XGBoost

XGBoost与GBDT

> XGBoost是对GBDT的一种高效实现，其中加入了许多独特的思路与方法，提升了模型的效率与鲁棒性，因此在各大赛事中频频霸榜，备受好评。

对比原算法GBDT，XGBoost主要从下面三个方面做了优化：

	GBDT	XGBoost
算法本身的优化	1. 只支持决策树作为基学习器	1. 支持许多其他的基分类器，例如线性分类器
	2. 平方损失	2. 加入了正则化损失
	3. 损失函数对误差部分只做一阶泰勒展开（负梯度方向）	3. 损失函数对误差部分做二阶泰勒展开，拟合更加准确
算法运行效率的优化		1. 对每个基学习器（如决策树）的建立过程进行并行选择，找到合适的子树分裂特征和特征值。
		2. 在并行选择之前，先对所有的特征的值进行排序分组，方便前面说的并行选择。
		3. 对分组的特征，选择合适的分组大小，使用CPU缓存进行读取加速。
		4. 将各个分组保存到多个硬盘以提高IO速度。
算法健壮性的优化		1. 对于缺失值的特征，通过枚举所有缺失值在当前节点是进入左子树还是右子树来决定缺失值的处理方式。
		2. 算法本身加入了L1和L2正则化项，可以防止过拟合，泛化能力更强。

XGBoost模型构建

假设数据为：$\mathcal{D}=\{(\mathbf{x}_i,y_i)\}\,(|\mathcal{D}|=n,\mathbf{x}_i\in\mathbb{R}^m,y_i\in\mathbb{R})$

(1) 构造目标函数：
假设有K棵树，则第i个样本的输出为$\hat{y}_i=\phi\left(\mathrm{x}_i\right)=\sum_{k=1}^{K}f_k\left(\mathrm{x}_i\right)$，　$f_k\in\mathcal{F}$，其中，
$\mathcal{F}=\left\{f(\mathbf{x})=w_{q(\mathbf{x})}\right\}\left(q:\mathbb{R}^m\rightarrow T,w\in\mathbb{R}^T\right)$

因此，目标函数的构建为：

$$\mathcal{L}(\phi)=\sum_i l\left(\hat{y}_i,y_i\right)+\sum_k\Omega\left(f_k\right)$$

其中，$\sum_i l\left(\hat{y}_i,y_i\right)$为loss function，$\sum_k\Omega\left(f_k\right)$为正则化项。

(2) 叠加式的训练(Additive Training)：

给定样本x_i，$\hat{y}_i^{(0)}=0$(初始预测)，$\hat{y}_i^{(1)}=\hat{y}_i^{(0)}+f_1(x_i)$，
$\hat{y}_i^{(2)}=\hat{y}_i^{(0)}+f_1(x_i)+f_2(x_i)=\hat{y}_i^{(1)}+f_2(x_i)$.....以此类推，可以得到：$\hat{y}_i^{(K)}=\hat{y}_i^{(K-1)}+f_K(x_i)$，其中，$\hat{y}_i^{(K-1)}$ 为前K-1棵树的预测结果，$f_K(x_i)$ 为第K棵树的预测结果。

因此，目标函数可以分解为：

$$\mathcal{L}^{(K)}=\sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(K-1)}+f_K\left(\mathrm{x}_i\right)\right)+\sum_k \Omega\left(f_k\right)$$

由于正则化项也可以分解为前K-1棵树的复杂度加第K棵树的复杂度，因此：
$\mathcal{L}^{(K)}=\sum_{i=1}^{n} l\left(y_i, \hat{y}_i^{(K-1)}+f_K\left(\mathrm{x}_i\right)\right)+\sum_{k=1}^{K-1} \Omega\left(f_k\right)+\Omega\left(f_K\right)$，由于$\sum_{k=1}^{K-1} \Omega\left(f_k\right)$在模型构建到第K棵树的时候已经固定，无法改变，因此是一个已知的常数，可以在最优化的时候省去，故：

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

(3) 使用泰勒级数近似目标函数：

$$\mathcal{L}^{(K)} \simeq \sum_{i=1}^{n}\left[l\left(y_i, \hat{y}^{(K-1)}\right)+g_i f_K\left(\mathrm{x}_i\right)+\frac{1}{2} h_i f_K^2\left(\mathrm{x}_i\right)\right]+\Omega\left(f_K\right)$$

