

# GBDT算法原理与系统设计简介

wepon

<http://wepon.me>

2017. 09. 20

# 内容

- 泰勒公式
- 最优化方法
  - 梯度下降法 (Gradient descend method)
  - 牛顿法 (Newton's method)
- 从参数空间到函数空间
  - 从Gradient descend 到 Gradient boosting
  - 从Newton's method 到 Newton Boosting
- Gradient Boosting Tree 算法原理
- Newton Boosting Tree 算法原理: 详解 XGBoost
- 更高效的工具包 LightGBM
- 参考文献

# 泰勒公式

- 定义：泰勒公式是一个用函数在某点的信息描述其附近取值的公式。局部有效性

# 泰勒公式

- 定义：泰勒公式是一个用函数在某点的信息描述其附近取值的公式。局部有效性

- 基本形式：
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n$$

- 一阶泰勒展开： $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$

- 二阶泰勒展开： $f(x) \approx f(x_0) + f'(x_0)(x - x_0) + f''(x_0) \frac{(x - x_0)^2}{2}$

# 泰勒公式

- 定义：泰勒公式是一个用函数在某点的信息描述其附近取值的公式。局部有效性

- 基本形式：
$$f(x) = \sum_{n=0}^{\infty} \frac{f^{(n)}(x_0)}{n!} (x - x_0)^n$$

- 一阶泰勒展开： $f(x) \approx f(x_0) + f'(x_0)(x - x_0)$

- 二阶泰勒展开： $f(x) \approx f(x_0) + f'(x_0)(x - x_0) + f''(x_0) \frac{(x - x_0)^2}{2}$

- 迭代形式：假设  $x^t = x^{t-1} + \Delta x$ ，将  $f(x^t)$  在  $x^{t-1}$  处进行泰勒展开

$$\begin{aligned} f(x^t) &= f(x^{t-1} + \Delta x) \\ &\approx f(x^{t-1}) + f'(x^{t-1})\Delta x + f''(x^{t-1})\frac{\Delta x^2}{2} \end{aligned}$$

# 梯度下降法（Gradient Descend Method）

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。

# 梯度下降法（Gradient Descend Method）

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。

- 迭代公式： $\theta^t = \theta^{t-1} + \Delta\theta$

# 梯度下降法（Gradient Descend Method）

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。

- 迭代公式： $\theta^t = \theta^{t-1} + \Delta\theta$
- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行一阶泰勒展开：

$$\begin{aligned} L(\theta^t) &= L(\theta^{t-1} + \Delta\theta) \\ &\approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta \end{aligned}$$



# 梯度下降法（Gradient Descend Method）

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。

- 迭代公式： $\theta^t = \theta^{t-1} + \Delta\theta$
- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行一阶泰勒展开：

$$\begin{aligned} L(\theta^t) &= L(\theta^{t-1} + \Delta\theta) \\ &\approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta \end{aligned}$$

- 要使得  $L(\theta^t) < L(\theta^{t-1})$ ，可取： $\Delta\theta = -\alpha L'(\theta^{t-1})$ ，则： $\theta^t = \theta^{t-1} - \alpha L'(\theta^{t-1})$

# 梯度下降法（Gradient Descend Method）

在机器学习任务中，需要最小化损失函数  $L(\theta)$ ，其中  $\theta$  是要求解的模型参数。梯度下降法常用来求解这种无约束最优化问题，它是一种迭代方法：选取初值  $\theta^0$ ，不断迭代，更新  $\theta$  的值，进行损失函数的极小化。

- 迭代公式：  $\theta^t = \theta^{t-1} + \Delta\theta$
- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行一阶泰勒展开：

$$\begin{aligned} L(\theta^t) &= L(\theta^{t-1} + \Delta\theta) \\ &\approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta \end{aligned}$$

- 要使得  $L(\theta^t) < L(\theta^{t-1})$ ，可取：  $\Delta\theta = -\alpha L'(\theta^{t-1})$ ，则：  $\theta^t = \theta^{t-1} - \alpha L'(\theta^{t-1})$

这里  $\alpha$  是步长，可通过line search确定，但一般直接赋一个小的数。

# 牛顿法 (Newton's Method)

- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行二阶泰勒展开:

$$L(\theta^t) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta + L''(\theta^{t-1})\frac{\Delta\theta^2}{2}$$

