梯度下降算法

GBDT, XGBoost, LightGBM

Yimeng Ren

1 梯度下降算法

根据网上资源讲解，首先这三种算法都属于 Boosting 方法，且 GBDT 是机器学习算法，XGBoost 和 LightGBM 是 GBDT 的算法实现。

Boosting 方法训练基分类器时采用串行的方式，各个基分类器之间有依赖。其基本思想是根据当前模型损失函数的负梯度信息来训练新加入的基分类器，然后将训练好的基分类器以累加的形式结合到现有模型中。这个过程是在不断地减小损失函数，使得模型偏差不断降低。但 Boosting 的过程并不会显著降低方差。这是因为 Boosting 的训练过程使得各基分类器之间是强相关的，缺乏独立性，所以并不会对降低方差有作用。

1.1 前向分步算法

AdaBoost 算法的另一个角度：模型为加法模型、损失函数为指数函数、学习算法为前向分步算法时的二类分类学习方法。

考虑加法模型：

$$f(x) = \sum_{m=1}^{M} \beta_m b\left(x; \gamma_m\right)$$

其中，$b\left(x; \gamma_m\right)$ 为基函数，γ_m 为基函数的参数，β_m 为基函数的系数。

在给定训练数据和损失函数 $L(y, f(x))$ 的条件下，学习加法模型 $f(x)$ 成为经验风险极小化，即损失函数极小化的问题：

$$\min_{\beta_m, \gamma_m} \sum_{i=1}^{N} L\left(y_i, \sum_{m=1}^{M} \beta_m b\left(x_i; \gamma_m\right)\right)$$

前向分步算法（forward stagewise algorithm）求解该问题的想法：学习的是加法模型，如果能从前向后，每一步只学习一个基函数及其系数，逐步逼近优化目标，就可以简化优化的复杂度。每一步只需要优化损失函数：

$$\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, \beta b\left(x_i; \gamma\right)\right)$$

前向分步算法：

输入：训练数据集 $T = \{(x_1, y_1), (x_2, y_2), \cdots, (x_N, y_N)\}$，损失函数 $L(y, f(x))$，基函数集 $\{b(x; \gamma)\}$

输出：加法模型 $f(x)$

（1）初始化 $f_0(x) = 0$；

（2）对 $m = 1, 2, ..., M$：

（a）最小化损失函数

$$(\beta_m, \gamma_m) = \arg\min_{\beta,\gamma} \sum_{i=1}^{N} L\left(y_i, f_{m-1}\left(x_i\right) + \beta b\left(x_i; \gamma\right)\right)$$

得到参数 β_m, γ_m

（b）更新

$$f_m(x) = f_{m-1}(x) + \beta_m b\left(x; \gamma_m\right)$$

（3）得到加法模型：

$$f(x) = f_M(x) = \sum_{m=1}^{M} \beta_m b\left(x; \gamma_m\right)$$

这样，前向分步算法将同时求解从 $m = 1\ to\ M$ 的所有参数 β_m, γ_m 的优化问题简化为逐次求解每个 β_m, γ_m 的优化问题。

1.2 GBDT

提升方法采用加法模型（基函数的线性组合）与前向分步算法，以决策树为基函数的提升方法称为提升树（boosting tree）。

$$f_M(x) = \sum_{m=1}^{M} T\left(x; \Theta_m\right)$$

其中，$T(x;\Theta_m)$ 表示决策树，Θ_m 为决策树的参数，M 为树的个数。

首先确定初始提升树 $f_0(x)=0$，第 m 步的模型是：

$$f_m(x)=f_{m-1}(x)+T(x;\Theta_m)$$

其中，$f_{m-1}(x)$ 为当前模型，通过经验风险极小化确定下一棵决策树的参数 Θ_m：

$$\hat{\Theta}_m=\arg\min_{\Theta_m}\sum_{i=1}^{N}L(y_i,f_{m-1}(x_i)+T(x_i;\Theta_m))$$

若这里的损失函数为平方损失，则称为 L2-Boosting 算法。

回归问题提升树使用以下前向分步算法：

$$\begin{aligned}&f_0(x)=0\\&f_m(x)=f_{m-1}(x)+T(x;\Theta_m),\quad m=1,2,\cdots,M\\&f_M(x)=\textstyle\sum_{m=1}^{M}T(x;\Theta_m)\end{aligned}$$

回归问题的提升树算法：

输入：训练数据集 $T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\},x_i\in\mathcal{X}\subseteq\mathbf{R}^n,y_i\in\mathcal{Y}\subseteq\mathbf{R}$

输出：提升树 $f_M(x)$

（1）初始化 $f_0(x)=0$;

