Lecture Notes

Supervised Learning - Boosting

In this module, you grasped the concepts of another supervised learning algorithm called Boosting. We learned some of the popular algorithms, namely AdaBoost, Gradient Boosting and a modification of Gradient Boosting, XGBoost.

Boosting

Boosting was first introduced in 1997 by Freund and Schapire in the popular algorithm, AdaBoost. It was originally designed for classification problems. Since its inception, many new boosting algorithms have been developed those tackle regression problems also and have become famous as they are used in the top solutions of many Kaggle competitions.

Let's start off with the basics of Boosting and move on to the boosting algorithms.

An ensemble is a collection of models which ideally should predict better than individual models. The key idea of boosting is to create an ensemble which makes high errors only on the less frequent data points.

Boosting leverages the fact that we can build a series of models specifically targeted at the data points which have been incorrectly predicted by the other models in the ensemble. If a series of models keep reducing the average error, we will have an ensemble having extremely high accuracy.

Boosting is a way of generating a strong model from a weak learning algorithm.

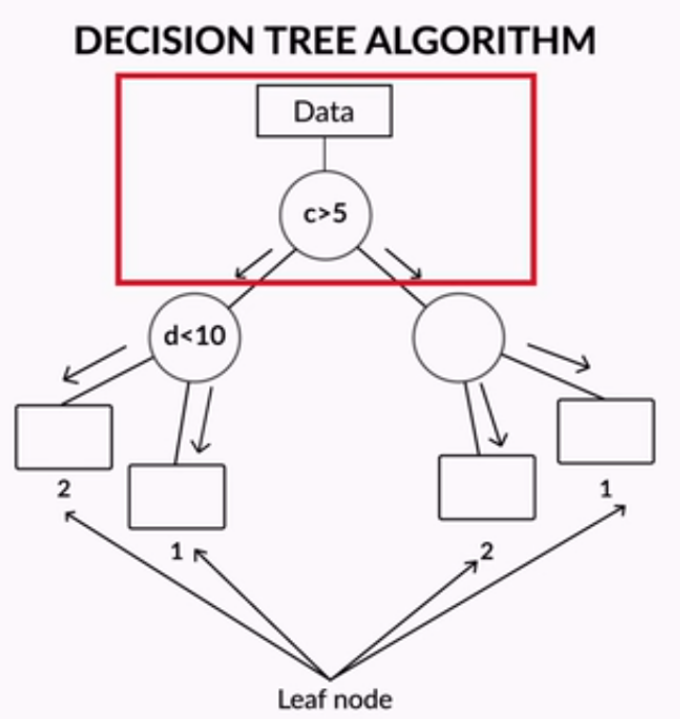
A **weak learning algorithm** produces a model that does marginally better than a random guess. A random guess has a 50% chance of being right. Hence, any such model created by the weak learning algorithm shall have, say 60-70% chance of being correct.

There are other ways we can make the decision tree a weak learner like

1. max_depth: The maximum depth of the tree
2. min_samples_split: The minimum number of samples required to split an internal node
3. min_samples_leaf: The minimum number of samples required to be at a leaf node:
4. min_weight_fraction_leaf: The minimum weighted fraction of the sum total of weights (of all the input samples) required to be at a leaf node
5. max_leaf_nodes: The maximum number of leaf nodes that can be generated

6. min_impurity_decrease: A node will be split if this split induces a decrease of the impurity greater than or equal to this value
7. min_impurity_split: A node will split if its impurity is above the threshold, otherwise it is a leaf

The following picture delivers an example of a weak learner. We have a learning algorithm, decision tree. By restricting the depth of the tree, we can make this decision tree a weak learner. The red box indicates only one split which is also known as a stump.



The final objective is to create a strong model by making an ensemble of such weak models.

Note that SVM, regression and other techniques are algorithms which are used to create models. The function of these algorithms is to minimize the loss.

Here, it is important to understand the loss functions for regression and classification problems are different. Until now, we have defined the error function for a regression setting as the sum of squared difference between the actual and the predicted values while the misclassification rate for a classification problem.

AdaBoost

AdaBoost stands for Adaptive Boosting, was developed by Schapire and Freund, who later on won the 2003 Godel Prize for their work. In this method, every subsequent model is built on a new distribution. This new distribution is created by changing the probability or weights attached with every point. Here, we explain the AdaBoost algorithm using the classification setting in which the target values are +1/-1.

