**LSTM with Grey-wolf optimizer is already implemented in other papers. I want to the accurate novelty**

In this Document I Clearly mentioned algorithm flow , I get input dataset from kaggle, I clearly describe about using algorithm flow and processing steps , using techniques.

I developed your code everything as per the implementation plan.

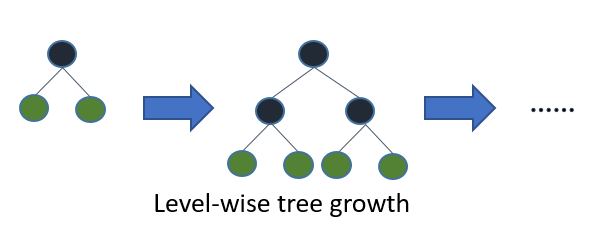
Newly, I applied for LightGBM, Xgboost, and RDBM algorithm

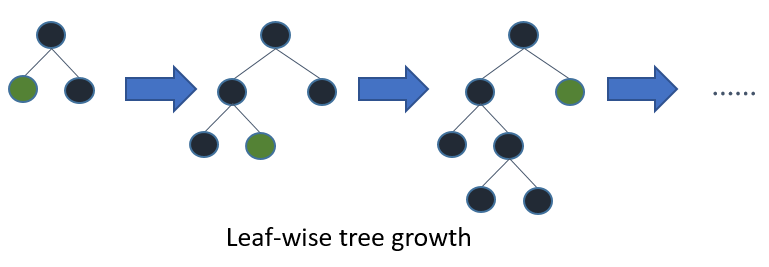
**LightGBM :**

Light GBM is a fast, distributed, high-performance gradient [boosting](https://courses.analyticsvidhya.com/courses/ensemble-learning-and-ensemble-learning-techniques?utm_source=blog&utm_medium=which-algorithm-takes-the-crown-light-gbm-vs-xgboost) framework based on decision tree algorithm, used for ranking, classification and many other machine learning tasks.

Since it is based on decision tree algorithms, it splits the tree leaf wise with the best fit whereas other boosting algorithms split the tree depth wise or level wise rather than leaf-wise. So when growing on the same leaf in Light GBM, the leaf-wise algorithm can reduce more loss than the level-wise algorithm and hence results in much better accuracy which can rarely be achieved by any of the existing boosting algorithms. Also, it is surprisingly very fast, hence the word ‘Light’.

Before is a diagrammatic representation by the makers of the Light GBM to explain the difference clearly.





Leaf wise tree growth in Light GBM.

Leaf wise splits lead to increase in complexity and may lead to overfitting and it can be overcome by specifying another parameter max-depth which specifies the depth to which splitting will occur.

Below, we will see the steps to install Light GBM and run a model using it. We will be comparing the results with XGBOOST results to prove that you should take Light GBM in a ‘LIGHT MANNER’.

Let us look at some of the advantages of Light GBM.

## Advantages of Light GBM

1. ****Faster training speed and higher efficiency****: Light GBM use histogram based algorithm i.e it buckets continuous feature values into discrete bins which fasten the training procedure.
2. ****Lower memory usage:**** Replaces continuous values to discrete bins which result in lower memory usage.
3. ****Better accuracy than any other boosting algorithm:**** It produces much more complex trees by following leaf wise split approach rather than a level-wise approach which is the main factor in achieving higher accuracy. However, it can sometimes lead to overfitting which can be avoided by setting the max\_depth parameter.
4. ****Compatibility with Large Datasets:**** It is capable of performing equally good with large datasets with a significant reduction in training time as compared to XGBOOST.
5. ****Parallel learning supported.****

