Boosting Machines

In previous module we saw models based on bagging; random forests and extraTrees where each individual model was independent and eventual prediction of the ensemble of these models was a simple majority vote or average depending on whether the problem was of classification of regression.

The randomness in the process, helped the model become less affected by noise and more generalizable. However this bagging did not really help in underlying models to become better at extracting more complex patterns.

Boosting machines go that extra step, in modern implementation, both the ideas; using randomness to make models generalizable and boosting[which we'll study in few moments] are used . You can consider boosting machines to be more powerful than bagging models . However that comes with downside of them being prone to over-fitting in theory. We'll learn about Extreme Gradient Boosting (Xgboost) which goes one step further and adds the element of regularization to boosting machines and some other radical changes to become one of the most successful Machine Learning Algorithms in the recent history.

Just like bagging , boosting machines also are made up of multiple individual models . However in boosting machines , individual models are built sequentially and eventual model is summation of individual models not the average . Formally for a boosting machine, our $F(X_i)$ or more formally with `t` individual models , $F_t(X_i)$ is written as :

$$F_t(X_i) = f_1(X_i) + f_2(X_i) \ldots\ldots\ldots\ldots + f_t(X_i)$$

where $f_t(X_i)$ represents individual models . As mentioned earlier , these individual models are built sequentially. Patterns which could not be captured by the model so far become target for the next guy. Its fairly intuitive to understand in context of regression model [The one with the continuous numeric target] .

for f_1 , target is simply the original outcome y_i, but as we go forward , the target simply becomes errors remaining so far :

$$f_1 \rightarrow y_i$$
$$f_2 \rightarrow (y_i - f_1)$$
$$f_3 \rightarrow (y_i - f_1 - f_2)$$
$$\vdots$$
$$f_{t+1} \rightarrow (y_i - F_t)$$

by the looks of it this looks like a recipe for disaster , certainly overfitting . Remember that we are trying to reduce the error here on training data, and if we keep on fitting models on the residuals , eventually we'll start to severly overfit.

Why Weak-Learners

In order to avoid the over-fitting , we can chose our individual models to be weak-learners. Models which are incapable of over-fitting themselves. In fact the ones which are not good models taken individually. Individually they are only capable of capturing most strong and hence reliable patterns. And upon boosting , such consistent patterns extracted (though with changing target for each) taken together make for a very strong and yet somewhat robust to over-fitting model.

What weak-learners

In theory, any kind of base model can be made a weak learner. Few examples :

- Linear Regression which makes use of say only $(\frac{1}{10})^{th}$ of the variables at each step
- Linear Regression with very very high penalty [Large value of λ in L1/L2 regularization]
- Tree Stumps : Very Shallow Decision Trees [low depth]

In Practice however , `Tree Stumps` are popular and you will find them implemented almost everywhere.

Remeber that decision trees start to overfit [extract niche patterns from the training data] when we grow them too large and start to pick partition rules from smaller and smaller chunk of the training data. Shallow decision trees do not reach to that point and hence are incapable of individually overfitting .

Gradient Boosting Machines

So far we haven't formally discussed how do we go about building this so called boosted model. Although the example taken above in context of regression models, seem like a no-brainer. However, thats only good for building intuition in the beginning. It fails to generalize pretty fast with slight changes in the cost functions . Gradient Boosting Machines combine the idea of `Gradient Descent` and `Boosting` to give a generic method which works with any cost/loss formulation.

Lets see how do we transition from Gradient Decent for parameters to Gradient Boosting Machines . If you recall, idea behind gradient decent was that in each iteration we update the parameters by changing [updating] them like this :

$$\beta \rightarrow \beta - \eta * \nabla C$$

or

$$\Delta\beta = -\eta * \nabla C$$

- Parameters βs are getting updated and the gradient of the cost is taken w.r.t. to parameters
- In context of boosting machines however , the Model itself is getting updated. in every iteration $F_{t+1} = F_t + f_{t+1}$., model is updated by f_{t+1} . This update, using the `Gradient Descent` will be equal to $-\eta * \nabla C$, however the gradient here will be

taken w.r.t. to what is being updated that is F_t.

- Before we are able to take the derivative/gradient of the loss w.r.t. F_t, we'll need to write the cost/loss in terms of F_t
- f_{t+1} being equal to $-\eta * \nabla C$ doesn't intuitively make sense . f_{t+1} is after all a model [Tree stump]. What does it mean for a model to be `equal` to something. It means that while we are building the model we'll take $-\eta * \nabla C$ as the target for the model .

Formally if we write our cost/loss =$\sum C(y_i, F_t)$

where

$$F_t = f_1 + f_2 + \cdots + f_t$$

then

$$f_{t+1} \rightarrow -\eta * \frac{\delta C}{\delta F_t} \qquad \cdots (1)$$

GBM for Regression

for regression if you recall our traditional loss is nothing but squared error

$$C = (y_i - F_t)^2$$

if we take the derivation of this w.r.t. F_t, we get :

$$\frac{\delta C}{\delta F_t} = -2 * (y_i - F_t)$$

putting this back in $\cdots (1)$, we get

$$f_{t+1} \rightarrow \eta * (y_i - F_t)$$

This simply means that every next shallow decision tree will take small fraction of the remaining error as its target. η is there to ensure that any one individual model doesn't end up contributing too heavily towards the eventual prediction.

