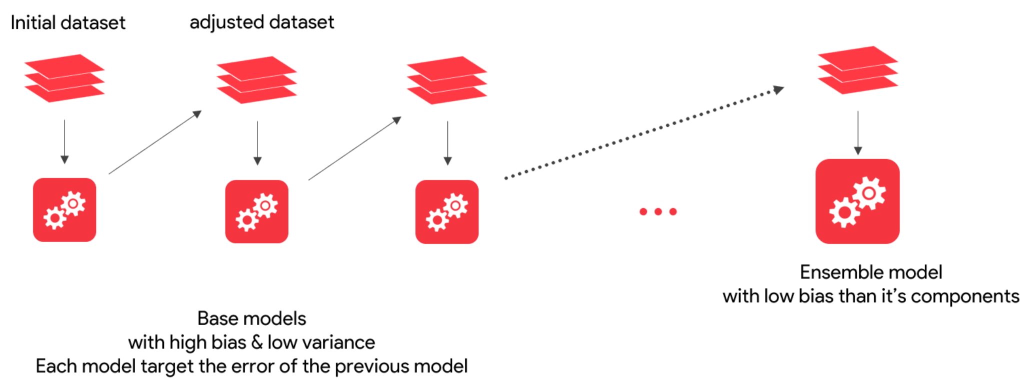
Supervised Learning - Boosting

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.

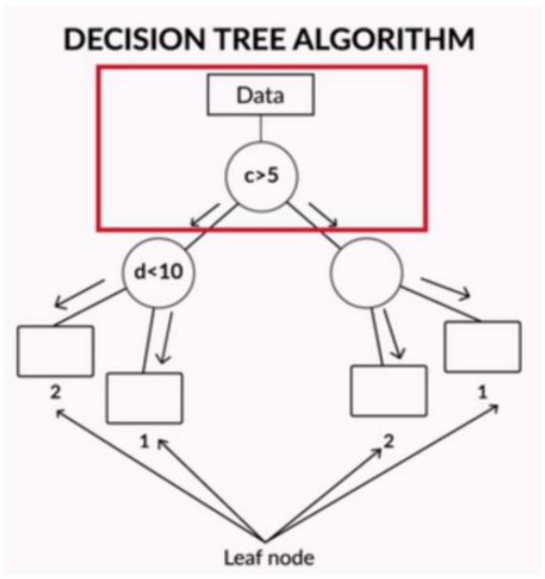

Initial dataset
adjusted dataset
Ensemble model
with low bias than it's components
Base models
with high bias & low variance
Each model target the error of the previous model

Boosting

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.

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.

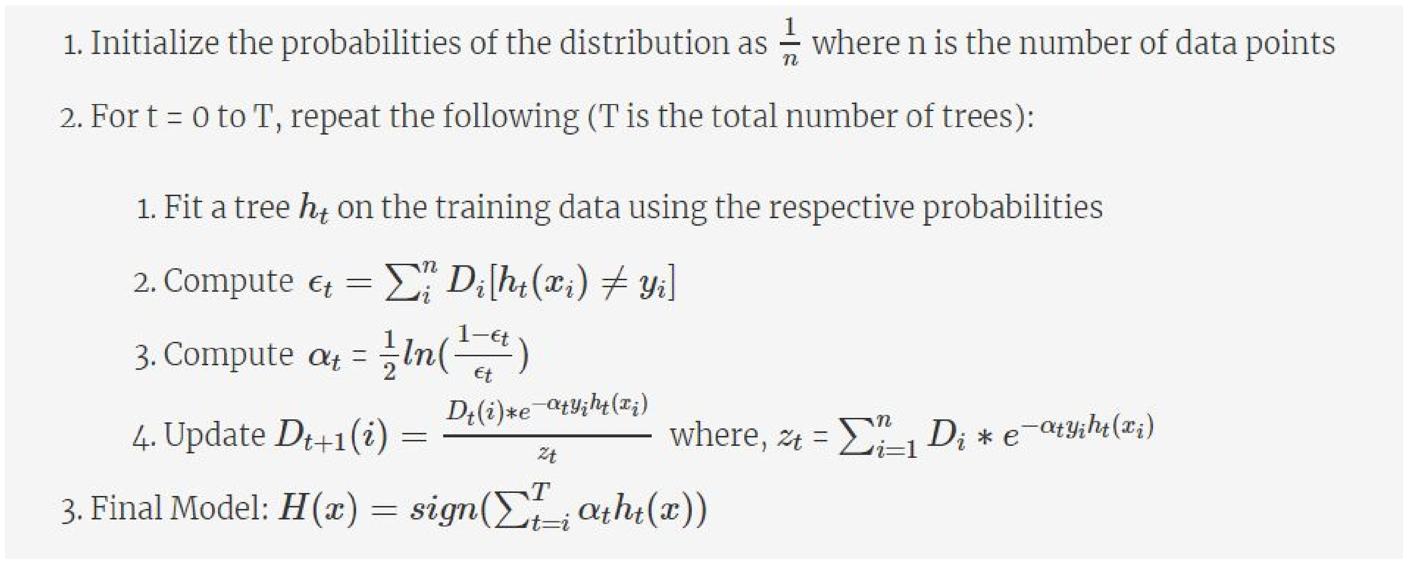
At an iteration t, we have a distribution Dt of the training data T, with the data points having probability pd(t)(xi) on which we fit a model Ht and then use the results to create a new distribution Dt+1. The final model H, we build is an ensemble of all the individual models Hi with weights αi.