（2）对 $m=1,2,...,M$：

（a）计算残差 $r_{mi}=y_i-f_{m-1}(x_i),\quad i=1,2,\cdots,N$

（b）拟合残差 r_{mi} 学习一个回归树，得到 $T(x;\Theta_m)$

（c）更新 $f_m(x)=f_{m-1}(x)+T(x;\Theta_m)$

（3）得到回归问题提升树：

$$f_M(x)=\sum_{m=1}^{M}T(x;\Theta_m)$$

当损失函数为平方损失和指数损失函数时，优化比较简单，但对于一般损失而言，使用**梯度提升算法**，这是利用最速下降法的近似方法，其关键是利用损失函数的负梯度在当前模型的值：

$$-\left[\frac{\partial L(y,f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

作为回归问题提升树算法中的残差的近似值，拟合一个回归树。

梯度提升算法:

输入：训练数据集 $T=\{(x_1,y_1),(x_2,y_2),\cdots,(x_N,y_N)\},x_i\in\mathcal{X}\subseteq\mathbf{R}^n,y_i\in\mathcal{Y}\subseteq\mathbf{R}$，损失函数 $L(y,f(x))$

输出：回归树 $\hat{f}(x)$

（1）初始化

$$f_0(x)=\arg\min_c\sum_{i=1}^{N}L(y_i,c)$$

估计使损失函数极小化的常数值，它是只有一个根节点的树。

（2）对 $m=1,2,...,M$:

（a）对 $i=1,2,...,N$，计算:

$$r_{mi}=-\left[\frac{\partial L(y_i,f(x_i))}{\partial f(x_i)}\right]_{f(x)=f_{m-1}(x)}$$

对于平方损失函数，这个就是通常所说的残差；对于一般损失函数，它是残差的近似值。

（b）对 r_{mi} 拟合一个回归树，得到第 m 棵树的叶节点区域 $R_{mj},j=1,2,...,J$

（c）对 $j=1,2,...,J$，计算:

$$c_{mj}=\arg\min_c\sum_{x_i\in R_{mj}}L(y_i,f_{m-1}(x_i)+c)$$

利用线性搜索估计叶节点区域的值，使损失函数极小化。

（d）更新 $f_m(x)=f_{m-1}(x)+\sum_{j=1}^{J}c_{mj}I\left(x\in R_{mj}\right)$

（3）得到回归树

$$\hat{f}(x)=f_M(x)=\sum_{m=1}^{M}\sum_{j=1}^{J}c_{mj}I\left(x\in R_{mj}\right)$$

一般意义上的**梯度下降提升算法**:

在每一轮迭代中，首先计算出当前模型在所有样本上的负梯度，然后以该值为目标训练一个新的基分类器进行拟合并计算出该基分类器的权重，最终实现对模型的更新。

$1:F_0(x)=\arg\min_\rho\sum_{i=1}^{N}L(y_i,\rho)$

2: For $m = 1$ to M do:

3: $\widetilde{y}_i = -\left[\frac{\partial L(y_i, F(x_i))}{\partial F(x_i)}\right]_{F(x)=F_{m-1}(x)}, i = 1, ..., M$

4: $\mathrm{a}_m = \arg\min_{a,\beta} \sum_{i=1}^{N} \left[\widetilde{y}_i - \beta h\left(x_i : a\right)\right]^2$

5: $\rho_m = \arg\min_{\rho} \sum_{i=1}^{N} L\left(y_i, F_{m-1}\left(x_i\right) + \rho h\left(x_i : a_m\right)\right)$

6: $F_m(x) = F_{m-1}(x) + \rho_m h\left(x : a_m\right)$

7: end For

8: end Algorithm

局限性:

1) GBDT 在高维稀疏的数据集上，表现不如支持向量机或者神经网络。

2) GBDT 在处理文本分类特征问题上，相对其他模型的优势不如它在处理数值特征时明显。

3) 训练过程需要串行训练，只能在决策树内部采用一些局部并行的手段提高训练速度。

1.3 XGBoost (Extreme Gradient Boosting)

XGBoost 是陈天奇等人开发的一个开源机器学习项目，高效地实现了 GBDT 算法并进行了算法和工程上的许多改进。

原始的 GBDT 算法基于经验损失函数的负梯度来构造新的决策树，只是在决策树构建完成后再进行剪枝。而 XGBoost 在决策树构建阶段就加入了正则项:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(t-1)} + f_t\left(\mathbf{x}_i\right)\right) + \Omega\left(f_t\right)$$

其中 $\Omega(f) = \gamma T + \frac{1}{2}\lambda\|w\|^2$，T 为叶节点数量，$w$ 为叶节点权重，保证叶子结点尽量少，节点数值 w 不极端。