Let us first summarize the notations which we will be using through the lecture notes for the AdaBoost process:

At an iteration t, we have a distribution D_t of the training data T, with the data points having probability $p_{d(t)}(x_i)$ on which we fit a model H_t and then use the results to create a new distribution D_{t+1}. The final model H, we build is an ensemble of all the individual models H_i with weights α_i.

Now, let's look at the objective function:

$$\text{Objective Function} = \min_h E_D[L(h(x_i), y_i)] = \min_h \sum_{i=1}^{n} p_D(x_i).\, L(h(x_i), y_i)$$

The objective function is the expected value of loss function. The AdaBoost process starts off with uniform distribution for all the points but as we move on to make additional models that add to the previous model, the distribution changes and hence, the objective function is expressed in terms of the probabilities of the different data points. The probabilities of the data points on which the next additional model is built are changed in such a way that the algorithm increases the probabilities of the points that were incorrectly classified by the current model and lowers the probabilities of the points that were correctly classified.

There are essentially two steps involved in the AdaBoost scheme of things:

1. Modify the current distribution to create a new distribution to generate a new model

   The probability distribution is modified using the following formula:

$$p_{d(t+1)}(x_i) = \frac{p_{d(t)}(x_i).e^{-\alpha_t y_i h_t(x_i)}}{z_t} \text{ where,}$$

$$z_t = \sum_{i=1}^{n} p_{d(t)}(x_i).\, e^{-\alpha_t y_i h_t(x_i)}$$

We can see that if the prediction is correct, $y_i.h_t(x_i) > 0$ and since $\alpha_t > 0$(will look about this in the next section), $p_{d(t+1)}(x_i) < p_{d(t)}(x_i)$.
Also, when the prediction is wrong, $y_i.h_t(x_i) < 0$ and since $\alpha_t > 0$, $p_{d(t+1)}(x_i) < p_{d(t)}(x_i)$.

2. Calculation of the weights given to each of the models to get the final ensemble

   The formula used for assigning the weights and the error calculation is as follows:

$$\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t}) \text{ where,}$$

$$\epsilon_t = \sum_{i:h_t(x_i \neq y_i)} p_{D(t)}(x_i)$$

   We have discussed earlier that the prediction generated by a weak learner is generally greater than random chance that is 50%. Hence the error shall always be less than 50% or 0.5.

In other words, every subsequent model we build after changing the distribution has a misclassification rate, here the error, $\epsilon_t < 0.5$. With this in mind, we can see that the $\alpha_t > 0$ as the error $\epsilon_t < 0.5$, $((1- \epsilon_t)/ \epsilon_t)$ = *positive* and the *ln(positive) > 0*.

Here is the summary of the AdaBoost algorithm:

1. Initialize the probabilities of the distribution as 1/n where n is the number of data points
2. For t = 0 to T, repeat the following (T is total number of trees):
   1. Fit a tree h_t on the training data using the respective probabilities
   2. Compute

$$\epsilon_t = \sum_{i:h_t(x_i \neq y_i)} p_{D(t)}(x_i)$$

3. Compute

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

4. Update

$$p_{d(t+1)}(x_i) = \frac{p_{d(t)}(x_i).e^{-\alpha_t y_i h_t(x_i)}}{z_t}$$

where,

$$z_t = \sum_{i=1}^{n} p_{d(t)}(x_i).\, e^{-\alpha_t y_i h_t(x_i)}$$

5. Final Model:

$$H = \sum_{t=1}^{T} \alpha_t h_t$$

We can realize here that as we increase the number of trees/ iterations, the error will keep on decreasing.

Before you apply the AdaBoost algorithm, you should specifically remove the Outliers. Since AdaBoost tends to boost up the probabilities of misclassified points and there is a high chance that outliers will be misclassified, it will keep increasing the probability associated with the outliers and make the progress difficult. Some of the ways to remove outliers are:

- Boxplots
- Cook's distance
- Z-score

Gradient Boosting

We shall now look into another popular boosting algorithm - **Gradient Boosting**.