GBM for Classification

If you revisit your `Intro to ML` chapter and look up the discussion on cost/loss for classfication, you'll find this formulation for loss

$$C = -[y_i * F_t - log(1 + e^{F_t})]$$

lets take its derivative w.r.t. F_t

$$\frac{\delta C}{\delta F_t} = -[y_i - \frac{e^{F_t}}{1 + e^{F_t}}]$$

$$= -[y_i - \frac{1}{1 + e^{-F_t}}]$$

in the same section you would have found that probability p_i is actually represented as this :

$$p_i = \frac{1}{1 + e^{-F_t}}$$

using the same we can write :

$$\frac{\delta C}{\delta F_t} = -[y_i - p_i]$$

putting this back in $\cdots (1)$, we get

$$f_{t+1} \rightarrow \eta * (y_i - p_i)$$

Couple things that you need to realize about gbm for classification :

- Each individual shallow tree is a regression tree [not a classification tree] as the target for them is a continuous numeric number
- p_i is not a simple summation of probability scores given by individual shallow tree models. No.
- $F_{t+1} = F_t + f_{t+1}$ and $p_{t+1} = \frac{1}{1+e^{-(F_t + f_{t+1})}}$

You should realize that this process can be done for any alternate cost/loss formulation as well. All that we need to do is that to write the cost w.r.t. model itself and then define its derivative.

Next we look at parameters to tune in GBM

GBM parameters

- n_estimators [default = 100]:

Number of boosted individual models . there is no upper limit.

- learning_rate [default=0.1]

learning rate shrinks the contribution of each tree by `learning_rate`.There is a trade-off between learning_rate and n_estimators. A small learning rate will require large number of n_estimator.

- max_depth [default=3]

depth of the individual tree models . High number here will lead to complex/overfit model

- min_samples_split[default=2]

The minimum number of samples required to split an internal node:

```
- If int, then consider `min_samples_split` as the minimum number.
- If float, then `min_samples_split` is a fraction and
  `ceil(min_samples_split * n_samples)` are the minimum
  number of samples for each split.
```

- min_samples_leaf[default = 1]

The minimum number of samples required to be at a leaf node. A split point at any depth will only be considered if it leaves at least `min_samples_leaf` training samples in each of the left and right branches. This may have the effect of smoothing the model, especially in regression.

```
- If int, then consider `min_samples_leaf` as the minimum number.
- If float, then `min_samples_leaf` is a fraction and
  `ceil(min_samples_leaf * n_samples)` are the minimum
  number of samples for each node.
```

Note : Both `min_samples_leaf` and `min_samples_split` however might be rendered useless as we keep max_depth low to ensure individual models remaining weak learners. The situation for all practical purposes might never arise where these two become relevant due to shallow trees

- subsample [default =1]

The fraction of samples to be used for fitting the individual base learners. If smaller than 1.0 this results in Stochastic Gradient Boosting.

- max_features [default = None]

The number of features to consider when looking for the best split:

```
- If int, then consider `max_features` features at each split.
- If float, then `max_features` is a fraction and
  `int(max_features * n_features)` features are considered at each
  split.
- If "auto", then `max_features=sqrt(n_features)`.
- If "sqrt", then `max_features=sqrt(n_features)`.
- If "log2", then `max_features=log2(n_features)`.
- If None, then `max_features=n_features`.
```

Regression problem with GBM

`Problem Statement` : We are going to build a model for predicting bike sharing numbers given various weather factors and other characteristic for the days. Lets read the data

```
import pandas as pd
import numpy as np
file=r'/Users/lalitsachan/Dropbox/0.0 Data/Cycle_Shared.csv'
bike=pd.read_csv(file)
bike.head()
```

	instant	dteday	season	yr	mnth	holiday	weekday	workingday	weathersit	temp	atemp	hum
0	1	2011-01-01	1	0	1	0	6	0	2	0.344167	0.363625	0.805833
1	2	2011-01-02	1	0	1	0	0	0	2	0.363478	0.353739	0.696087
2	3	2011-01-03	1	0	1	0	1	1	1	0.196364	0.189405	0.437273
3	4	2011-01-04	1	0	1	0	2	1	1	0.200000	0.212122	0.590435
4	5	2011-01-05	1	0	1	0	3	1	1	0.226957	0.229270	0.436957

```
bike.shape
```

```
(731, 16)
```

- instant : will be dropped because of being an id variable
- dteday : we'll extract day of month from this and drop the column
- season, mnth, weekday, weathersit : will be treated as categorical variable
- casual,registered : cnt our target variable is a simple sum of these
- yr : will be dropped as future instances will not take the same value so no point in treating this as categorical variable and it takes too few values for it to be treated like a numeric one

```
bike.drop(['yr','instant','casual','registered'],axis=1,inplace=True)
```

```
bike['dteday']=pd.to_datetime(bike['dteday'])
bike['day']=bike['dteday'].dt.month
del bike['dteday']
```

```
for col in ['season', 'mnth', 'weekday', 'weathersit']:

    k=bike[col].value_counts()
    cats=k.index[k>30][:-1]
    for cat in cats :
        name=col+'_'+str(cat)
        bike[name]=(bike[col]==cat).astype(int)

    del bike[col]

