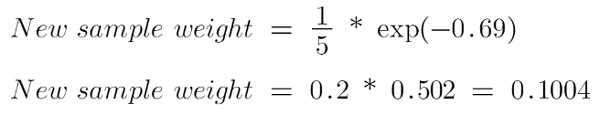


The amount of, say (alpha) will be ***negative***when the sample is **correctly classified**.

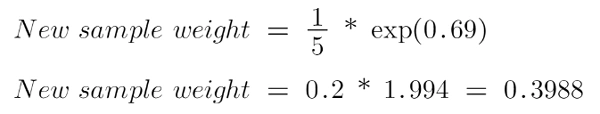
The amount of, say (alpha) will be ***positive*** when the sample is **miss-classified.**

There are four correctly classified samples and 1 wrong. Here, the ***sample weight*** of that datapoint is*1/5,* and the ***amount of say/performance of the stump***of***Gender*** is *0.69*.

New weights for *correctly classified* samples are:

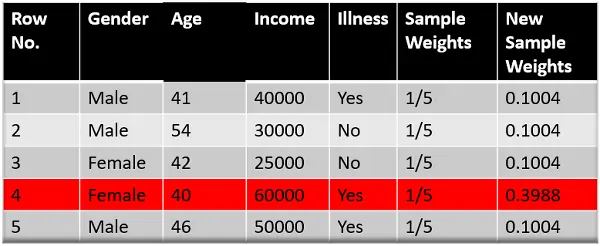


For *wrongly classified* samples, the updated weights will be:



**Note**

See the sign of alpha when I am putting the values, the **alpha is negative** when the data point is correctly classified, and this *decreases the sample weight* from 0.2 to 0.1004. It is **positive** when there is **misclassification**, and this will *increase the sample weight* from 0.2 to 0.3988



We know that the total sum of the sample weights must be equal to 1, but here if we sum up all the new sample weights, we will get 0.8004. To bring this sum equal to 1, we will normalize these weights by dividing all the weights by the total sum of updated weights, which is 0.8004. So, after normalizing the sample weights, we get this dataset, and now the sum is equal to 1.



Step 5: Decrease Errors

Now, we need to make a new dataset to see if the errors decreased or not. For this, we will remove the “sample weights” and “new sample weights” columns and then, based on the “new sample weights,” divide our data points into buckets.



Step 6: New Dataset

We are almost done. Now, what the algorithm does is selects random numbers from 0-1. Since incorrectly classified records have higher sample weights, the probability of selecting those records is very high.

Suppose the 5 random numbers our algorithm take is 0.38,0.26,0.98,0.40,0.55.

Now we will see where these random numbers fall in the bucket, and according to it, we’ll make our new dataset shown below.



This comes out to be our new dataset, and we see the data point, which was wrongly classified, has been selected 3 times because it has a higher weight.

***ReadMore about the***[***Gradient Boosting Algorithm for Beginners***](https://www.analyticsvidhya.com/blog/2021/09/gradient-boosting-algorithm-a-complete-guide-for-beginners/)

Step 7: Repeat Previous Steps

Now this act as our new dataset, and we need to repeat all the above steps i.e.

* Assign equal weights to all the data points.
* Find the stump that does the best job classifying the new collection of samples by finding their Gini Index and selecting the one with the lowest Gini index.
* Calculate the “Amount of Say” and “Total error” to update the previous sample weights.
* Normalize the new sample weights.

Iterate through these steps until and unless a low training error is achieved.

Suppose, with respect to our dataset, we have constructed 3 decision trees (DT1, DT2, DT3) in a ***sequential manner.*** If we send our **test data** now, it will pass through all the decision trees, and finally, we will see which class has the majority, and based on that, we will do predictions  
for our test dataset.

Python implementation of AdaBoost

To implement the AdaBoost algorithm in Python, you can either build it from scratch or use libraries like Scikit-learn.

Building AdaBoost from Scratch

Here’s a simple implementation of the AdaBoost algorithm using only NumPy:

python

import numpy as np

class DecisionStump:

def \_\_init\_\_(self):

self.polarity = 1

self.feature\_idx = None

self.threshold = None

self.alpha = None

def predict(self, X):

n\_samples = X.shape[0]

predictions = np.ones(n\_samples)

feature\_column = X[:, self.feature\_idx]

if self.polarity == 1:

predictions[feature\_column < self.threshold] = -1

else:

predictions[feature\_column > self.threshold] = -1

return predictions

class AdaBoost:

def \_\_init\_\_(self, n\_clf=5):

self.n\_clf = n\_clf

self.clfs = []

def fit(self, X, y):

n\_samples, n\_features = X.shape

w = np.full(n\_samples, (1 / n\_samples))

for \_ in range(self.n\_clf):

clf = DecisionStump()

min\_error = float('inf')

for feature\_i in range(n\_features):

X\_column = X[:, feature\_i]

thresholds = np.unique(X\_column)

for threshold in thresholds:

predictions = np.ones(n\_samples)

predictions[X\_column < threshold] = -1

error = sum(w[y != predictions])

if error > 0.5:

error = 1 - error

p = -1

else:

p = 1

if error < min\_error:

clf.polarity = p

clf.threshold = threshold

clf.feature\_idx = feature\_i

min\_error = error

EPS = 1e-10

clf.alpha = 0.5 \* np.log((1.0 - min\_error + EPS) / (min\_error + EPS))

predictions = clf.predict(X)

w \*= np.exp(-clf.alpha \* y \* predictions)

w /= np.sum(w)

self.clfs.append(clf)

def predict(self, X):

clf\_preds = [clf.alpha \* clf.predict(X) for clf in self.clfs]

y\_pred = np.sum(clf\_preds, axis=0)

return np.sign(y\_pred)

Using Scikit-learn

If you prefer a more straightforward approach, you can use the Scikit-learn library, which has a built-in AdaBoost classifier. Here’s how to do it:

from sklearn.ensemble import AdaBoostClassifier

from sklearn.tree import DecisionTreeClassifier

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import accuracy\_score

import pandas as pd

# Load dataset

data = pd.read\_csv("Iris.csv") # Adjust the file path as necessary

X = data.iloc[:, :-1].values # Features

y = data.iloc[:, -1].values # Target

# Split the dataset

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# Create the AdaBoost classifier

abc = AdaBoostClassifier(base\_estimator=DecisionTreeClassifier(max\_depth=1), n\_estimators=50)

# Fit the model

abc.fit(X\_train, y\_train)

# Predict and evaluate

y\_pred = abc.predict(X\_test)

print("Accuracy:", accuracy\_score(y\_test, y\_pred))