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 :

$$F_t(X_i) = f_1(X_i) + f_2(X_i) ..... + f_t(X_i)$$

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 severely 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.

Remember 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 :

$$W \rightarrow W - \eta * \nabla \mathcal{L}$$

or

$$\Delta W = -\eta * \nabla \mathcal{L}$$

- Parameters Ws 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 $-\eta * \nabla \mathcal{L}$, 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 $-\eta * \nabla \mathcal{L}$ 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 $-\eta * \nabla \mathcal{L}$ as the target for the model which it is trying to match by making predictions as close to it as possible.

Formally if we write our cost/loss =$\sum \mathcal{L}(y_i, F_t)$

where

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

then

$$f_{t+1} \rightarrow -\eta * \frac{\partial \mathcal{L}}{\partial F_t} \qquad \cdots (1)$$

GBM for Regression

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

$$\mathcal{L} = (y_i - F_t)^2$$

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

$$\frac{\partial \mathcal{L}}{\partial F_t} = -2 * (y_i - F_t)$$

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 classification, you'll find this formulation for loss

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

lets take its derivative w.r.t. F_t

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

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

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 :

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

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`.
```


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
- We brought in gradient descent with wishful assumption that each individual tree's contribution [prediction] to the updation of the model will be small. i.e. predictions from individual models will be small. But there is no mathematical guarantee in the formulation for the same.

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 (predictions at node) 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 doesn't make sense in one go, don't worry about it. You can always give more readings to this to understand mathematical details better.

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

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

Where the first term $\mathcal{L}$ represents the traditional loss that we have been using so far, for GBM, the second term Ω 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 formulation that eventually make it a great algorithm. One of them is what we are going to discuss next. It is using taylor's expansion of the traditional 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 :

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

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

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

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 $\sum_{i=1}^{n} \mathcal{L}(y_i, F_{t-1})$ will be constant , we don't have to worry about optimizing that in the objective function. we can ignore that
- $g_i = \frac{\partial \mathcal{L}(y_i, F_{t-1})}{\partial F_{t-1}}$
- $h_i = \frac{\partial^2 \mathcal{L}(y_i, F_{t-1})}{\partial^2 F_{t-1}}$

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

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

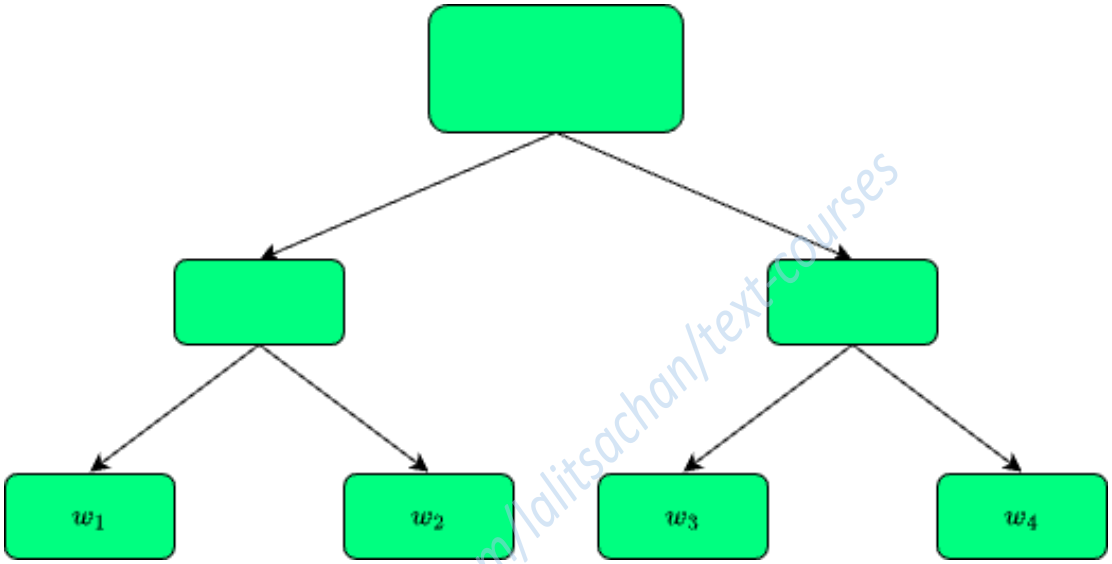