```

```
x_train=bike.drop('cnt',1)
y_train=bike['cnt']
```

```
gbm_params={'n_estimators':[50,100,200],
           'learning_rate': [0.01,.05,0.1,0.4,0.8,1],
            'max_depth':[1,2,3,4,5,6],
            'subsample':[0.5,0.8,1],
            'max_features':[5,10,15,20,28]
           }
# Note : keep in mind that this dataset is too small and ideally we should avoid complex models
```

```
from sklearn.ensemble import GradientBoostingRegressor
from sklearn.model_selection import RandomizedSearchCV
```

```
model=GradientBoostingRegressor()
random_search=RandomizedSearchCV(model,scoring='neg_mean_absolute_error',
                                 param_distributions=gbm_params,
                                 cv=10,n_iter=10,
                                 n_jobs=-1,verbose=False)
```

```
random_search.fit(x_train,y_train)
```

```
def report(results, n_top=3):
    for i in range(1, n_top + 1):
        candidates = np.flatnonzero(results['rank_test_score'] ==i)
        for candidate in candidates:
            print("Model with rank: {0}".format(i))
            print("Mean validation score: {0:.5f} (std: {1:.5f})".format(
                  results['mean_test_score'][candidate],
                  results['std_test_score'][candidate]))
            print("Parameters: {0}".format(results['params'][candidate]))
            print("")
```

```
report(random_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: -1569.07622 (std: 664.47666)
Parameters: {'subsample': 0.8, 'n_estimators': 50, 'max_features': 28, 'max_depth': 5,
'learning_rate': 0.01}

Model with rank: 2
Mean validation score: -1645.59695 (std: 197.54250)
Parameters: {'subsample': 1, 'n_estimators': 50, 'max_features': 20, 'max_depth': 3, 'learning_rate':
0.05}
```

```
Model with rank: 3
Mean validation score: -1658.20089 (std: 169.47697)
Parameters: {'subsample': 0.5, 'n_estimators': 100, 'max_features': 10, 'max_depth': 2,
'learning_rate': 0.05}

Model with rank: 4
Mean validation score: -1775.09852 (std: 213.59569)
Parameters: {'subsample': 1, 'n_estimators': 200, 'max_features': 15, 'max_depth': 6,
'learning_rate': 0.4}

Model with rank: 5
Mean validation score: -1787.23971 (std: 198.50882)
Parameters: {'subsample': 0.5, 'n_estimators': 200, 'max_features': 5, 'max_depth': 4,
'learning_rate': 0.05}
```

This gives us our best model parameters as well as its performance . We can further extract variable importance and make partial dependence plots here also as we did for randomForest results in the earlier module

Classification problem with GBM

`Problem Statement` : Given the demographic details , we'll a model to predict whether a person's income is greater than $50K or not

```
# many imports will not be explicitly done now onwards here
# because they have already happened earlier in this chapter
# if you are working on this in a fresh notebook
# do include those imports
file = r'/Users/lalitsachan/Dropbox/0.0 Data/census_income.csv'
cd=pd.read_csv(file)
cd.head()
```

	age	workclass	fnlwgt	education	education.num	marital.status	occupation	relationship	race	sex	capita
0	39	State-gov	77516	Bachelors	13	Never-married	Adm-clerical	Not-in-family	White	Male	2174
1	50	Self-emp-not-inc	83311	Bachelors	13	Married-civ-spouse	Exec-managerial	Husband	White	Male	0
2	38	Private	215646	HS-grad	9	Divorced	Handlers-cleaners	Not-in-family	White	Male	0
3	53	Private	234721	11th	7	Married-civ-spouse	Handlers-cleaners	Husband	Black	Male	0
4	28	Private	338409	Bachelors	13	Married-civ-spouse	Prof-specialty	Wife	Black	Female	0

```
cd.shape
```

```
(32561, 15)
```

- workclass,marital.status , occupation, relationship, race,sex, native.country : will create dummies
- education: will dropped because it has direct correspondence with education.num [check with pd.crosstab]
- Y : target variable , will be converted to 1/0

```
cd.drop('education',1,inplace=True)
```

```
cd['Y'].unique()
```

```
array([' <=50K', ' >50K'], dtype=object)
```

```
cd['Y']=(cd['Y']==' >50K').astype(int)
for col in ['workclass','marital.status' , 'occupation',
            'relationship', 'race','sex', 'native.country']:
    k=cd[col].value_counts()
    cats=k.index[k>300][:-1]
    for cat in cats :
        name=col+'_'+str(cat)
        cd[name]=(cd[col]==cat).astype(int)

    del cd[col]
```

```
cd.shape
```

```
(32561, 41)
```

```
x_train=cd.drop('Y',1)
y_train=cd['Y']