## Important Parameters of light GBM

* ****task**** : default value = train ; options = train , prediction ; Specifies the task we wish to perform which is either train or prediction.
* ****application****: default=regression, type=enum, options= options :

regression : perform regression task

binary : Binary clssification

multiclass: Multiclass Classification

lambdarank : lambdarank application

* ****data****: type=string; training data , LightGBM will train from this data
* ****num\_iterations****: number of boosting iterations to be performed ; default=100; type=int
* ****num****\_****leaves**** : number of leaves in one tree ; default = 31 ; type =int
* ****device**** : default= cpu ; options = gpu,cpu. Device on which we want to train our model. Choose GPU for faster training.
* ****max\_depth****: Specify the max depth to which tree will grow. This parameter is used to deal with overfitting.
* ****min\_data\_in\_leaf****: Min number of data in one leaf.
* ****feature\_fraction****: default=1 ; specifies the fraction of features to be taken for each iteration
* ****bagging\_fraction****: default=1 ; specifies the fraction of data to be used for each iteration and is generally used to speed up the training and avoid overfitting.
* ****min\_gain\_to\_split****: default=.1 ; min gain to perform splitting
* ****max\_bin**** : max number of bins to bucket the feature values.
* ****min\_data\_in\_bin**** : min number of data in one bin
* ****num\_threads****: default=OpenMP\_default, type=int ;Number of threads for Light GBM.
* ****label**** : type=string ; specify the label column
* ****categorical\_feature**** : type=string ; specify the categorical features we want to use for training our model
* ****num\_class****: default=1 ; type=int ; used only for multi-class classification

## The Power of XGBoost

The beauty of this powerful algorithm lies in its scalability, which drives fast learning through parallel and distributed computing and offers efficient memory usage.

It’s no wonder then that CERN recognized it as the best approach to classify signals from the Large Hadron Collider. This particular challenge posed by CERN required a solution that would be scalable to process data being generated at the rate of 3 petabytes per year and effectively distinguish an extremely rare signal from background noises in a complex physical process. XGBoost emerged as the most useful, straightforward and robust solution.

Now, let’s deep dive into the inner workings of XGBoost.

## Why ensemble learning?

XGBoost is an [ensemble learning](https://courses.analyticsvidhya.com/courses/ensemble-learning-and-ensemble-learning-techniques?utm_source=blog&utm_medium=an-end-to-end-guide-to-understand-the-math-behind-xgboost) method. Sometimes, it may not be sufficient to rely upon the results of just one machine learning model. Ensemble learning offers a systematic solution to combine the predictive power of multiple learners. The resultant is a single model which gives the aggregated output from several models.

The models that form the ensemble, also known as base learners, could be either from the same learning algorithm or different learning algorithms. ****Bagging and boosting are two widely used ensemble learners****. Though these two techniques can be used with several statistical models, the most predominant usage has been with decision trees.

Let’s briefly discuss bagging before taking a more detailed look at the concept of boosting.

### Bagging

While decision trees are one of the most easily interpretable models, they exhibit highly variable behavior. Consider a single training dataset that we randomly split into two parts. Now, let’s use each part to train a decision tree in order to obtain two models.

When we fit both these models, they would yield different results. Decision trees are said to be associated with high variance due to this behavior. Bagging or boosting aggregation helps to reduce the variance in any learner. Several decision trees which are generated in parallel, form the base learners of bagging technique. Data sampled with replacement is fed to these learners for training. The final prediction is the averaged output from all the learners.

### Boosting

In boosting, the trees are built sequentially such that each subsequent tree aims to reduce the errors of the previous tree. Each tree learns from its predecessors and updates the residual errors. Hence, the tree that grows next in the sequence will learn from an updated version of the residuals.

The base learners in boosting are weak learners in which the bias is high, and the predictive power is just a tad better than random guessing. Each of these weak learners contributes some vital information for prediction, enabling the boosting technique to produce a strong learner by effectively combining these weak learners. The final strong learner brings down both the bias and the variance.

****In contrast to bagging techniques like Random Forest, in which trees are grown to their maximum extent, boosting makes use of trees with fewer splits****. Such small trees, which are not very deep, are highly interpretable. Parameters like the number of trees or iterations, the rate at which the gradient boosting learns, and the depth of the tree, could be optimally selected through validation techniques like k-fold cross validation. ****Having a large number of trees might lead to overfitting. So, it is necessary to carefully choose the stopping criteria for boosting.****

The boosting [ensemble technique](https://courses.analyticsvidhya.com/courses/ensemble-learning-and-ensemble-learning-techniques?utm_source=blog&utm_medium=an-end-to-end-guide-to-understand-the-math-behind-xgboost) consists of three simple steps:

* An initial model F0 is defined to predict the target variable y. This model will be associated with a residual (y – F0)
* A new model h1 is fit to the residuals from the previous step
* Now, F0 and h1 are combined to give F1, the boosted version of F0. The mean squared error from F1 will be lower than that from F0:

IMG_256

To improve the performance of F1, we could model after the residuals of F1 and create a new model F2:

IMG_257

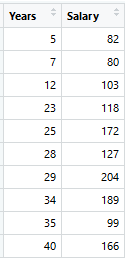
This can be done for *‘m’*iterations, until residuals have been minimized as much as possible:

IMG_258

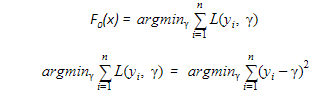
Here, the additive learners do not disturb the functions created in the previous steps. Instead, they impart information of their own to bring down the errors.

## Demonstrating the Potential of Boosting

Consider the following data where the years of experience is predictor variable and salary (in thousand dollars) is the target. Using regression trees as base learners, we can create an [ensemble model](https://courses.analyticsvidhya.com/courses/ensemble-learning-and-ensemble-learning-techniques?utm_source=blog&utm_medium=an-end-to-end-guide-to-understand-the-math-behind-xgboost) to predict the salary. For the sake of simplicity, we can choose square loss as our loss function and our objective would be to minimize the square error.



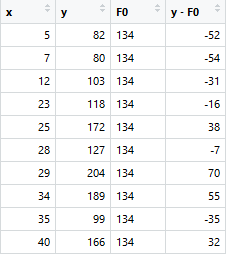
As the first step, the model should be initialized with a function F0(x). F0(x) should be a function which minimizes the loss function or MSE (mean squared error), in this case:



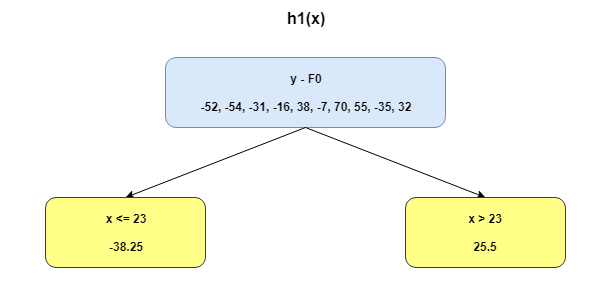
Taking the first differential of the above equation with respect to γ, it is seen that the function minimizes at the mean i=1nyin. So, the boosting model could be initiated with:

IMG_257

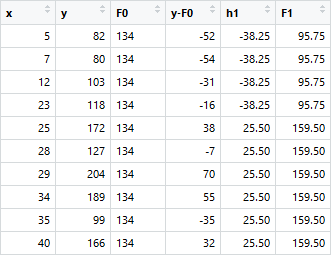
F0(x) gives the predictions from the first stage of our model. Now, the residual error for each instance is (yi – F0(x)).



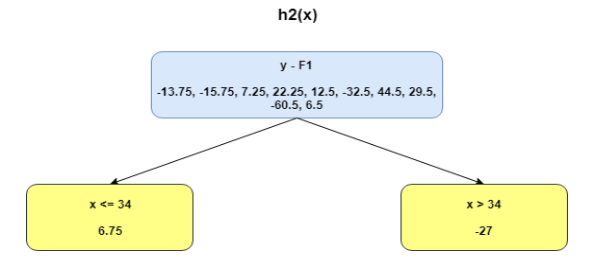
We can use the residuals from F0(x) to create h1(x). h1(x) will be a regression tree which will try and reduce the residuals from the previous step. The output of h1(x) won’t be a prediction of y; instead, it will help in predicting the successive function F1(x) which will bring down the residuals.