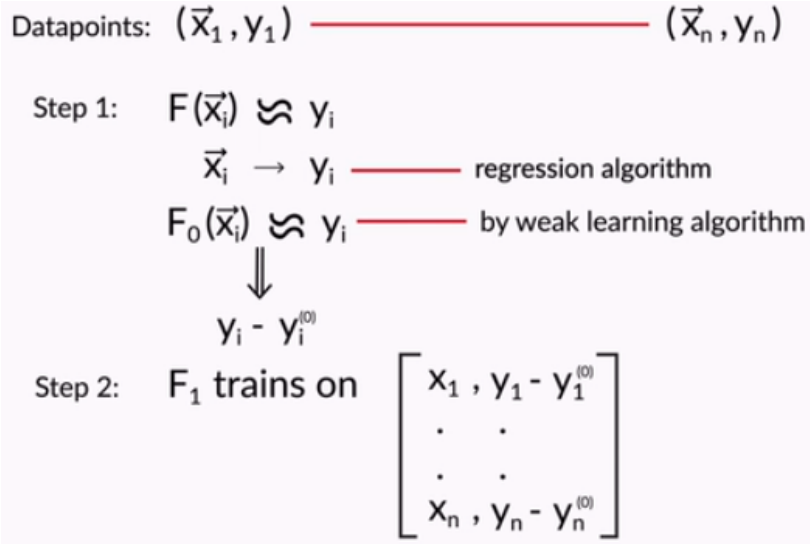
Let's get started with the Gradient Boosting algorithm in a regression setting. So, the loss function we use here is the square loss function.

In other words,

$$\text{Loss function} = (y_i - y_{pred})^2$$

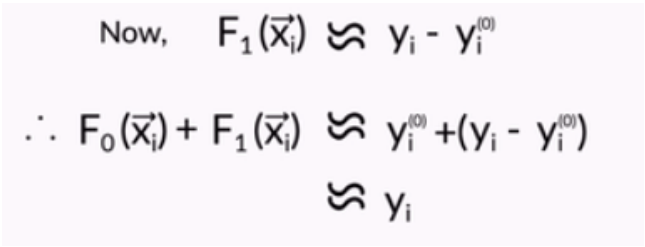
The regression algorithm we use here is the weak learning algorithm.

Let's dive into the algorithm step by step.



We first fit a regression model on the original data (x_i, y_i) to get F_0 which gives the predictions as $y_i^{(0)}$ where i ranges from 1 to n for the number of data points n.

We then find the residue which is $(y_i - y_i^{(0)})$ and fit a regression model on the data $(x_i, (y_i - y_i^{(0)}))$ to F_1.



Before we move on to the next step, let's get an intuition on how modelling on the residuals help us in improving the predictions. We can see in the above image that F_1 predicts a value close to $(y_i - y_i^{(0)})$ and adding this value to the value predicted by the model F_0, $y_i^{(0)}$, we are closing in on the actual or ground truth value y_i.

Step 3: F_2 trains on $\begin{bmatrix} x_1 , & y_1 - [F_0(\vec{x}_1) + F_1(\vec{x}_1)] \\ \vdots & \vdots \\ x_n , & y_n - [F_0(\vec{x}_n) + F_1(\vec{x}_n)] \end{bmatrix}$

The new residual we get is $y_i - (F_0 + F_1)$. Hence, the new model F_2 trains on $(x_i, y_i - (F_0 + F_1))$.
Let's generalize this to an iteration t, where the residual will be $y_i - (F_0 + F_1 + F_2 + \ldots\ldots\ldots + F_t)$. Hence, F_{t+1} will train on $(x_i, y_i - (F_0 + F_1 + F_2 + \ldots\ldots\ldots + F_t))$.

We see that in the gradient boosting algorithm, we keep repeating the following 2 steps on every iteration:

1. Find the residual and fit a new model on the residual
2. Add the new model to the older model and continue the next iteration

Now, what has Gradient got to do here? Let's go through the next couple of paragraphs to understand the same:

We see that the loss function we use is the square loss which is $\sum(y_i - y_{pred})^2$. By differentiating this loss function with respect to y_{pred}, we can see that it becomes the negative of residual, $-2\sum(y_i - y_{pred})$. In other words, our target value, the residual is a step in the direction of the negative gradient of the loss function.

This is essentially the fundamental of gradient descent. In the gradient descent algorithm, a function *G(x)* decreases fastest around a point *a* if one goes along the negative gradient of *G(x)*.

Let us revise the basics of Gradient Descent. For a one a function *z = G(x)*, we follow the following iterative process to arrive at a minima of the function *G(x)*:

$$a_{n+1} = a_n - \lambda \frac{dG(x)}{dx}$$

And if the function *z = G(x, y)*, that is, if it is expressed as a function of 2 independent variables, we have the following formulation:

$$x_{n+1} = x_n - \lambda \frac{\partial G(x,y)}{\partial x}$$

$$y_{n+1} = y_n - \lambda \frac{\partial G(x,y)}{\partial y}$$

We shall now look at the formal setting of the gradient boosting process. Here, there is a slight change in the nomenclature we have used until now.