```

```
gbm_params={'n_estimators':[50,100,200,500],
           'learning_rate': [0.01,.05,0.1,0.4,0.8,1],
            'max_depth':[1,2,3,4,5,6],
            'subsample':[0.5,0.8,1],
            'max_features':[0.1,0.3,0.5,0.8,1]
           }
```

```
from sklearn.ensemble import GradientBoostingClassifier
```

```
model=GradientBoostingClassifier()
random_search=RandomizedSearchCV(model,scoring='roc_auc',
                                 param_distributions=gbm_params,
                                 cv=10,n_iter=10,
                                 n_jobs=-1,verbose=False)
```

```
random_search.fit(x_train,y_train)
```

```
report(random_search.cv_results_,5)
```

```
Model with rank: 1
Mean validation score: 0.92640 (std: 0.00292)
Parameters: {'subsample': 0.8, 'n_estimators': 500, 'max_features': 0.3, 'max_depth': 4,
'learning_rate': 0.1}

Model with rank: 2
Mean validation score: 0.91770 (std: 0.00358)
Parameters: {'subsample': 1, 'n_estimators': 200, 'max_features': 1, 'max_depth': 3, 'learning_rate':
0.4}

Model with rank: 3
Mean validation score: 0.91750 (std: 0.00400)
Parameters: {'subsample': 1, 'n_estimators': 500, 'max_features': 0.3, 'max_depth': 4,
'learning_rate': 0.01}

Model with rank: 4
Mean validation score: 0.91161 (std: 0.00430)
Parameters: {'subsample': 1, 'n_estimators': 100, 'max_features': 0.8, 'max_depth': 4,
'learning_rate': 1}

Model with rank: 5
Mean validation score: 0.90675 (std: 0.00456)
Parameters: {'subsample': 1, 'n_estimators': 500, 'max_features': 0.3, 'max_depth': 2,
'learning_rate': 0.01}
```

This gives us our best model parameters as well as its performance . We can further extract variable importance and make partial dependence plots here also as we did for randomForest results in the earlier module

Extreme Gradient Boosting Machines (Xgboost)

Although boosting machines are very powerful algorithms to extract very complex patterns and they have been fairly popular in their glory days. However they had few issues which were needed to be addressed with few new clever ideas. Xgboost does just that. The issues that it addresses are :

- GBM relies on individual models to be weak learners , but there is no framework enforcing this; ensuring that individual models are weak learners
- Entire focus instead is on brining down the cost, without any regularization on the complexity of the model which is a recipe for overfitting eventually
- The contribution from individual models (scores/update) are not aligned with the idea of optimizing the cost. they are simple averages instead .

Lets see how Xgboost addresses these concerns. The discussion will be deeply mathematical and pretty complex at places. Even if it doesnt make sense in one go, dont worry about it. Focus on implementation and usage steps, you can always give more passes to understand mathematical details later on

New cost formulation with regularization

We'll be using the word `obj` short for objective function, often used in optimization as an alternate name for loss/cost . Here is the new objective for t^{th} individual model that we are trying to add to our overall model F_{t-1} so far, as an update to arrive at F_t

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_t) + \sum_{i=1}^{t} \Omega(f_i) \qquad \cdots (2)$$

Where the first term represents the traditional loss that we have been using so far, for GBM, the second term represents the regularization/penalty on complexity for each of the individual models

As mentioned earlier , Xgboost , along with using regularization , uses some clever modification to loss formualtion that eventuall make it a great algorithm. One of them is what we are going to discuss next. It is using taylor's expansion of the traiditional loss

Taylor's expansion of loss

$$g(x+a) = g(x) + ag'(x) + \frac{a^2}{2}g''(x) + \frac{a^3}{6}g^{(3)}(x) + \cdots$$

This expression written above represents , general taylor's expansion of a function `g` where `a` is a very small quantity and g' is first order derivative , g'' is second order derivative and so on. Generally , higher order terms (terms with higher powers of `a` than a^2) are ignored , considering them be too close to zero , given `a` is a very small number . Lets Re-write (2) using this idea :

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1} + f_t) + \sum_{i=1}^{t} \Omega(f_i)$$

We have simply written F_t as $F_{t-1} + f_t$, here f_t is a small update , like `a` in above expression. $L(y_i, F_{t-1})$ is our `g` here in the expression

$$obj^{(t)} = \sum_{i=1}^{n} L(y_i, F_{t-1}) + f_t * \frac{\delta L(y_i, F_{t-1})}{\delta F_{t-1}} + \frac{{f_t}^2}{2} * \frac{\delta^2 L(y_i, F_{t-1})}{\delta^2 F_{t-1}} + \sum_{i=1}^{t} \Omega(f_i)$$

Notice that we have ignored higher order terms . Lets simplify this expression so that we don't have to write such complex mathematical expressions every time we write this objective function

- For an update f_t on an existing model F_{t-1}, the term $\sum_{i=1}^{n} L(y_i, F_{t-1})$ will be constant , we don't have to worry about optimizing that in the objective function. we can ignore that
- $g_i = \frac{\delta L(y_i, F_{t-1})}{\delta F_{t-1}}$
- $h_i = \frac{\delta^2 L(y_i, F_{t-1})}{\delta^2 F_{t-1}}$

Considering these ideas , our objective function can be written as follows :

$$obj^{(t)} = \sum_{i=1}^{n} [f_t * g_i + \frac{{f_t}^2}{2} * h_i] + \sum_{i=1}^{t} \Omega(f_i) \qquad \cdots (3)$$