其中，$g_i=\partial_{\hat{y}(t-1)} l\left(y_i, \hat{y}^{(t-1)}\right)$和$h_i=\partial_{\hat{y}^{(t-1)}}^2 l\left(y_i, \hat{y}^{(t-1)}\right)$

由于$\sum_{i=1}^{n} l\left(y_i, \hat{y}^{(K-1)}\right)$在模型构建到第K棵树的时候已经固定，无法改变，因此是一个已知的常数，可以在最优化的时候省去，故：

$$\tilde{\mathcal{L}}^{(K)}=\sum_{i=1}^{n}\left[g_i f_K\left(\mathbf{x}_i\right)+\frac{1}{2} h_i f_K^2\left(\mathbf{x}_i\right)\right]+\Omega\left(f_K\right)$$

构造一棵树

定义：

- 样本所在的节点位置$q(x)$
- 有哪些样本落在节点j上 $I_j=i \mid q(x_i)=j$
- 每个结点的预测值 $w_{q(x)}$
- 模型复杂度 $\Omega(f_K)$，它可以由叶子节点的个数以及节点函数值来构建，则：
  $\Omega\left(f_K\right)=\gamma J+\frac{1}{2} \lambda \sum_{j=1}^{J} w_j^2$ 。

将目标函数$\tilde{\mathcal{L}}^{(K)}$用上述符号替换，得到：

$$\begin{aligned}\tilde{\mathcal{L}}^{(K)} & =\sum_{i=1}^{n}\left[g_i f_K\left(\mathrm{x}_i\right)+\frac{1}{2} h_i f_K^2\left(\mathrm{x}_i\right)\right]+\gamma J+\frac{1}{2} \lambda \sum_{j=1}^{J} w_j^2 \\ & =\sum_{j=1}^{J}\left[\left(\sum_{i \in I_j} g_i\right) w_j+\frac{1}{2}\left(\sum_{i \in I_j} h_i+\lambda\right) w_j^2\right]+\gamma J\end{aligned}$$

令$G_j=\sum_{i \in I_j} g_i$，$H_j=\sum_{i \in I_j} h_i$。不难看出上式是一个关于w的二次函数，使用一阶条件得到：

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

以及

$$\tilde{\mathcal{L}}^{(K)}(q)=-\frac{1}{2}\sum_{j=1}^{J}\frac{(G_j)^2}{H_j+\lambda}+\gamma J$$

假设当前节点左右子树的一阶二阶导数和为G_L, H_L, G_R, H_L，我们希望每次做左右子树分裂时，可以最大程度地减少损失函数的损失。也即目标函数可以被写为：

$$\begin{aligned}&\max -\frac{1}{2}\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}+\gamma J-(-\frac{1}{2}\frac{G_L^2}{H_L+\lambda}-\frac{1}{2}\frac{G_R^2}{H_R+\lambda}+\gamma(J+1)\\&=\max\frac{1}{2}\frac{G_L^2}{H_L+\lambda}+\frac{1}{2}\frac{G_R^2}{H_R+\lambda}-\frac{1}{2}\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}-\gamma\end{aligned}$$

