Boosting Algorithms

Definition: The term 'Boosting' refers to a family of algorithms which converts weak learner to strong learners.

Boosting is an ensemble method for improving the model predictions of any given learning algorithm. The idea of boosting is to train weak learners sequentially, each trying to correct its predecessor.

Boosting algorithms grant superpowers to machine learning models to improve their prediction accuracy.

What is Boosting?

Boosting is a method of converting a set of weak learners into strong learners. Suppose we have a binary classification task. A weak learner has an error rate that is slightly lesser than 0.5 in classifying the object, i.e the weak learner is slightly better than deciding from a coin toss. A strong learner has an error rate closer to 0. To convert a weak learner into strong learner, we take a family of weak learners, combine them and vote. This turns this family of weak learners into strong learners.

Let's understand this definition in detail by solving a problem of spam email identification:

How would you classify an email as SPAM or not? Like everyone else, our initial approach would be to identify 'spam' and 'not spam' emails using following criteria. If:

1. Email has only one image file (promotional image), It's a SPAM
2. Email has only link(s), It's a SPAM
3. Email body consist of sentence like "You won a prize money of $ xxxxxx", It's a SPAM
4. Email from our official domain "Analyticsvidhya.com" , Not a SPAM
5. Email from known source, Not a SPAM

Above, we've defined multiple rules to classify an email into 'spam' or 'not spam'. But, do you think these rules individually are strong enough to successfully classify an email? No.

Individually, these rules are not powerful enough to classify an email into 'spam' or 'not spam'. Therefore, these rules are called as weak learner.

To convert weak learner to strong learner, we'll combine the prediction of each weak learner using methods like:

- Using average/ weighted average

- Considering prediction has higher vote

For example: Above, we have defined 5 weak learners. Out of these 5, 3 are voted as ‘SPAM’ and 2 are voted as ‘Not a SPAM’. In this case, by default, we’ll consider an email as SPAM because we have higher(3) vote for ‘SPAM’.

To find weak rule, we apply base learning (ML) 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.

Types of Boosting Algorithms

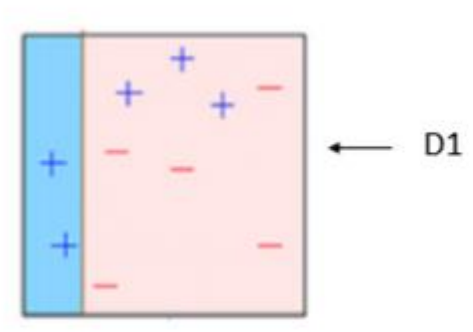
There are many boosting algorithms which use other types of engine such as:

1. AdaBoost (**Ada**ptive **Boost**ing)
2. Gradient Tree Boosting
3. XGBoost

AdaBoost (Adaptive Boosting)

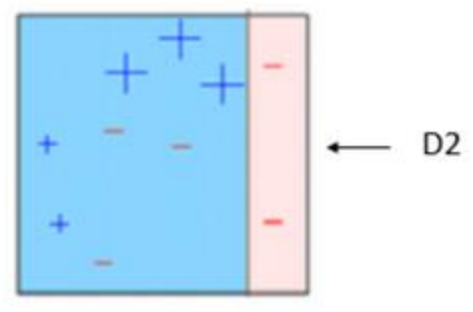
AdaBoost works on improving the areas where the base learner fails. The base learner is a machine learning algorithm which is a weak learner and upon which the boosting method is

applied to turn it into a strong learner. Any machine learning algorithm that accept weights on training data can be used as a base learner. In the example taken below, Decision stumps are used as the base learner.
We take the training data and randomly sample points from this data and apply decision stump algorithm to classify the points. After classifying the sampled points we fit the decision tree stump to the complete training data. This process iteratively happens until the complete training data fits without any error or until a specified maximum number of estimators.
After sampling from training data and applying the decision stump, the model fits as showcased below.



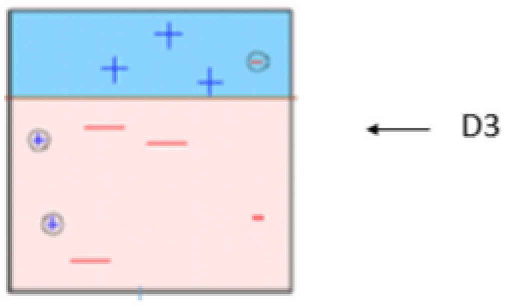
Decision Stump 1

We can observe that three of the positive samples are misclassified as negative. Therefore, we exaggerate the weights of these misclassified samples so that that they have a better chance of being selected when sampled again.