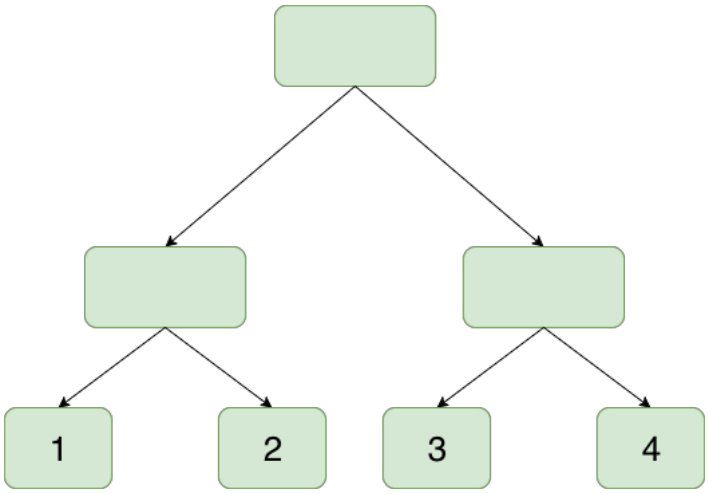
We'll see in sometime , how this helps . So far, we haven't concretely defined the penalty term.

Regularization Term

We have just written some general function Ω without really being explicit about it . Before blindly diving into how we define Ω, lets consider, what do we want to penalize here, what do we want to add the regularization term for :

- We want to ensure that individual models remain weak learners. In context of shallow decision tree, we need to ensure the tree size remains small. We can ensure that by adding penalty to the tree size (number of terminal nodes in the tree)
- We want the updates coming from individual models to be small , we can add L1/L2 penalty on the scores coming from individual trees

But what are these `scores` coming from individual trees. How do we represent a tree mathematically ? Lets have a look at what a tree is :



As we know from our discussion in the last module, every observation which ends up in the terminal node `1` will have some fixed score w_1 and so on. Lets say $q(x_i)$ represents the set of rules which tree is made of . The outcome of $q(x_i)$ for each observation x_i is one of these numbers : `{1,2,3,4}`. Essentially telling that in which terminal node the observation will end up . We can formally define our f_t or decision tree as follows :

$$f_t(x_i) = w_{q(x_i)}$$

Where $q(x_i) \in \{1, 2, 3 \cdots T\}$, given there are `T` terminal nodes in the tree

Now that we have defined the model f_t (Decision Tree in this case) , lets move on to formalize a regularization term for the same. considering the two concerns that we raised above , here is one regularization/penalty proposed by `Xgboost` team . You can always come up with some formulation of your own, this one works pretty well though.

First term penalizes size `T` of the trees thus ensuring that they are weak learners. Second term penalizes outcome/scores of trees , thus ensuring that update coming from individual model remains small . γ here controls the extent of penalty on the size of the penalty , and λ controls the extent of `L2` penalty on the scores .

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2$$

Consolidating updates to cost formulation

Adding the proposed regularization to the objective function (3) that we arrived at earlier, gets us to this :

$$obj^{(t)} = \sum_{i=1}^{n} [g_i w_{q(x_i)} + \frac{1}{2} h_i {w^2}_{q(x_i)}] + \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2 \qquad \cdots (4)$$

(4) still has some issues though. First term is written in terms of summation over all observation and second summation term is written as summation over terminal nodes of the tree. In order to consolidate things , we need to bring them under summation over same thing.

consider set I_j which contain indices `i` for observations which end up in j^{th} node . outcome of the model , for j^{th} node is already defined as w_j . Using this, we can write this all the summation terms in (4) over nodes of the tree .

$$obj^{(t)} = \sum_{j=1}^{T} [(\sum_{i \in I_j} g_i) w_j + \frac{1}{2} (\sum_{i \in I_j} h_i + \lambda) {w_j}^2] + \gamma T$$

we'll simplify this a little by considering $G_j = \sum_{i \in I_j} g_i$ and $H_j = \sum_{i \in I_j} h_i$

$$obj^{(t)} = \sum_{j=1}^{T} [G_j w_j + \frac{1}{2} (H_j + \lambda) {w_j}^2] + \gamma T \qquad \cdots (5)$$

Optimal value of model score/weights w_j

In traditional decision trees (same idea gets used in traditional `GBM` also) , score/outcome of the trees at nodes (w_j for j^{th} node) is simple average of the values in the node. However `Xgboost` team modified that idea as well. Instead of using simple average of outcomes in j^{th} node as output, they use optimal value as per the objective score in (5)

(5) is quadratic equation in w_j, it takes minimum value when :

$$w_j = -\frac{G_j}{H_j + \lambda}$$

putting this optimal value of w_j back in (5) we get :

$$obj^{(t)} = -\frac{1}{2} \sum_{j=1}^{T} \frac{{G_j}^2}{H_j + \lambda} + \gamma T \qquad \cdots (6)$$

Ok, we have put a lot of work in developing/understanding this `objective function`. Where do we use this ? . This is the objective function used to select rules when we are adding new partion to our tree (essentially building the tree)

Consider that we are splitting a node into its children. Some of the observations will go to right hand side and some will go to left hand side node . Change in objective function when we create this nodes going to be :

$$\text{Gain} = \frac{1}{2}\Big[\frac{{G_L}^2}{H_L+\lambda} + \frac{{G_R}^2}{H_R+\lambda} - \frac{(G_L+G_R)^2}{H_L+H_R+\lambda}\Big] - \gamma$$

This will be evaluated for each candidate rule, and the one with highest gain will be selected as the rule for that particular node. This is another big change to how trees were being built in tradition algorithms .