XGBoost 算法的步骤和 GBDT 基本相同，都是首先初始化为一个常数，GBDT 是根据一阶导数，XGBoost 是根据一阶导数 g_i 和二阶导数 h_i，迭代生成基学习器，相加更新学习器。

对上述损失函数做二阶泰勒展开，其中 g 为一阶导数，h 为二阶导数，最后一项为罚:

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^{n}\left[l\left(y_i, \widehat{y}^{(t-1)}\right) + g_i f_t\left(\mathrm{x}_i\right) + \frac{1}{2} h_i f_t^2\left(\mathrm{x}_i\right)\right] + \Omega\left(f_t\right)$$
$$\text{where } g_i = \partial_{\widehat{y}(t-1)} l\left(y_i, \widehat{y}^{(t-1)}\right) \text{ and } h_i = \partial^2_{\widehat{y}^{(t-1)}} l\left(y_i, \widehat{y}^{(t-1)}\right)$$

$$
\begin{aligned}
Obj^{(t)} &\simeq \sum_{i=1}^{n}\left[g_i f_t\left(x_i\right)+\frac{1}{2} h_i f_t^2\left(x_i\right)\right]+\Omega\left(f_t\right) \\
&=\sum_{i=1}^{n}\left[g_i w_{q\left(x_i\right)}+\frac{1}{2} h_i w_{q\left(x_i\right)}^2\right]+\gamma T+\lambda \frac{1}{2} \sum_{j=1}^{T} w_j^2 \\
&=\sum_{j=1}^{T}\left[\left(\sum_{i \in I_j} g_i\right) w_j+\frac{1}{2}\left(\sum_{i \in I_j} h_i+\lambda\right) w_j^2\right]+\gamma T
\end{aligned}
$$

通过对 w_j 求导等于 0，可以得到：

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

将 w_j^* 代入得到：

$$
Obj=-\frac{1}{2} \sum_{j-1}^{T} \frac{G_j^2}{H_j+\lambda}+\gamma T
$$

这个分数越小，代表树的结构越好。在构建树的过程中，XGBoost 使用贪心 + 二次优化，从树深度 0 开始，每一个节点都遍历所有的特征，对每个特征进行分割，选取增益最好的那个特征，增益的计算使用了优化后 object 中的部分，并且引入了叶子节点的惩罚项。

防止过拟合：

- 加入因子 η 控制每一步提升的步长：$\widehat{y}_i^t=\widehat{y}_i^{(t-1)}+\eta f_t\left(x_i\right)$
- 进行列抽样

2 LightGBM

比 XGBOOST 更快–LightGBM 介绍

GBDT 在每一次迭代的时候，都需要遍历整个训练数据多次。如果把整个训练数据装进内存则会限制训练数据的大小；如果不装进内存，反复地读写训练数据又会消耗非常大的时间。

XGBoost 基于预排序的方法构建决策树：首先，对所有特征都按照特征的数值进行预排序。其次，在遍历分割点的时候用 O(#data) 的代价找到一个特征上的最好分割点。最后，在找到一个特征的最好分割点后，将数据分裂成左右子节点。

优点：能精确地找到分割点；

缺点：首先，空间消耗大。这样的算法需要保存数据的特征值，还保存了特征排序的结果（例如，为了后续快速的计算分割点，保存了排序后的索引），这就需要消耗训练数据两倍的内存。其次，时间上也有较大的开销，在遍历每一个分割点的时候，都需要进行分裂增益的计算，消耗的代价

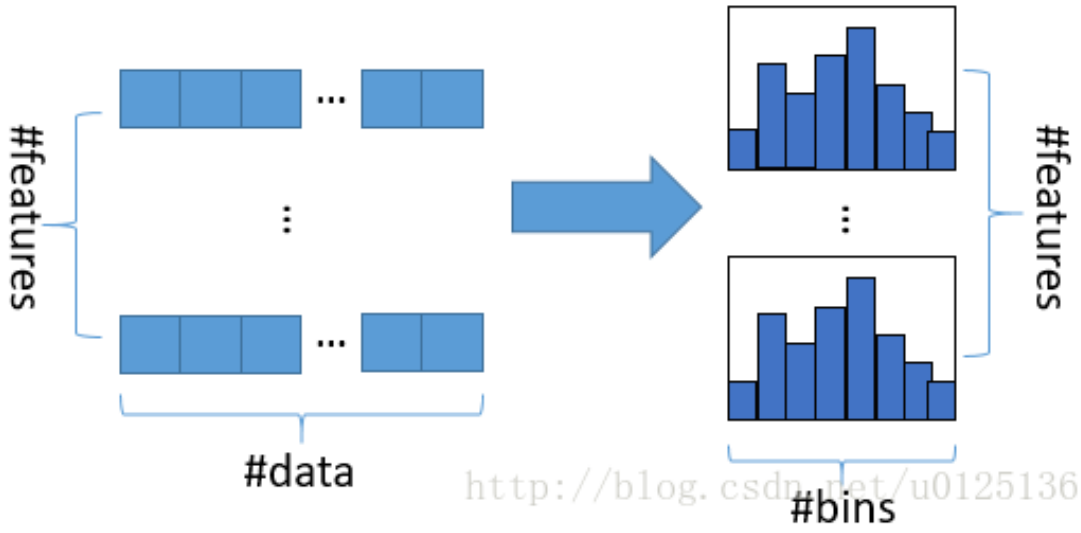
大。最后，对 cache 优化不友好。在预排序后，特征对梯度的访问是一种随机访问，并且不同的特征访问的顺序不一样，无法对 cache 进行优化。同时，在每一层长树的时候，需要随机访问一个行索引到叶子索引的数组，并且不同特征访问的顺序也不一样，也会造成较大的 cache miss。