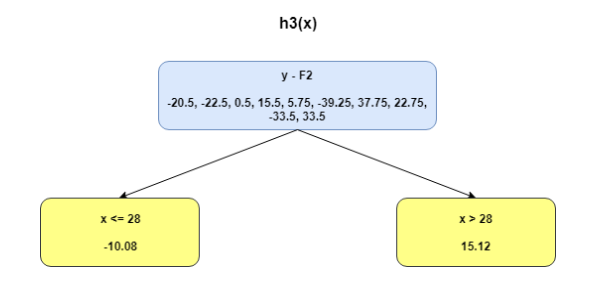


The additive model h1(x) computes the mean of the residuals (y – F0) at each leaf of the tree. The boosted function F1(x) is obtained by summing F0(x) and h1(x). This way h1(x) learns from the residuals of F0(x) and suppresses it in F1(x).



This can be repeated for 2 more iterations to compute h2(x) and h3(x). Each of these additive learners, hm(x), will make use of the residuals from the preceding function, Fm-1(x).







## Using gradient descent for optimizing the loss function

In the case discussed above, MSE was the loss function. The mean minimized the error here. When MAE (mean absolute error) is the loss function, the median would be used as F0(x) to initialize the model. A unit change in y would cause a unit change in MAE as well.

For MSE, the change observed would be roughly exponential. Instead of fitting hm(x) on the residuals, fitting it on the gradient of loss function, or the step along which loss occurs, would make this process generic and applicable across all loss functions.

Gradient descent helps us minimize any differentiable function. Earlier, the regression tree for hm(x) predicted the mean residual at each terminal node of the tree. In gradient boosting, the average gradient component would be computed.

For each node, there is a factor γ with which hm(x) is multiplied. This accounts for the difference in impact of each branch of the split. Gradient boosting helps in predicting the optimal gradient for the additive model, unlike classical gradient descent techniques which reduce error in the output at each iteration.

The following steps are involved in gradient boosting:

* F0(x) – with which we initialize the boosting algorithm – is to be defined:

IMG_256

* The gradient of the loss function is computed iteratively:

IMG_257

* Each hm(x) is fit on the gradient obtained at each step
* The multiplicative factor γm for each terminal node is derived and the boosted model Fm(x) is defined:

IMG_258

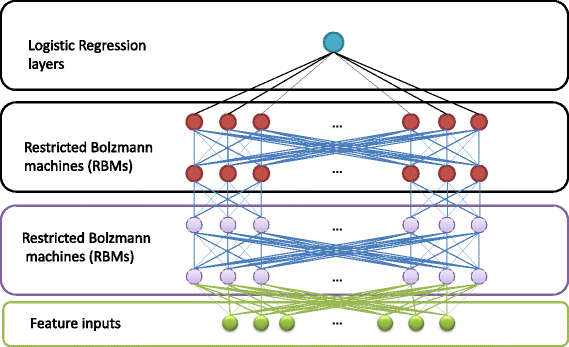
## Unique features of XGBoost

XGBoost is a popular implementation of gradient boosting. Let’s discuss some features of XGBoost that make it so interesting.

* **Regularization:**XGBoost has an option to penalize complex models through both L1 and L2 regularization. Regularization helps in preventing overfitting
* **Handling sparse data:**Missing values or data processing steps like one-hot encoding make data sparse. XGBoost incorporates a sparsity-aware split finding algorithm to handle different types of sparsity patterns in the data
* **Weighted quantile sketch:** Most existing tree based algorithms can find the split points when the data points are of equal weights (using quantile sketch algorithm). However, they are not equipped to handle weighted data. XGBoost has a distributed weighted quantile sketch algorithm to effectively handle weighted data
* **Block structure for parallel learning:**For faster computing, XGBoost can make use of multiple cores on the CPU. This is possible because of a block structure in its system design. Data is sorted and stored in in-memory units called blocks. Unlike other algorithms, this enables the data layout to be reused by subsequent iterations, instead of computing it again. This feature also serves useful for steps like split finding and column sub-sampling
* **Cache awareness:** In XGBoost, non-continuous memory access is required to get the gradient statistics by row index. Hence, XGBoost has been designed to make optimal use of hardware. This is done by allocating internal buffers in each thread, where the gradient statistics can be stored
* **Out-of-core computing:**This feature optimizes the available disk space and maximizes its usage when handling huge datasets that do not fit into memory

1. **RDBN- Proposed Algorithm :**

# Deep Belief Networks



Before understanding what a DBN is, we will first look at RBMs,****Restricted Boltzmann Machines****.