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. With each new model, the distribution of the data changes.

There are essentially two steps involved in the AdaBoost algorithm:

1. Modify the current distribution to create a new distribution to generate a new model
2. Calculation of the weights given to each of the models to get the final ensemble

Here is the pseudo-code for Adaboost



```
1. Initialize the probabilities of the distribution as $\frac{1}{n}$ where n is the number of data points
2. For t = 0 to T, repeat the following (T is the total number of trees):
    1. Fit a tree $h_t$ on the training data using the respective probabilities
    2. Compute $\epsilon_t = \sum_i^n D_i[h_t(x_i) \neq y_i]$
    3. Compute $\alpha_t = \frac{1}{2} ln(\frac{1-\epsilon_t}{\epsilon_t})$
    4. Update $D_{t+1}(i) = \frac{D_t(i) * e^{-\alpha_t y_i h_t(x_i)}}{z_t}$ where, $z_t = \sum_{i=1}^n D_i * e^{-\alpha_t y_i h_t(x_i)}$
3. Final Model: $H(x) = sign(\sum_{t=i}^T \alpha_t h_t(x))$
```

We can see that if the prediction is correct, yi.ht(xi) > 0 and since αt > 0(will look about this in the next section), pd(t+1)(xi) < pd(t)(xi).
Also, when the prediction is wrong, yi.ht(xi) < 0 and since αt > 0, pd(t+1)(xi) < pd(t)(xi).

[redacted]

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.

We continue this iteration until

- Low training error is achieved
- A preset number of weak learners have been added

We then make the final prediction by adding up the weighted prediction of every classifier.$H(x)=sign(\sum_{t=i}^{T} \alpha_t h_t(x))$

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

Gradient Boosting like AdaBoost trains many models in a gradual, additive, and sequential manner. But the major difference between the two is how they identify & handle the shortcomings of weak learners through loss functions.

To summarize here are the broader points on how does a GBM learn:

- We build the first weak learner using a sample from the training data; we will consider a decision tree as the weak learner or the base model. It may not necessarily be a stump, can grow a bigger tree but will still be weak i.e. still not be fully grown.
- Then the predictions are made on the training data using the decision tree just built.
- The gradient, in our case the residuals are computed and these residuals are the new response or target values for the next weak learner.
- A new weak learner is built with the residuals as the target values and a sample of observations from the original training data.
- Add the predictions obtained from the current weak learner to the predictions obtained from all the previous weak learners. The predictions obtained at each step are multiplied by the learning rate so that no