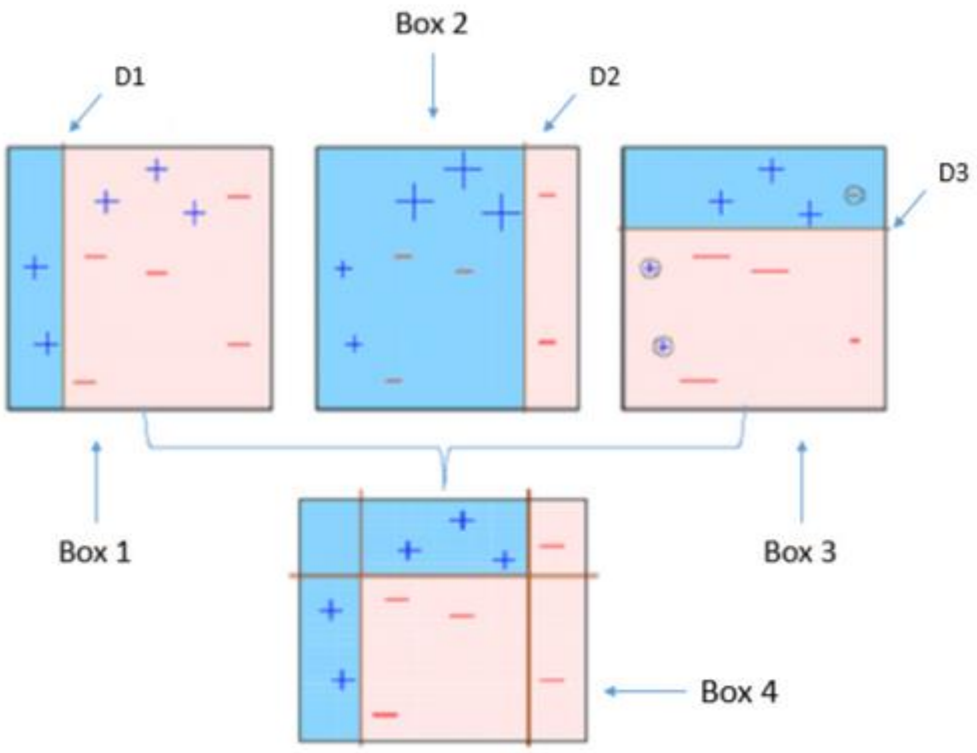
Decision Stump 2

When data is sampled next time, the decision stump 2 is combined with decision stump 1 to fit the training data. Therefore we have a miniature ensemble here trying to fit the data perfectly. This miniature ensemble of two decision stumps misclassifies three negative samples as positive. Therefore, we exaggerate the weights of these misclassified samples so that that they have a better chance of being selected when sampled again.