# 牛顿法 (Newton's Method)

- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行二阶泰勒展开:

$$L(\theta^t) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta + L''(\theta^{t-1})\frac{\Delta\theta^2}{2}$$

为了简化分析过程, 假设参数是标量 (即  $\theta$  只有一维), 则可将一阶和二阶导数分别记为  $g$  和  $h$ :

$$L(\theta^t) \approx L(\theta^{t-1}) + g\Delta\theta + h\frac{\Delta\theta^2}{2}$$

# 牛顿法 (Newton's Method)

- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行二阶泰勒展开:

$$L(\theta^t) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta + L''(\theta^{t-1})\frac{\Delta\theta^2}{2}$$

为了简化分析过程, 假设参数是标量 (即  $\theta$  只有一维), 则可将一阶和二阶导数分别记为  $g$  和  $h$ :

$$L(\theta^t) \approx L(\theta^{t-1}) + g\Delta\theta + h\frac{\Delta\theta^2}{2}$$

- 要使得  $L(\theta^t)$  极小, 即让  $g\Delta\theta + h\frac{\Delta\theta^2}{2}$  极小, 可令: 
$$\frac{\partial \left( g\Delta\theta + h\frac{\Delta\theta^2}{2} \right)}{\partial \Delta\theta} = 0$$

求得  $\Delta\theta = -\frac{g}{h}$ , 故  $\theta^t = \theta^{t-1} + \Delta\theta = \theta^{t-1} - \frac{g}{h}$

# 牛顿法 (Newton's Method)

- 将  $L(\theta^t)$  在  $\theta^{t-1}$  处进行二阶泰勒展开:

$$L(\theta^t) \approx L(\theta^{t-1}) + L'(\theta^{t-1})\Delta\theta + L''(\theta^{t-1})\frac{\Delta\theta^2}{2}$$

为了简化分析过程, 假设参数是标量 (即  $\theta$  只有一维), 则可将一阶和二阶导数分别记为  $g$  和  $h$ :

$$L(\theta^t) \approx L(\theta^{t-1}) + g\Delta\theta + h\frac{\Delta\theta^2}{2}$$

- 要使得  $L(\theta^t)$  极小, 即让  $g\Delta\theta + h\frac{\Delta\theta^2}{2}$  极小, 可令: 
$$\frac{\partial \left( g\Delta\theta + h\frac{\Delta\theta^2}{2} \right)}{\partial \Delta\theta} = 0$$

$$\text{求得 } \Delta\theta = -\frac{g}{h}, \text{ 故 } \theta^t = \theta^{t-1} + \Delta\theta = \theta^{t-1} - \frac{g}{h}$$

参数  $\theta$  推广到向量形式, 迭代公式:  $\theta^t = \theta^{t-1} - H^{-1}g$

这里  $H$  是海森矩阵

# 从参数空间到函数空间

- GBDT 在函数空间中利用梯度下降法进行优化
- XGBoost 在函数空间中用牛顿法进行优化

注：实际上GBDT泛指所有梯度提升树算法，包括XGBoost，它也是GBDT的一种变种，这里为了区分它们，GBDT特指“Greedy Function Approximation: A Gradient Boosting Machine”里提出的算法，它只用了一阶导数信息。