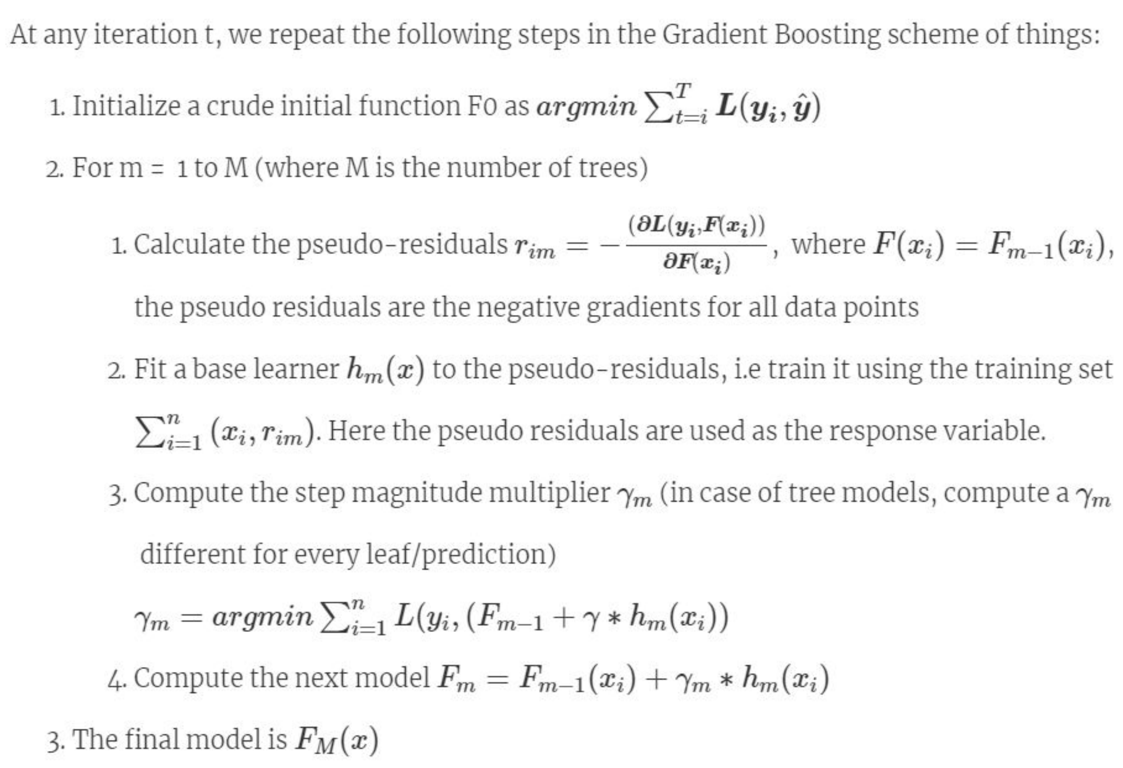
[redacted]

[redacted]

single model makes a huge contribution to the ensemble thereby avoiding overfitting. Essentially, with the addition of each weak learner, the model takes a very small step in the right direction.

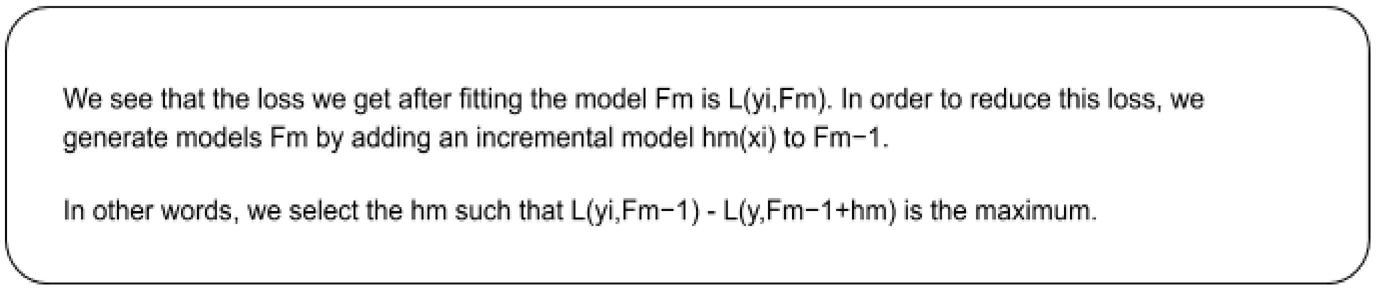
- The next weak learner fits on the residuals obtained till now and these steps are repeated, either for a prespecified number of weak learners or if the model starts overfitting i.e. it starts to capture the niche patterns of the training data.
- GBM makes the final prediction by simply adding up the predictions from all the weak learners (multiplied by the learning rate)