Decision Stump 3

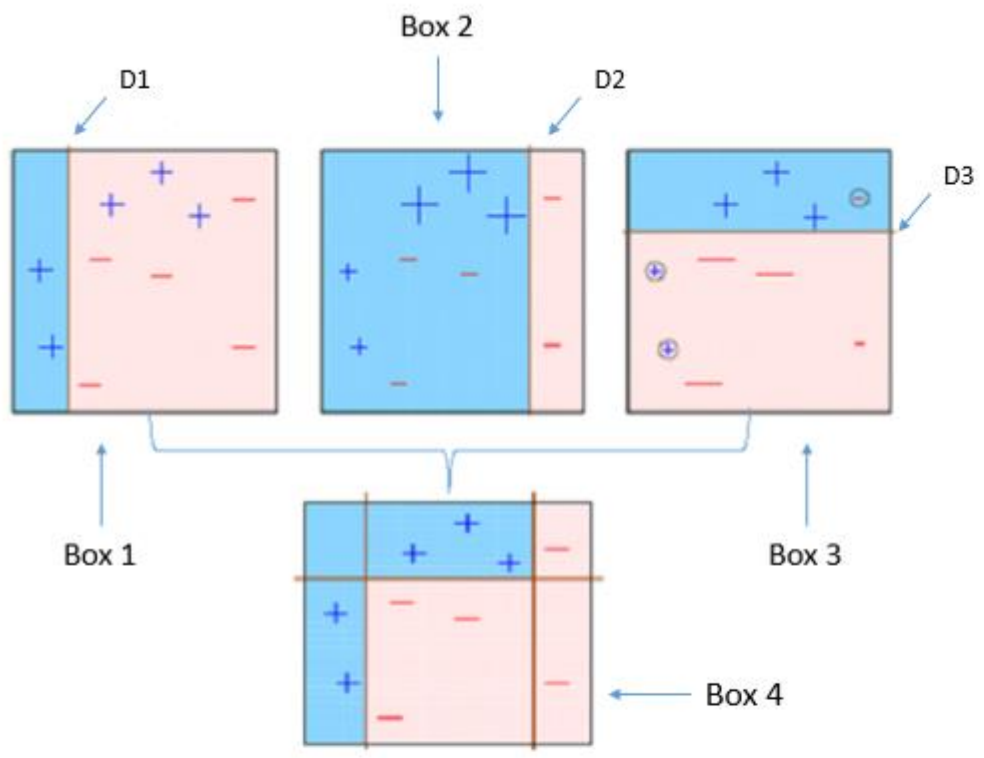
The previously misclassified samples are chosen and decision stump 3 is applied to fit the training data. We can find that two positive samples are classified as negative and one negative sample is classified as positive. Then the ensemble of three decision stumps(1, 2 and 3) are used to fit the complete training data. When this ensemble of three decision stumps are used the model fits the training data perfectly.



Ensemble of 3 Decision Stumps

The drawback of AdaBoost is that it is easily defeated by noisy data, the efficiency of the algorithm is highly affected by outliers as the algorithm tries to fit every point perfectly.

Adaboost combines multiple weak learners into a single strong learner. The weak learners in AdaBoost are decision trees with a single split, called decision stumps. When AdaBoost creates its first decision stump, all observations are weighted equally. To correct the previous error, the observations that were incorrectly classified now carry more weight than the observations that were correctly classified. AdaBoost algorithms can be used for both classification and regression problem.



As we see above, the first decision stump(D1) is made separating the (+) blue region from the (—) red region. We notice that D1 has three incorrectly classified (+) in the red region. The incorrect classified (+) will now carry more weight than the other observations and fed to the second learner. The model will continue and adjust the error faced by the previous model until the most accurate predictor is built.

```
# Adaboost
from sklearn.ensemble import AdaBoostClassifier
clf = AdaBoostClassifier()
# n_estimators = 50 (default value)
# base_estimator = DecisionTreeClassifier (default value)
clf.fit(X_train, y_train)
y_pred1 = clf.predict(X_test)
```

Boosting Algorithm: Gradient Boosting

In gradient boosting, it trains many model sequentially. Each new model gradually minimizes the loss function ($y = ax + b + e$, e needs special attention as it is an error term) of the whole system using Gradient Descent method. The learning procedure consecutively fit new models to provide a more accurate estimate of the response variable.

The principle idea behind this algorithm is to construct new base learners which can be maximally correlated with negative gradient of the loss function, associated with the whole ensemble.

GBM algorithm can be given by following steps.

- Fit a model to the data, F1(x) = y
- Create a new model, F2(x) = F1(x) + h1(x)
- By combining weak learner after weak learner, our final model is able to account for a lot of the error from the original model and reduces this error over time.

Gradient Boosting trains many models in a gradual, additive and sequential manner. The major difference between AdaBoost and Gradient Boosting Algorithm is how the two algorithms identify the shortcomings of weak learners (eg. decision trees). While the AdaBoost model identifies the shortcomings by using high weight data points, gradient boosting performs the same by using gradients in the loss function (*y=ax+b+e , e needs a special mention as it is the error term)*. The loss function is a measure indicating how good are model's coefficients are at fitting the underlying data. A logical understanding of loss function would depend on what we are trying to optimise. For example, if we are trying to predict the sales prices by using a regression, then the loss function would be based off the error between true and predicted house prices. Similarly, if our goal is to classify credit defaults, then the loss function would be a measure of how good our predictive model is at classifying bad loans. One of the biggest motivations of using gradient boosting is that it allows one to optimise a user specified cost function, instead of a loss function that usually offers less control and does not essentially correspond with real world applications.