We'll see in sometime , how this helps . So far, we haven't concretely defined the penalty term.

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 .

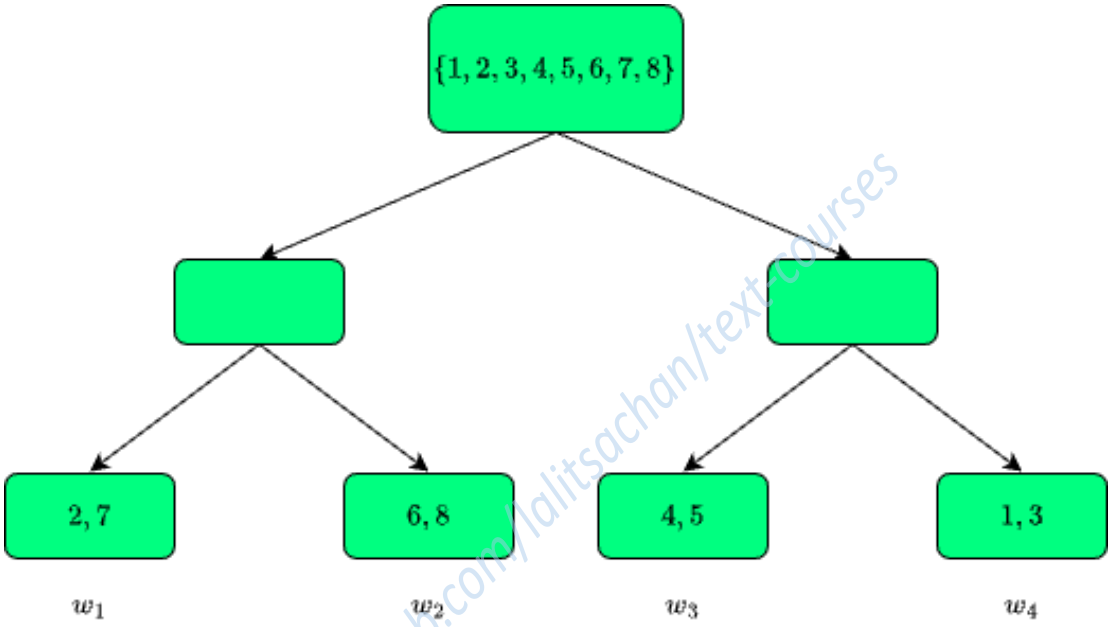
$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} {w_j}^2$$

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. Lets understand that with an example , consider the tree shown below :



in the root node we have 8 observation with their indices as shown in the figure , and these indices after passing through different rules end up in 4 leaf nodes as shown. Lets unwrap the expression $\sum_{i=1}^{n}[g_i w_{q(x_i)} + \frac{1}{2}h_i {w^2}_{q(x_i)}]$ for this example :

$$\begin{aligned}\sum_{i=1}^{n}[g_i w_{q(x_i)} + \frac{1}{2}h_i {w^2}_{q(x_i)}] &= [g_1 w_4 + g_2 w_1 + g_3 w_4 + g_5 w_3 + g_6 w_2 + g_7 w_1 + g8 w_2] \\ &+ \frac{1}{2}[h_1 w_4^2 + h_2 w_1^2 + h_3 w_4^2 + h_5 w_3^2 + h_6 w_2^2 + h_7 w_1^2 + h8 w_2^2] \\ &= [(g_2 + g_7)w_1 + (g_6 + g_8)w_2 + (g_4 + g_5)w_3 + (g_1 + g_3)w_4] \\ &+ \frac{1}{2}[(h_2 + h_7)w_1^2 + (h_6 + h_8)w_2^2 + (h_4 + h_5)w_3^2 + (h_1 + h_3)w_4^2]\end{aligned}$$

we just rearranged some terms here . Now 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 all the summation terms in (4) over nodes of the tree .

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

we'll simplify this a little by considering $G_j = \sum_{i \in I_j} g_i$ [sum of gradients of the observations in the j^{th} node] and $H_j = \sum_{i \in I_j} h_i$ [sum of hessians of the observations in the j^{th} node]

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

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 overfitting) 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]

max w_j value we allow for a leaf node to output. 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 helps 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.

CatBoost: Categorical Boosting Machines

So far, we've explored **Gradient Boosting Machines (GBM)** and **XGBoost**, two of the most widely used boosting algorithms. While both are powerful and effective, they have limitations, particularly when dealing with **categorical features** and **bias in residual estimation** (known as **prediction shift**).

CatBoost (**Cat**egorical **Boost**ing), developed by Yandex, introduces solutions to these challenges through:

1. **Native handling of categorical features**, reducing the need for preprocessing like one-hot encoding.
2. **Ordered Boosting**, a novel method that eliminates prediction shift and reduces overfitting.
3. **Symmetric (Oblivious) Trees**, improving training efficiency and model interpretability.

Let's explore the key ideas behind CatBoost in detail.

1. Handling High-Cardinality Categorical Features

In real-world datasets, many features are categorical—such as user IDs, product categories, or ZIP codes. Traditional boosting methods like GBM and XGBoost require **manual encoding** (one-hot encoding or label encoding). This approach works but has serious downsides:

- **Feature space explosion**: One-hot encoding creates **many** new binary columns, especially when categories have high cardinality.
- **Memory inefficiency**: Sparse representations slow down training and increase memory usage.
- **Overfitting risks**: One-hot encoding treats rare categories the same as common ones, often leading to poor generalization.

Instead of expanding categorical features into multiple columns, **CatBoost natively incorporates them into the model** using **target statistics**—computed in a way that prevents information leakage.