为了避免上述 XGBoost 的缺陷，并且能够在不损害准确率的条件下加快 GBDT 模型的训练速度，lightGBM 在传统的 GBDT 算法上进行了如下优化：

2.1 直方图算法

直方图算法的基本思想是：先把连续的浮点特征值离散化成 k 个整数，同时构造一个宽度为 k 的直方图。在遍历数据的时候，根据离散化后的值作为索引在直方图中累积统计量，当遍历一次数据后，直方图累积了需要的统计量，然后根据直方图的离散值，遍历寻找最优的分割点。

首先确定对于每一个特征需要多少个箱子（bin）并为每一个箱子分配一个整数；然后将浮点数的范围均分成若干区间，区间个数与箱子个数相等，将属于该箱子的样本数据更新为箱子的值；最后用直方图（#bins）表示。看起来很高大上，其实就是直方图统计，将大规模的数据放在了直方图中。

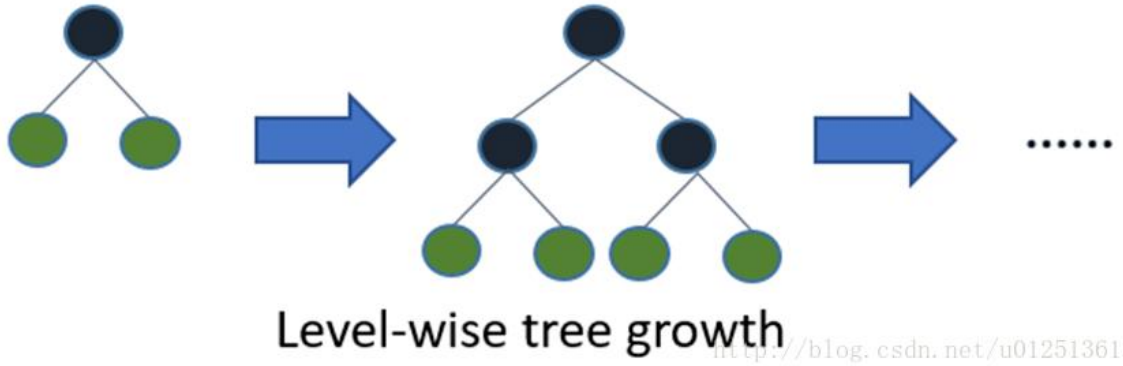


2.2 Leaf-wise 算法

在 Histogram 算法之上，LightGBM 进行进一步的优化。首先它抛弃了大多数 GBDT 工具使用的按层生长 (level-wise) 的决策树生长策略，而使用了带有深度限制的按叶子生长 (leaf-wise) 算法。

XGBoost 采用 Level-wise 的增长策略，该策略遍历一次数据可以同时分裂同一层的叶子，容易进行多线程优化，也好控制模型复杂度，不容易过拟合。但实际上 Level-wise 是一种低效的算法，

因为它不加区分的对待同一层的叶子，实际上很多叶子的分裂增益较低，没必要进行搜索和分裂，因此带来了很多没必要的计算开销。



LightGBM 采用 Leaf-wise 的增长策略，该策略每次从当前所有叶子中，找到分裂增益最大的一个叶子，然后分裂，如此循环。因此同 Level-wise 相比，Leaf-wise 的优点是：在分裂次数相同的情况下，Leaf-wise 可以降低更多的误差，得到更好的精度；Leaf-wise 的缺点是：可能会长出比较深的决策树，产生过拟合。因此 LightGBM 会在 Leaf-wise 之上增加了一个最大深度的限制，在保证高效率的同时防止过拟合。