Here is the pseudo-code for Gradient Boosting



At any iteration t, we repeat the following steps in the Gradient Boosting scheme of things:

```
1. Initialize a crude initial function F0 as $argmin \sum_{t=i}^{T} L(y_i, \hat{y})$
2. For m = 1 to M (where M is the number of trees)
    1. Calculate the pseudo-residuals $r_{im} = -\frac{(\partial L(y_i, F(x_i))}{\partial F(x_i)}$, where $F(x_i) = F_{m-1}(x_i)$,
       the pseudo residuals are the negative gradients for all data points
    2. Fit a base learner $h_m(x)$ to the pseudo-residuals, i.e train it using the training set
       $\sum_{i=1}^{n} (x_i, r_{im})$. Here the pseudo residuals are used as the response variable.
    3. Compute the step magnitude multiplier $\gamma_m$ (in case of tree models, compute a $\gamma_m$
       different for every leaf/prediction)
       $\gamma_m = argmin \sum_{i=1}^{n} L(y_i, (F_{m-1} + \gamma * h_m(x_i))$
    4. Compute the next model $F_m = F_{m-1}(x_i) + \gamma_m * h_m(x_i)$
3. The final model is $F_M(x)$
```



We see that the loss we get after fitting the model Fm is L(yi,Fm). In order to reduce this loss, we generate models Fm by adding an incremental model hm(xi) to Fm−1.

In other words, we select the hm such that L(yi,Fm−1) - L(y,Fm−1+hm) is the maximum.

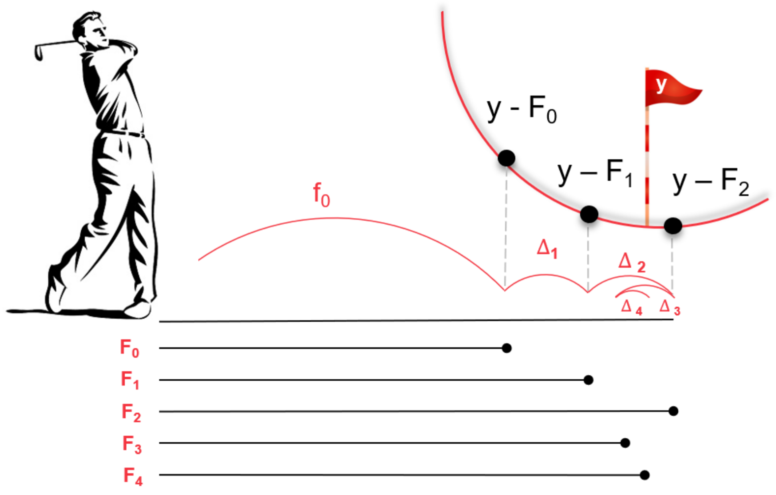
The minimization is essentially a gradient descent problem. Hence, to find an h_m which when added to F_{m-1} reduces the loss, we take a step in the direction where the Loss L(y_i,F_{m-1}) reduces (with respect to F_{m-1}).

Mathematically, we take a step of size γ in the direction −(∂ L(yi,F(xi))/ ∂ F(xi), where F(x_i)=F_{m-1}(x_i).

We stop when we see that the gradients are very close to zero.

[redacted]

[redacted]



Gradient Boosting

XGBoost

Extreme Gradient Boosting (XGBoost) is similar to the gradient boosting framework but more efficient and advanced implementation of the Gradient Boosting algorithm.

It was first developed by Taiqi Chen and became famous in solving the Higgs Boson problem. Due to its robust accuracy, it has been widely used in machine learning competitions as well. It uses more accurate approximations to tune the model and find the best fit.

Let’s have a look at some of the advantages of XGBoost:

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.

[redacted]

[redacted]

Because of parallel processing (speed) and model performance, we can say that XGBoost is gradient boosting on steroids.

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:

λ_t, the **learning rate,** is also known as **shrinkage.** It can be 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 a 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 the risk that we might actually miss the minima altogether (i.e. cross over it) because of the long stride we are taking at each iteration.

Some other ways of regularization are explicitly specifying the **number of trees T** and doing **subsampling**. 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.

Subsampling is training the model in each iteration on a fraction of data (similar to 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 any way built on the residual and gets a significantly different objective function than the previous one.

[redacted]