Ordered Target Statistics: Preventing Information Leakage

A naive way to encode categories is by replacing them with **the mean target value** (e.g., for a categorical feature "City", use the average price of houses in that city). However, this introduces **data leakage**: the model indirectly learns from the target variable, making it prone to overfitting.

CatBoost prevents this by using **Ordered Target Statistics**, a technique where the mean target for a category is computed **only from past observations**, ensuring that an instance does not "see" its own target during training.

Mathematical Formulation

Given a categorical feature x_i and its corresponding target y_i, we want to compute a target-based encoding $\hat{y}_{\pi(i)}$. Instead of using all data, CatBoost **randomly permutes** the dataset and computes target statistics only from earlier samples:

$$\hat{y}_{\pi(i)} = \frac{\sum_{j=1}^{i-1} \mathbb{I}[x_{\pi(j)} = x_{\pi(i)}] \cdot y_{\pi(j)} + a}{\sum_{j=1}^{i-1} \mathbb{I}[x_{\pi(j)} = x_{\pi(i)}] + b}$$

Where:

- $\pi(i)$ is the **random permutation order** for training examples.
- $\mathbb{I}[\cdot]$ is an indicator function ensuring only past samples contribute.
- a is a **prior term** (e.g., global mean) to prevent noise from rare categories.
- b is a **smoothing factor** to control regularization.

This ensures that **no category sees its own label** when computing statistics, preventing leakage and improving generalization.

Lets understand this with a small example

Example: Ordered Target Statistics in Action

CatBoost internally replaces categorical values with **numerical encodings** computed from **target statistics**. This example shows exactly how that happens and how CatBoost prevents data leakage in the process.

Let's consider a dataset with a categorical feature `"City"` and a target:

Use case 1: Regression (`House Price`)

Let's say we have the following dataset (before permutation):

Index	City	House Price
1	A	300
2	B	500
3	A	350
4	C	400
5	B	450
6	A	320

Step 1: Apply a Random Permutation

CatBoost first shuffles the data into a new order before computing encodings. This permutation defines which rows are considered "previous" for any given row:

New Index	City	House Price
1	B	500
2	A	300
3	B	450
4	C	400
5	A	350
6	A	320

Step 2: Compute Ordered Target Encoding

CatBoost computes the **encoded value for each categorical observation** using:

$$\hat{y}_{\pi(i)} = \frac{\sum_{j=1}^{i-1} \mathbb{I}[x_{\pi(j)} = x_{\pi(i)}] \cdot y_{\pi(j)} + a}{\sum_{j=1}^{i-1} \mathbb{I}[x_{\pi(j)} = x_{\pi(i)}] + b}$$

Let's use:

- $a = 400$ (global mean target value)
- $b = 1$ (smoothing factor)

Now we compute encoding row by row:

New Index	City	House Price	Ordered Target Encoding ($\hat{y}$)	Calculation
1	B	500	400	prior only
2	A	300	400	prior only
3	B	450	$\frac{500+400}{1+1} = 450$	1 prev
4	C	400	400	prior only
5	A	350	$\frac{300+400}{1+1} = 350$	1 prev
6	A	320	$\frac{300+350+400}{2+1} = 350$	2 prev

So after encoding, the categorical feature `"City"` is **replaced** with the numeric column `"Ordered Target Encoding"` — which is dynamic, history-dependent, and leakage-free.

Same idea works for classification tasks as well

Let's consider a dataset where the target is binary (1 = Purchased, 0 = Not Purchased):

Use case 2: Classification (`Purchased`)

Index	City	Purchased (1/0)
1	X	1
2	Y	0
3	X	1
4	Z	1
5	Y	0
6	X	0
7	Y	1
8	Z	1
9	X	0
10	Y	1

Step 1: Random Permutation