## Restricted Boltzmann Machines:

what a factor analysis is, RBMs can be considered as a binary version of Factor Analysis. So instead of having a lot of factors deciding the output, we can have binary variable in the form of 0 or 1.

***For Example:*** *If you a read a book, and then judge that book on the scale of two: that is either you like the book or you do not like the book. In this kind of scenarios we can use RBMs, which will help us to determine the reason behind us making those choices.*

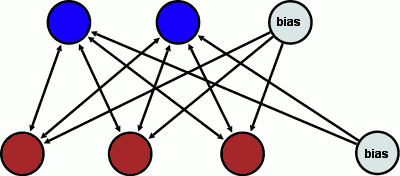
RBMs take a probabilistic approach for Neural Networks, and hence they are also called as ****Stochastic Neural Networks.****

If we decompose RBMs, they have three parts:-

1. One Input Layer aka ****Visible Unit****
2. One Hidden Layer aka ****Hidden Unit****
3. One ****Bias Unit****

In the example that I gave above, visible units are nothing but whether you like the book or not. Hidden Unit helps to find what makes you like that particular book. Bias is added to incorporate different kinds of properties that different books have.

Let us visualize the RBMs:



*Red is Visible Unit, Blue is Hidden Unit*

Let us look at the steps that RBN takes to learn the decision making process:-

1. Compute ****Activation Energy****
2. Calculate ****Sigmoid****of Activation Energy
3. This will give us a probability. Using this probability Hidden unit can****turn on or turn off**** any of the nodes in visible unit.

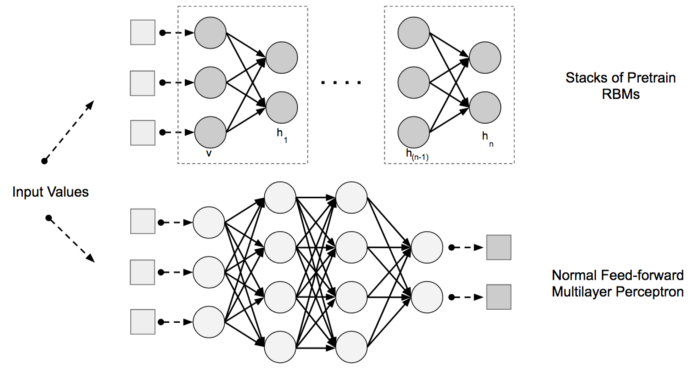
Now that we have basic idea of Restricted Boltzmann Machines, let us move on to Deep Belief Networks

## Deep Belief Networks

DBNs have two phases:-

1. Pre-train Phase
2. Fine-tune Phase

Pre-train phase is nothing but multiple layers of RBNs, while Fine Tune Phase is a feed forward neural network. Let us visualize both the steps:-

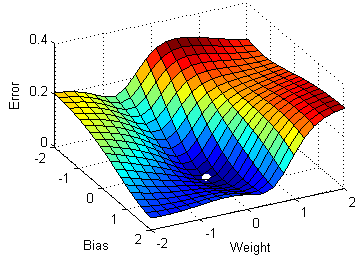


How DBNs work?

1. Find the features of Visible Units using Contrastive Divergence Algorithm
2. Find the Hidden Unit Features, and the feature of features found in above step
3. When the hidden layer learning phase is over, we call it as a trained DBN

In this Deep Belief Neural Network used to SGD Optimizer:

SGD algorithm is an extension of the Gradient Descent and it overcomes some of the disadvantages of the GD algorithm. Gradient Descent has a disadvantage that it requires a lot of memory to load the entire dataset of n-points at a time to compute the derivative of the loss function. **In the** **SGD algorithm derivative is computed taking one point at a time.**



SGD performs a parameter update for each training example **x(i)** and label**y(i)**:

**θ = θ − α⋅∂(J(θ;x(i),y(i)))/∂θ**

where **{x(i) ,y(i)}** are the training examples.

To make the training even faster we take a Gradient Descent step for each training example.

**Advantage**:

Memory requirement is less compared to the GD algorithm as the derivative is computed taking only 1 point at once.