Before starting with building `Xgboost` models, lets take a quick look at parameters associated with the implementation in python and what should we keep in mind when we are tuning them .

Xgboost parameters

- n_estimators [default = 100]

Number of boosted trees in the model. If learning_rate/eta is small, you'll need higher number of boosted trees to capture proper patterns in the data

- eta [default=0.1, alias: learning_rate]

Step size shrinkage used in update to prevents overfitting. After each boosting step, we can directly get the weights of new features, and eta shrinks the feature weights to make the boosting process more conservative.
range: [0,1]

alias is nothing but an alternate name for the argument

- gamma [default=0, alias: min_split_loss]

Minimum loss reduction required to make a further partition on a leaf node of the tree. The larger gamma is, the more conservative(less prone to overfittng) the algorithm will be.
range: [0,∞]

- max_depth [default=3]

Maximum depth of a tree. Increasing this value will make the model more complex and more likely to overfit.

- min_child_weight [default=1]

Minimum sum of instance weight (hessian) needed in a child. If the tree partition step results in a leaf node with the sum of instance weight less than min_child_weight, then the building process will give up further partitioning. In linear regression task, this simply corresponds to minimum number of instances needed to be in each node. for classification it corresponds to minimum amount of impurity required to split a node. The larger min_child_weight is, the more conservative the algorithm will be.
range: [0,∞]

- max_delta_step [default=0]

Maximum delta step we allow each leaf output to be. If the value is set to 0, it means there is no constraint. If it is set to a positive value, it can help making the update step more conservative. Usually this parameter is not needed, **but it might help in logistic regression when class is extremely imbalanced.** Set it to value of 1-10 might help control the update.
range: [0,∞]

- subsample [default=1]

Subsample ratio of the training instances. Setting it to 0.5 means that XGBoost would randomly sample half of the training data prior to growing trees. and this will prevent overfitting. Subsampling will occur once in every boosting iteration.
range: (0,1]

- colsample_bytree, colsample_bylevel, colsample_bynode [default=1]

This is a family of parameters for subsampling of columns. - All colsample_by* parameters have a range of (0, 1], the default value of 1, and specify the fraction of columns to be subsampled. `colsample_bytree` is the subsample ratio of columns when constructing each tree. Subsampling occurs once for every tree constructed. `colsample_bylevel` is the subsample ratio of columns for each level. Subsampling occurs once for every new depth level reached in a tree. Columns are subsampled from the set of columns chosen for the current tree. `colsample_bynode` is the subsample ratio of columns for each node (split). Subsampling occurs once every time a new split is evaluated. Columns are subsampled from the set of columns chosen for the current level.colsample_by* parameters work cumulatively. For instance, the combination {'colsample_bytree':0.5, 'colsample_bylevel':0.5, 'colsample_bynode':0.5} with 64 features will leave 4 features to choose from at each split.

- lambda [default=1, alias: reg_lambda]

L2 regularization term on weights. Increasing this value will make model more conservative.

- alpha [default=0, alias: reg_alpha]

L1 regularization term on weights. Increasing this value will make model more conservative.
tree_method string [default= auto]

- scale_pos_weight [default=1]

Control the balance of positive and negative weights, **useful for unbalanced classes.** A typical value to consider: $\frac{\text{Frequency of negative instances}}{\text{Frequency of positive instances}}$. Not Relevant for regression problems

Regression with Xgboost (Sequential Tuning)

If we tune all the parameters together , there are chances that our results will be much far from the best. There are many parameters where variation doesn't impact the performance too much and we can tune them later once we have fixed values of parameters with volatile performance.

As a general strategy you can start with tuning number of trees or n_estimators , in case of boosting machines , learning_rate is directly related with n_estimators . A very low learning_rate will need high number of n_estimators . We can start with a decent fixed learning rate and tune n_estimaors for it.

All can be left as default for now except subsample , colsample_bytree and colsample_bylevel, these are set to default=1, we'll take a more conservative value 0.8

```
from sklearn.model_selection import GridSearchCV
from xgboost.sklearn import XGBRegressor
xgb_params = { "n_estimators":[25,50,100,150,200] }

xgb1=XGBRegressor(subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)
grid_search=GridSearchCV(xgb1,cv=10,param_grid=xgb_params,scoring='neg_mean_absolute_error',verbose=False,n_jobs=-1)
x_train=bike.drop('cnt',1)
y_train=bike['cnt']
grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1633.71079 (std: 423.52205)
Parameters: {'n_estimators': 25}

Model with rank: 2
Mean validation score: -1711.53315 (std: 185.46924)
Parameters: {'n_estimators': 50}

Model with rank: 3
Mean validation score: -1758.83604 (std: 187.03617)
Parameters: {'n_estimators': 100}
```