Let's assume CatBoost shuffles the rows like this:

New Index	City	Purchased
5	Y	0
1	X	1
3	X	1
6	X	0
2	Y	0
7	Y	1
4	Z	1
8	Z	1
9	X	0
10	Y	1

Step 2: Compute Ordered Target Encoding

Let's again set:

- $a = 0.5$ (prior probability of class 1)
- $b = 1$ (smoothing factor)

New Index	City	Purchased	Ordered Encoding	Calculation
1	Y	0	0.5	prior only
2	X	1	0.5	prior only
3	X	1	$\frac{1+0.5}{1+1} = 0.75$	1 prev
4	X	0	$\frac{1+1+0.5}{2+1} = 0.833$	2 prev
5	Y	0	$\frac{0+0.5}{1+1} = 0.25$	1 prev
6	Y	1	$\frac{0+0+0.5}{2+1} = 0.167$	2 prev
7	Z	1	0.5	prior only
8	Z	1	$\frac{1+0.5}{1+1} = 0.75$	1 prev
9	X	0	$\frac{1+1+0+0.5}{3+1} = 0.625$	3 prev
10	Y	1	$\frac{0+0+1+0.5}{3+1} = 0.375$	3 prev

Key Observations

- The **first time** a category appears, we use just the prior a.

- Later encodings are **moving averages** between historical outcomes and the prior.
- As more data accumulates, the encoding becomes **more data-driven** and **less influenced by the prior**.
- This **replaces** the original categorical feature with a **numeric column** suitable for gradient-based optimization.
- Most importantly, **the encoding for each row is computed without ever using its own label**, ensuring zero leakage.

2. Prediction Shift & Ordered Boosting

A common issue in boosting algorithms is **prediction shift**, which occurs when models compute residuals using predictions from the same dataset they were trained on.

Why Does Prediction Shift Happen?

In standard GBM and XGBoost:

1. A weak model f_1 is trained on the dataset to approximate the target y.
2. The next weak model f_2 is trained on the residuals $(y - f_1(x))$.
3. The process continues, with each weak learner training on the errors left by the previous models.

However, these residuals are computed using models trained **on the same data**, meaning the errors themselves are biased. Over time, this bias compounds, leading to:

- **Overfitting**: The model fits training residuals too well, reducing generalization.
- **Unstable updates**: Errors estimated on the same data may not reflect true generalization error.

Ordered Boosting: How CatBoost Fixes This

CatBoost introduces **Ordered Boosting**, which ensures that residuals for each example are computed **only using models trained on previous data points**.

Instead of directly using residuals from the full dataset, CatBoost:

- Randomly shuffles the dataset into **multiple permutations**.
- For each example x_i, the residual is computed only from **models trained on earlier samples** in the permutation.
- This prevents the model from "seeing" its own training errors, eliminating bias.

Mathematically, for an example x_i appearing at position i in permutation π, its residual is computed using:

$$F_t^{(i)} = \sum_{s=1}^{t} f_s^{(<i)}(x_i)$$

Where $f_s^{(<i)}$ is the s^{th} weak learner trained **only on samples appearing before** x_i in the permutation.

This ensures that the residuals remain unbiased, preventing overfitting and improving robustness.

3. Oblivious (Symmetric) Trees

Unlike GBM and XGBoost, which use **asymmetric trees** where each node is split independently, CatBoost uses **symmetric (oblivious) trees**, where the same split is applied at every depth.

Mathematical Structure

A symmetric tree of depth d can be written as:

$$q(x) \in \{1, 2, \ldots, 2^d\}$$

Where every sample x follows a **fixed decision path** determined by **the same sequence of split conditions** at each level.

Advantages of Symmetric Trees