# 从Gradient Descend 到 Gradient Boosting

参数空间

$$\theta^t = \theta^{t-1} + \theta_t$$

第t次迭代  
后的参数

第t-1次迭  
代后的参数

第t次迭代  
的参数增量

$$\theta_t = -\alpha_t g_t$$

参数更新方向为负梯度方向

$$\theta = \sum_{t=0}^T \theta_t$$

最终参数等于每次迭代的增量的累加和，

$\theta_0$  为初值



# 从Gradient Descend 到 Gradient Boosting

参数空间

$$\theta^t = \theta^{t-1} + \theta_t$$

第t次迭代  
后的参数

第t-1次迭  
代后的参数

第t次迭代  
的参数增量

$$\theta_t = -\alpha_t g_t$$

参数更新方向为负梯度方向

$$\theta = \sum_{t=0}^T \theta_t$$

最终参数等于每次迭代的增量的累加和，

$\theta_0$  为初值

函数空间

$$f^t(x) = f^{t-1}(x) + f_t(x)$$

第t次迭代  
后的函数

第t-1次迭  
代后的函数

第t次迭代  
的函数增量

$$f_t(x) = -\alpha_t g_t(x)$$

同样地，拟合负梯度

$$F(x) = \sum_{t=0}^T f_t(x)$$

最终函数等于每次迭代的增量的累加和，

$f_0(x)$  为模型初始值，通常为常数

# 从Newton's Method 到 Newton Boosting

参数空间

$$\theta^t = \theta^{t-1} + \theta_t$$

第t次迭代  
后的参数

第t-1次迭  
代后的参数

第t次迭代  
的参数增量

$$\theta_t = -H_t^{-1} g_t$$

与梯度下降法唯一不同的就是参数增量

$$\theta = \sum_{t=0}^T \theta_t$$

最终参数等于每次迭代的增量的累加和，

$\theta_0$  为初值

函数空间

$$f^t(x) = f^{t-1}(x) + f_t(x)$$

第t次迭代  
后的函数

第t-1次迭  
代后的函数

第t次迭代  
的函数增量

$$f_t(x) = -\frac{g_t(x)}{h_t(x)}$$

最终函数等于每次迭代的增量的累加和，

$f_0(x)$  为模型初始值，通常为常数

# 小结

- Boosting 算法是一种加法模型（additive training）

$$F(x) = \sum_{t=0}^T f_t(x)$$

# 小结

- Boosting 算法是一种加法模型（additive training）

$$F(x) = \sum_{t=0}^T f_t(x)$$

- 基分类器  $f$  常采用回归树[Friedman 1999] 和逻辑回归[Friedman 2000]。下文以回归树展开介绍。树模型有以下优缺点：

- 可解释性强
- 可处理混合类型特征
- 具体伸缩不变性（不用归一化特征）
- 有特征组合的作用
- 可自然地处理缺失值
- 对异常点鲁棒
- 有特征选择作用
- 可扩展性强，容易并行
- 缺乏平滑性（回归预测时输出值只能输出有限的若干种数值）
- 不适合处理高维稀疏数据

# Gradient Boosting Tree 算法原理

- Friedman于论文“[Greedy Function Approximation...](#)”中最早提出GBDT

# Gradient Boosting Tree 算法原理

- Friedman于论文“**Greedy Function Approximation...**”中最早提出GBDT
- 其模型  $F$  定义为加法模型：

$$F(x; w) = \sum_{t=0}^T \alpha_t h_t(x; w_t) = \sum_{t=0}^T f_t(x; w_t)$$

其中， $x$ 为输入样本， $h$ 为分类回归树， $w$ 是分类回归树的参数， $\alpha$  是每棵树的权重。

# Gradient Boosting Tree 算法原理

- Friedman于论文” **Greedy Function Approximation...**”中最早提出GBDT
- 其模型  $F$  定义为加法模型：

$$F(x; w) = \sum_{t=0}^T \alpha_t h_t(x; w_t) = \sum_{t=0}^T f_t(x; w_t)$$

其中， $x$ 为输入样本， $h$ 为分类回归树， $w$ 是分类回归树的参数， $\alpha$  是每棵树的权重。

- 通过最小化损失函数求解最优模型：

$$F^* = \arg \min_F \sum_{i=0}^N L(y_i, F(x_i; w))$$

NP难问题 -> 通过贪心法，迭代求局部最优解

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$

2. for t = 1 to T do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第t棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$



# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$

2. for t = 1 to T do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第t棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应：

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树：

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长：

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型：

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树:

$$w^* = \arg \min_w \sum_{i=1}^N \left( \tilde{y}_i - h_t(x_i; w) \right)^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Gradient Boosting Tree 算法原理

输入：  $(x_i, y_i), T, L$

1. 初始化  $f_0$
2. for  $t = 1$  to  $T$  do

2.1 计算响应:

$$\tilde{y}_i = - \left[ \frac{\partial L(y_i, F(x_i))}{\partial F(x_i)} \right]_{F(x)=F_{t-1}(x)}, i=1,2,\dots,N$$