we'll use `n_estimators` as 25 here onwards. Lets tune other remaining parameters sequentially. Next we'll tune max_depth,gamma and min_child_weight, which control overfit by controlling size of individual trees

```
xgb_params={
            "gamma":[0,2,5,8,10],
            "max_depth": [2,3,4,5,6,7,8],
            "min_child_weight":[0.5,1,2,5,10]
            }
xgb2=XGBRegressor(n_estimators=25,subsample=0.8,
                  colsample_bylevel=0.8,colsample_bytree=0.8)
random_search = RandomizedSearchCV( xgb2, param_distributions = xgb_params,
                                    n_iter = 20, cv= 10,
                                    scoring ='neg_mean_absolute_error',
                                    n_jobs =-1, verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1561.90731 (std: 420.92672)
Parameters: {'min_child_weight': 0.5, 'max_depth': 2, 'gamma': 2}

Model with rank: 1
Mean validation score: -1561.90731 (std: 420.92672)
Parameters: {'min_child_weight': 0.5, 'max_depth': 2, 'gamma': 8}

Model with rank: 3
Mean validation score: -1633.71079 (std: 423.52205)
Parameters: {'min_child_weight': 1, 'max_depth': 3, 'gamma': 5}

Model with rank: 3
Mean validation score: -1633.71079 (std: 423.52205)
Parameters: {'min_child_weight': 1, 'max_depth': 3, 'gamma': 8}
```

Next we'll tune subsampling arguments to take care of potential effect of noisy observations [if this was classification problem , we'd have tuned `max_delta_step` and `scale_pos_weight` to take care of impact of class imbalance in the data before doing this]. Notice that now we'll use the values of parameters tuned so far from the best model outcomes from the gridsearch .\

```
xgb_params={
            'subsample':[i/10 for i in range(5,11)],
            'colsample_bytree':[i/10 for i in range(5,11)],
            'colsample_bylevel':[i/10 for i in range(5,11)]
            }
xgb3=XGBRegressor(learning_rate=0.1,n_estimators=25,
                  min_child_weight=0.5,gamma=2,max_depth=2)
random_search=RandomizedSearchCV(xgb3,param_distributions=xgb_params,cv=10,
                                 n_iter=20,scoring='neg_mean_absolute_error',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1539.31339 (std: 416.70732)
Parameters: {'subsample': 0.6, 'colsample_bytree': 0.6, 'colsample_bylevel': 0.6}

Model with rank: 2
Mean validation score: -1547.86783 (std: 420.01529)
Parameters: {'subsample': 1.0, 'colsample_bytree': 0.7, 'colsample_bylevel': 0.8}

Model with rank: 3
Mean validation score: -1553.33132 (std: 411.66023)
Parameters: {'subsample': 0.8, 'colsample_bytree': 0.8, 'colsample_bylevel': 0.6}
```

```
xgb_params={
            'reg_lambda':[i/10 for i in range(0,50)],
            'reg_alpha':[i/10 for i in range(0,50)]
            }
xgb4=XGBRegressor(n_estimators=25,min_child_weight=0.5,
                  gamma=2,max_depth=2, colsample_bylevel= 0.6,
                  colsample_bytree= 0.6, subsample= 0.6)
random_search=RandomizedSearchCV(xgb4,param_distributions=xgb_params,cv=10,
                                 n_iter=20,scoring='neg_mean_absolute_error',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: -1512.15494 (std: 450.56904)
Parameters: {'reg_lambda': 3.5, 'reg_alpha': 2.3}

Model with rank: 2
Mean validation score: -1514.04499 (std: 434.73313)
Parameters: {'reg_lambda': 2.2, 'reg_alpha': 4.1}

Model with rank: 3
Mean validation score: -1514.41400 (std: 435.72180)
Parameters: {'reg_lambda': 2.3, 'reg_alpha': 3.0}
```

Considering the best parameter values thus obtained , our final model is :

```
xgb5=XGBRegressor(n_estimators=25,min_child_weight=0.5,
                  gamma=2,max_depth=2, colsample_bylevel= 0.6,
                  colsample_bytree= 0.6, subsample= 0.6,
                 reg_lambda=3.5,reg_alpha=2.3)

```

Lets check its performance using cross validation

```
from sklearn.model_selection import cross_val_score
scores=-
cross_val_score(xgb5,x_train,y_train,scoring='neg_mean_absolute_error',verbose=False,n_jobs=-1,cv=10)

```

```
scores
```

```
array([1382.17801316, 1141.28446396, 1065.85468616, 1041.07993432,
        888.63474221, 1595.63349977, 1977.38205667, 1851.14672852,
       2266.21834198, 1913.92170945])
```

```
np.mean(scores)
```

```
1512.3334176203005
```

```
np.std(scores)
```

```
450.85032233797364
```

Couple comments for this specific problem here

- we should not build a complex model such as `Xgboost` for this small data
- Although the average performance is better than other alternatives , but the varition across data is huge
- performance is not very reliable, you should not be looking at just the average performance