- **Faster training**: The structured nature of trees allows for **vectorized computation**.
- **Regularization**: The model remains balanced, reducing overfitting.
- **Better interpretability**: Decisions at each level are consistent across all branches.

CatBoost Loss Function & Optimization

Like XGBoost, CatBoost optimizes a **regularized objective function**:

$$\mathcal{L} = \sum_{i=1}^{n} \mathcal{L}(y_i, F_t(x_i)) + \Omega(f_t)$$

Where the regularization term is:

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

CatBoost **further follows the same Taylor approximation** strategy introduced in XGBoost. The loss is expanded using the second-order Taylor expansion:

$$\mathcal{L}(y_i, F_t + f_t) \approx \mathcal{L}(y_i, F_t) + g_i f_t + \frac{1}{2} h_i f_t^2$$

Where:

- $g_i = \frac{\partial \mathcal{L}(y_i, F_t)}{\partial F_t}$ — Gradient
- $h_i = \frac{\partial^2 \mathcal{L}(y_i, F_t)}{\partial^2 F_t}$ — Hessian

The **leaf values** w_j for the tree are then computed to minimize this approximation. Just like XGBoost, this leads to:

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

Where:

- $G_j = \sum_{i \in I_j} g_i$, and
- $H_j = \sum_{i \in I_j} h_i$

So while the **optimization logic** mirrors that of XGBoost, **CatBoost's distinction lies in how the gradients are computed** — using **ordered permutations** to prevent bias and overfitting. This is what makes CatBoost robust even without aggressive regularization tuning.

CatBoost Parameters

CatBoost is designed to work well even with minimal tuning, but it exposes a rich set of hyperparameters for flexibility and control.

Core Training Parameters

Parameter	Description
`iterations`	Total number of boosting rounds. Equivalent to `n_estimators`.
`learning_rate`	Controls the contribution of each tree. Lower values require more iterations.
`depth`	Depth of the symmetric trees. Each tree will have 2^{depth} leaves.
`loss_function`	Defines the loss to be optimized. Examples: `"Logloss"`, `"RMSE"`, `"MAE"`, `"MultiClass"`.
`eval_metric`	Metric used for evaluation. Often same as loss, but can be different.
`random_strength`	Noise added to score when selecting best splits. Helps reduce overfitting.
`l2_leaf_reg`	L2 regularization term on leaf scores (like λ in XGBoost). Higher = more conservative.
`bootstrap_type`	Sampling method: `"Bayesian"`, `"Bernoulli"`, `"MVS"`, `"No"` (no sampling).
`subsample`	Ratio of samples used per tree if `bootstrap_type` supports it.
`min_data_in_leaf`	Minimum number of samples required in each leaf. Controls overfitting.

Categorical Feature Handling

Parameter	Description
`cat_features`	List of column indices or names that are categorical.
`one_hot_max_size`	Threshold to decide which categorical features to one-hot encode directly.
`permutation_count`	Number of random permutations for ordered boosting and target statistics.
`target_stats_smoothing`	Controls smoothing applied to target statistics (blends category mean with prior).

Overfitting Control

Parameter	Description
`early_stopping_rounds`	Stop training if validation metric doesn't improve.
`od_type`	Overfitting detection type: `"IncToDec"` or `"Iter"`.
`od_wait`	Number of rounds to wait before triggering early stopping.

CatBoost is a strong choice for datasets with categorical features and high risk of overfitting. Its built-in strategies help produce models that generalize well **without extensive tuning**, making it particularly valuable in real-world applications.

Refer to the notebook for coding hands-on.