XGBoost 节点分裂准则

XGBoost在生成新树的过程中，最基本的操作是节点分裂。节点分裂中最重要的环节是找到最优特征及最优切分点, 然后将叶子节点按照最优特征和最优切分点进行分裂。

精确贪心分裂算法

首先找到所有的候 选特征及所有的候选切分点, 求得L_{split} , 然后选择 L_{split}最大的特征及 对应切分点作为最优特征和最优切分点。

算法伪代码如下：

输入：训练集样本$I=(x_1,y_1),(x_2,y_2),\dots(x_m,y_m)$， 最大迭代次数$K$, 损失函数$L$， 正则化系数λ, γ 。

输出：强学习器$f(x)$

对迭代轮数$t=1,2,\dots T$有：

1. 初始化分裂收益和梯度统计：$score=0$，$G=\sum\limits_{i=1}^{m}g_i$，$H=\sum\limits_{i=1}^{m}h_i$
2. 对特征序号$k=1,2...K$：
   - $G_L=0,H_L=0$
   - 将样本按特征k从小到大排列，依次取出第i个样本，依次计算当前样本放入左子树后，左右子树一阶和二阶导数和：
     $$G_L=G_L+g_j,H_L=H_L+h_j$$
     $$G_R=G-G_L,H_R=H-H_L$$
   - 尝试更新最大的分数：
     $$score=\max(score,\frac{1}{2}\frac{G_L^2}{H_L+\lambda}+\frac{1}{2}\frac{G_R^2}{H_R+\lambda}-\frac{1}{2}\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}-\gamma)$$
3. 基于最大$score$对应的划分特征和特征值分裂子树。
4. 如果最大$score$为0，则当前决策树建立完毕，计算所有叶子区域的w_j^t, 得到弱学习器$h^t(x)$，更新强学习器$f^t(x)$,进入下一轮弱学习器迭代。如果最大$score$不是0，则转到第2步继续尝试分裂决策树。

基于直方图的近似算法

贪心算法能够保证模型很好地拟合训练数据，但是当数据不能完全加载到内存时间，算法在计算过程中需要不断地在内存和磁盘之间进行数据交换，非常耗时。为了能够更高效地选择最优特征及切分点，XGBoost提出一种近似算法来解决该问题，也就是基于直方图的近似算法。

其主要思想为：对某一特征寻找最优切分点时，首先对该特征的所有切分点按分位数 (如百分位) 分桶, 得到一个候选切分点集。特征的每一个切分点都可以分到对应的分桶; 然后，对每个桶计算特征统计G和H得到直方图, G为该桶内所有样本**一阶**特征统计g之和, H为该桶内所有样本**二阶**特征统计h之和; 最后，选择所有候选特征及候选切分点中对应桶的特征统计收益最大的作为最优特征及最优切分点。

近似算法实现了两种候选切分点的构建策略：全局策略和本地策略。全局策略是在树构建的初始阶段对每一个特征确定一个候选切分点的集合, 并在该树每一层的节点分裂中均采用此集合计算收益, 整个过程候选切分点集合不改变。本地策略则是在每一次节点分裂时均重新确定候选切分点。全局策略需要更细的分桶才能达到本地策略的精确度, 但全局策略在选取候选切分点集合时比本地策略更简单。**在XGBoost系统中, 用户可以根据需求自由选择使用精确贪心算法、近似算法全局策略、近似算法本地策略, 算法均可通过参数进行配置。**

XGBoost调参

- 学习速率（最重要，通常最先进行调整）
- max_depth 和 min_child_weight
- gamma参数和正则化参数alpha
- subsample 和 colsample_bytree

XGBoost实践

导入相关包

```
import pandas as pd
from sklearn import datasets
from sklearn.preprocessing import StandardScaler
from sklearn.pipeline import make_pipeline
from sklearn.model_selection import cross_val_score
from sklearn.tree import DecisionTreeClassifier
from xgboost import XGBClassifier
```

读入数据

```
iris = datasets.load_iris()
X = iris.data
y = iris.target
feature = iris.feature_names
data = pd.DataFrame(X, columns=feature)
data['target'] = y
```

准备模型

```
pipe_dt = make_pipeline(StandardScaler(), DecisionTreeClassifier())
xgb = XGBClassifier(learning_rate=0.25)
pipe_xgb = make_pipeline(StandardScaler(), xgb)
models = [("dt", pipe_dt), ("xgboost", pipe_xgb)]
```

结果对比

```
def evaluate_model(model, X, y):
    score = cross_val_score(model, X, y, scoring='accuracy', cv=5,
error_score='raise')
    return score
for (name, model) in models:
    score = evaluate_model(model, X, y)
    print(f"Model:{name}; Mean: {score.mean():.3f}; Std: {score.std():.3f}")
```

> Model:dt; Mean: 0.960; Std: 0.033
> Model:xgboost; Mean: 0.960; Std: 0.025

可以看到，由于iris数据集比较简单，xgboost模型与决策树模型的分类效果差不多，但是XGBoost提升了模型的鲁棒性，分类准确率的方差更小。