So here, we generate models F_{t+1} by adding an incremental model h_{t+1} to F_t. Hence we choose a h_{t+1} such that, $L(y, F_t + h_{t+1})$ is minimized. In other words, we select the h_{t+1} such that $L(y, F_t) - L(y, F_t + h_{t+1})$ is the maximum.

This minimization is essentially a gradient descent problem. Hence, by the gradient descent algorithm, to minimize $L(y, F_t)$, we should a step in the negative gradient of the Loss function $L(y, F_t)$.

Mathematically, we take a step λ in the direction $-\frac{\partial L(y,F_t)}{\partial F_t}$. Hence, by comparing it with the regression setting we discussed in the previous page where the negative of the gradients, the residue becomes the target value, the $-\frac{\partial L(y,F_t)}{\partial F_t}$ is the new target value on which we build the incremental model h_{t+1}.

Having understood the above procedure, let's summarize the gradient boosting algorithm:

1. Initialize a crude initial function F_0
2. For t = 0 to T-1 (where T is the number of trees):
    1. Compute the loss function $L(y, F_t)$
    2. Compute the new target values, the negative gradients = $-\frac{\partial L(y,F_t)}{\partial F_t}$ for all data points
    3. Fit a model h_{t+1} on these new target values
    4. Compute the next function $F_{t+1} = F_t + \lambda_t h_{t+1}$
3. Final model is F_T

We stop when we see that the gradients are very close to zero. For a regression setting, this means that when the residuals are very close to zero, we stop iterating. Here, λ_t, known as the **learning rate,** is typically between 0 and 1.

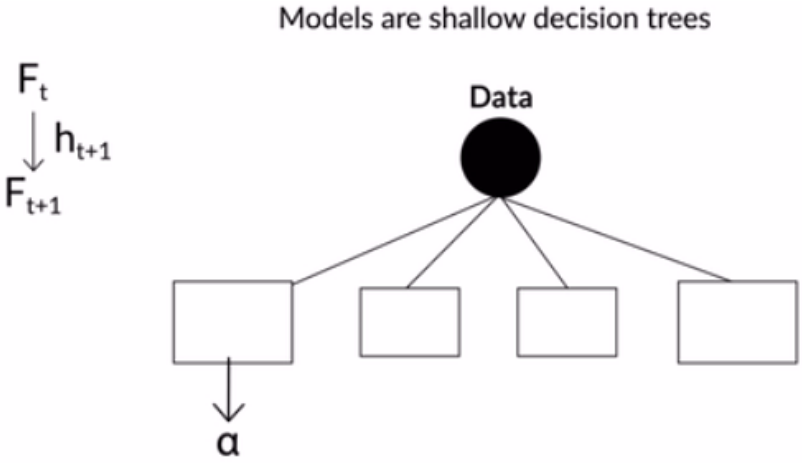
We can use different loss functions depending upon the kind of model we wish to build. AdaBoost uses the exponential loss function. We have also seen that squared error loss functions (for regression) are very sensitive to outliers. To tackle this problem, we can use alternate loss functions, such as the **Huber loss criterion** defined as follows:

$$L(y, F_t) = \begin{cases} (y - F_t)^2 & for\ |\ y - F_t\ |\ <= \delta \\ 2\delta|(y - F_t)| - \delta^2 & otherwise \end{cases}$$

XGBoost

We shall now look at XGBoost, a modification of Gradient Boosting which is widely used in the industry. It was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. XG Boost is an abbreviation for **extreme gradient boosting**. It involves Gradient Boosting on shallow decision trees.

Let's first look at the Gradient Tree Boosting algorithm:



The basic idea behind this algorithm is that we add an incremental model h_{t+1} to F_t to get F_{t+1} as discussed previously. Here, we have used h_{t+1} as a stump for demonstration purpose. The incremental tree has leaf values α_j's which need to be calculated to get the model h_{t+1}. Let's look at the following gradient tree boosting algorithm to understand how we do this:

1. Initialize a crude initial function F0
2. For t = 0 to T-1 where T is the number of trees, repeat the following
    1. Create a loss function $L(y_i, F_t(x_i))$
    2. Compute the new target values, the negative gradients = $-\frac{\partial L(y,F_t)}{\partial F_t}$ for all data points
    3. Fit a shallow decision tree on the above data points to get J_t terminal nodes represented as R_j, j = 1 to J_t
    4. For each terminal node (j = 1 to J_t):
       Find the α_j such that it minimizes $\sum_{x_i \epsilon R_j}^{L} (y_i, F_t(x_i) + \alpha_j)$.
       All these α_j constitute the incremental tree h_{t+1}.
    5. Compute the next function $F_{t+1} = F_t + \lambda_t\ h_{t+1}$
3. F_T is the final output we have.