Classification with Xgboost

We'll use the same sequential tuning of the parameters for the classification problem that we earlier solved with GBM

```
x_train=cd.drop('Y',1)
y_train=cd['Y']

from xgboost.sklearn import XGBClassifier
```

```
xgb_params = {
               "n_estimators":[100,500,700,900,1000,1200,1500]
            }
xgb1=XGBClassifier(subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)
grid_search=GridSearchCV(xgb1,cv=5,param_grid=xgb_params,
                         scoring='roc_auc',verbose=False,n_jobs=-1)
grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92749 (std: 0.00197)
Parameters: {'n_estimators': 500}

Model with rank: 2
Mean validation score: 0.92722 (std: 0.00170)
Parameters: {'n_estimators': 700}

Model with rank: 3
Mean validation score: 0.92683 (std: 0.00152)
Parameters: {'n_estimators': 900}
```

```
xgb_params={
            "gamma":[0,2,5,8,10],
            "max_depth": [2,3,4,5,6,7,8],
            "min_child_weight":[0.5,1,2,5,10]
            }
xgb2=XGBClassifier(n_estimators=500,subsample=0.8,
                   colsample_bylevel=0.8,colsample_bytree=0.8)
random_search=RandomizedSearchCV(xgb2,param_distributions=xgb_params,n_iter=20,
                                 cv=5,scoring='roc_auc',
                                 n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92711 (std: 0.00186)
Parameters: {'min_child_weight': 2, 'max_depth': 6, 'gamma': 5}

Model with rank: 2
Mean validation score: 0.92676 (std: 0.00215)
Parameters: {'min_child_weight': 1, 'max_depth': 8, 'gamma': 5}

Model with rank: 3
Mean validation score: 0.92670 (std: 0.00238)
Parameters: {'min_child_weight': 1, 'max_depth': 6, 'gamma': 10}
```

```
xgb_params={
            'max_delta_step':[0,1,3,6,10],
            'scale_pos_weight':[1,2,3,4]
            }
xgb3=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                   subsample=0.8,colsample_bylevel=0.8,colsample_bytree=0.8)

grid_search=GridSearchCV(xgb3,param_grid=xgb_params,
                         cv=5,scoring='roc_auc',n_jobs=-1,verbose=False)

grid_search.fit(x_train,y_train)
report(grid_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92796 (std: 0.00219)
Parameters: {'max_delta_step': 0, 'scale_pos_weight': 2}

Model with rank: 1
Mean validation score: 0.92796 (std: 0.00219)
Parameters: {'max_delta_step': 1, 'scale_pos_weight': 2}

Model with rank: 1
Mean validation score: 0.92796 (std: 0.00219)
Parameters: {'max_delta_step': 3, 'scale_pos_weight': 2}

Model with rank: 1
Mean validation score: 0.92796 (std: 0.00219)
Parameters: {'max_delta_step': 6, 'scale_pos_weight': 2}

Model with rank: 1
Mean validation score: 0.92796 (std: 0.00219)
Parameters: {'max_delta_step': 10, 'scale_pos_weight': 2}
```

```
xgb_params={
            'subsample':[i/10 for i in range(5,11)],
            'colsample_bytree':[i/10 for i in range(5,11)],
            'colsample_bylevel':[i/10 for i in range(5,11)]
            }
xgb4=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                       scale_pos_weight=2,max_delta_step=0
                   )
random_search=RandomizedSearchCV(xgb4,param_distributions=xgb_params,
                                 cv=5,n_iter=20,scoring='roc_auc',
                                n_jobs=-1,verbose=False)
random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92858 (std: 0.00187)
Parameters: {'subsample': 0.9, 'colsample_bytree': 0.5, 'colsample_bylevel': 1.0}

Model with rank: 2
Mean validation score: 0.92853 (std: 0.00218)
Parameters: {'subsample': 0.8, 'colsample_bytree': 0.5, 'colsample_bylevel': 0.9}

Model with rank: 3
Mean validation score: 0.92838 (std: 0.00202)
Parameters: {'subsample': 0.9, 'colsample_bytree': 0.9, 'colsample_bylevel': 1.0}
```

```
xgb_params={
            'reg_lambda':[i/10 for i in range(0,50)],
            'reg_alpha':[i/10 for i in range(0,50)]
            }
xgb5=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                       scale_pos_weight=2,max_delta_step=0,
                   colsample_bylevel=1.0, colsample_bytree= 0.5, subsample= 0.9)
random_search=RandomizedSearchCV(xgb5,param_distributions=xgb_params,
                                 cv=5,n_iter=20,scoring='roc_auc',
                                n_jobs=-1,verbose=False)

random_search.fit(x_train,y_train)
report(random_search.cv_results_,3)
```

```
Model with rank: 1
Mean validation score: 0.92874 (std: 0.00192)
Parameters: {'reg_lambda': 0.7, 'reg_alpha': 0.3}

Model with rank: 2
Mean validation score: 0.92860 (std: 0.00195)
Parameters: {'reg_lambda': 1.8, 'reg_alpha': 0.0}

Model with rank: 3
Mean validation score: 0.92853 (std: 0.00192)
Parameters: {'reg_lambda': 0.9, 'reg_alpha': 0.8}
```

```
xgb6=XGBClassifier(n_estimators=500,min_child_weight=2,gamma=5,max_depth=6,
                   scale_pos_weight=2,max_delta_step=0,
                   colsample_bylevel=1.0, colsample_bytree= 0.5, subsample= 0.9,
                  reg_lambda=0.7,reg_alpha=0.3)
scores=cross_val_score(xgb6,x_train,y_train,scoring='roc_auc',
                       verbose=False,n_jobs=-1,cv=10)

```

```
np.mean(scores)
```

```
0.9292637432983983
```

```
np.std(scores)
```

```
0.0030315969143107297
```

We'll conclude our discussion here on boosting machines . For making predictions on new data, same functions on the model object `predict` and `predict_proba` can be used . As usual you'll need to ensure that test data on which you want to make predictions has same columns and type as in the training data which was passed to the algorithm while training.

Note : different runs might lead to slightly different parameter values , however the performance wouldn't be wildly different