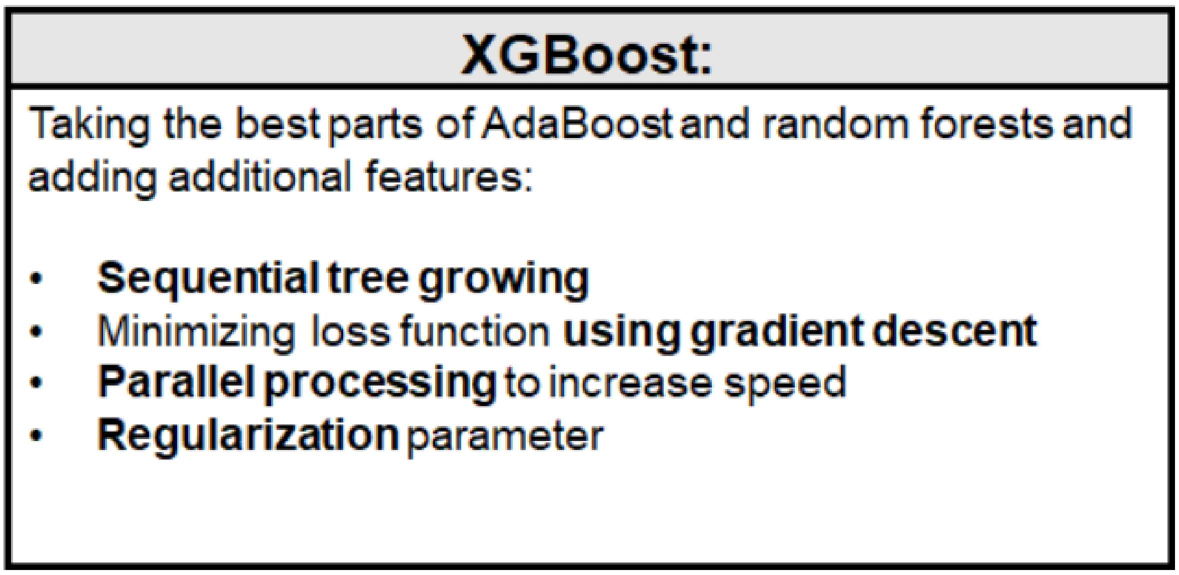
```
# Gradient Boosting
from sklearn.ensemble import GradientBoostingClassifier
clf1 = GradientBoostingClassifier()
# n_estimators = 100 (default)
# loss function = deviance(default) used in Logistic Regression
clf1.fit(X_train, y_train)
y_pred2 = clf1.predict(X_test)
```

XGBoost

XGBoost stands for eXtreme Gradient Boosting. XGBoost is an implementation of gradient boosted decision trees designed for speed and performance. Gradient boosting machines are generally very slow in implementation because of sequential model training. Hence, they are not very scalable. Thus, XGBoost is focused on computational speed and model performance. XGBoost provides:

- **Parallelization** of tree construction using all of your CPU cores during training.

- **Distributed Computing** for training very large models using a cluster of machines.
- **Out-of-Core Computing** for very large datasets that don't fit into memory.
- **Cache Optimization** of data structures and algorithm to make the best use of hardware.



XGBoost:

Taking the best parts of AdaBoost and random forests and adding additional features:

- **Sequential tree growing**
- Minimizing loss function **using gradient descent**
- **Parallel processing** to increase speed
- **Regularization** parameter

The main **advantages** of XGBoost is its lightning speed compared to other algorithms, such as AdaBoost, and its regularization parameter that successfully reduces variance. But even aside from the regularization parameter, this algorithm leverages a learning rate (shrinkage) and subsamples from the features like random forests, which increases its ability to generalize even further. **However**, XGBoost is more difficult to understand, visualize and to tune compared to AdaBoost and random forests. There is a multitude of hyperparameters that can be tuned to increase performance.

```
from xgboost import XGBClassifier
classifier = XGBClassifier()
classifier.fit(X_train, y_train)

y_pred = classifier.predict(X_test)
```