Please note here that the α_js are the values of the terminal nodes/leaves.
We stop here when we observe that the gradients are close to zero.

The mathematics behind the XGBoost model:
In an ideal machine learning model, the objective function is a sum of Loss function "L" and regularization "Ω"

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

The Additive strategy is:
Fix what have learned and add one tree at a time.

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

$$= \sum_{i=1}^{n} L(y_i, F_{t-1}(x_i) + h_t(x_i)) + \sum_{t=1}^{T} \Omega(h_t) + constant$$

The above described objective function can be approximated using Taylor series expansion and hence can be solved.
The above Gradient Tree boosting algorithm tends to overfit. Hence, XGBoost has a built-in regularization parameter defined as follows:

$$\omega = \gamma J_t + \tau \sum_{j=1}^{J_t} \alpha_j$$

where,

J_t is the number of leaf nodes and α_j are the scores/values of the leaves.

The γ and τ parameters have values greater than 0.

Hence, the target value in the step-2.2 above (in the gradient tree boosting algorithm) is calculated as $-\frac{\partial L(y,F_t)+\omega}{\partial F_t}$.

XGBoost uses pre-sorted algorithm & Histogram-based algorithm for computing the best split.

Now, let's discuss some of the frequently used parameters that are used to regularize the Tree Boosting algorithms like the Gradient Tree Boosting and the XGBoost:

1. λ_t, the **learning rate,** is used to regularize the gradient tree boosting algorithm. λ_t typically varies from 0 to 1. Smaller values of λ_t lead to a larger value of number of trees T (called *n_estimators* in the Python package xgboost). This is because with a slower learning rate, you need a larger number of trees to reach the minima. This in turn leads to longer training time.
   On the other hand, if λ_t is large, we may reach the same point with a lesser number of trees (*n_estimators*), but there's risk that we might miss the minima altogether (i.e. cross over it) because of the long stride we are taking at each iteration.

Note that you shouldn't tune both λ_t and number of trees T together since a high λ_t implies a low value of T and vice-versa.

2. Some other ways of regularization are explicitly specifying the **number of trees T** and doing **subsampling**. **Subsampling** is training the model in each iteration on a fraction of data (like how random forests build each tree). A typical value of subsampling is 0.5 while it ranges from 0 to 1. In random forests, subsampling is critical to ensure diversity among the trees, since otherwise all the trees will start with the same training data and therefore look similar. This is not a big problem in boosting, since each tree is anyway built on the residual and gets a significantly different objective function than the previous one.

Why is XGBoost so good?

1. Parallel Computing: when you run xgboost, by default, it would use all the cores of your laptop/machine enabling its capacity to do parallel computation
2. Regularization: The biggest advantage of xgboost is that it uses regularization and controls the overfitting and simplicity of the model which gives it better performance.
3. Enabled Cross Validation: XGBoost is enabled with internal Cross Validation function
4. Missing Values: XGBoost is designed to handle missing values internally. The missing values are treated in such a manner that if there exists any trend in missing values, it is captured by the model.
5. Flexibility: XGBoost is not just limited to regression, classification, and ranking problems, it supports user-defined objective functions as well. Furthermore, it supports user-defined evaluation metrics as well.

Disclaimer: All content and material on the UpGrad website is copyrighted material, either belonging to UpGrad or its bonafide contributors and is purely for the dissemination of education. You are permitted to access print and download extracts from this site purely for your own education only and on the following basis:

- You can download this document from the website for self-use only.
- Any copies of this document, in part or full, saved to disc or to any other storage medium may only be used for subsequent, self-viewing purposes or to print an individual extract or copy for non-commercial personal use only.
- Any further dissemination, distribution, reproduction, copying of the content of the document herein or the uploading thereof on other websites or use of content for any other commercial/unauthorized purposes in any way which could infringe the intellectual property rights of UpGrad or its contributors, is strictly prohibited.
- No graphics, images or photographs from any accompanying text in this document will be used separately for unauthorised purposes.
- No material in this document will be modified, adapted or altered in any way.
- No part of this document or UpGrad content may be reproduced or stored in any other web site or included in any public or private electronic retrieval system or service without UpGrad's prior written permission.
- Any rights not expressly granted in these terms are reserved.

----------THE END----------