2.2 学习第 $t$ 棵树:

$$w^* = \arg \min_w \sum_{i=1}^N (\tilde{y}_i - h_t(x_i; w))^2$$

2.3 line search找步长:

$$\rho^* = \arg \min_{\rho} \sum_{i=1}^N L(y_i, F_{t-1}(x_i) + \rho h_t(x_i; w^*))$$

2.4 令  $f_t = \rho^* h_t(x; w^*)$

更新模型:

$$F_t = F_{t-1} + f_t$$

3. 输出  $F_T$

# Newton Boosting Tree 算法原理：详解 XGBoost

- 模型函数形式
- 目标函数
  - 正则项
  - 误差函数的二阶泰勒展开
- 回归树的学习策略
  - 打分函数
  - 树节点分裂方法
  - 缺失值的处理
- 其它特性



# 模型函数形式

给定数据集  $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ ，XGBoost进行additive training，学习K棵树，采用以下函数对样本进行预测：

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^K f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

F表示假设空间：监督学习的目的在于学习一个由输入到输出的映射，这一映射由模型来表示，学习的目的就在于找到最好的这样的模型，模型属于由输入空间到输出空间的映射的集合，这个集合就是假设空间F，假设空间的确定意味着学习范围的确定。

# 模型函数形式

给定数据集  $\mathcal{D} = \{(\mathbf{x}_i, y_i)\}$ ，XGBoost进行additive training，学习K棵树，采用以下函数对样本进行预测：

$$\hat{y}_i = \phi(\mathbf{x}_i) = \sum_{k=1}^K f_k(\mathbf{x}_i), \quad f_k \in \mathcal{F},$$

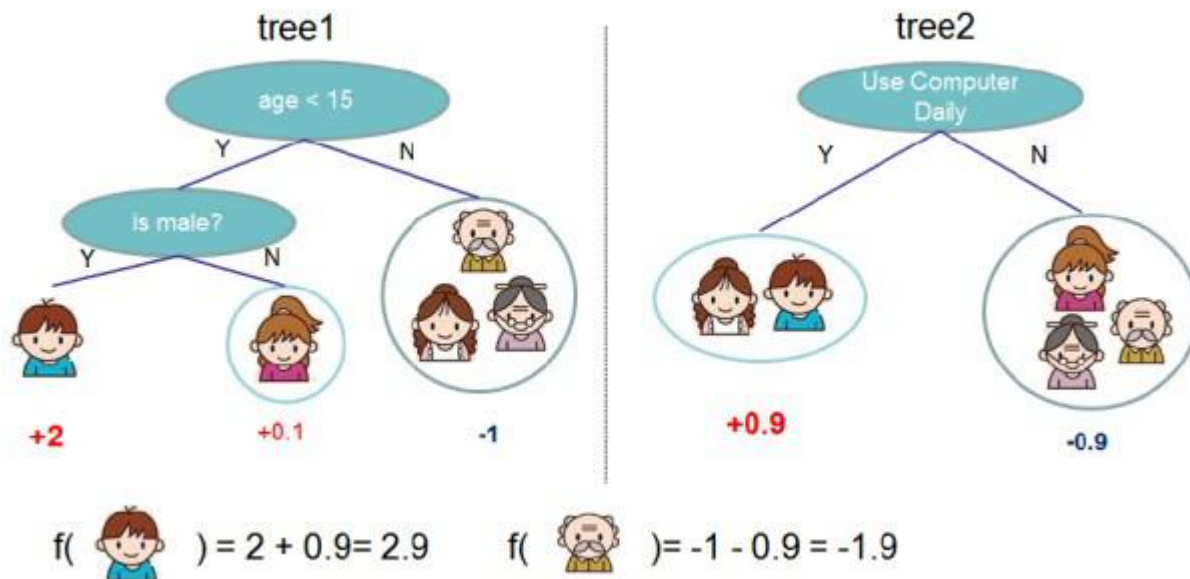
这里  $\mathcal{F}$  是假设空间， $f(\mathbf{x})$  是回归树（CART）：

$$\mathcal{F} = \{f(\mathbf{x}) = w_{q(\mathbf{x})}\}(q : \mathbb{R}^m \rightarrow T, w \in \mathbb{R}^T)$$

$q(\mathbf{x})$ 表示将样本 $\mathbf{x}$ 分到了某个叶子节点上， $w$ 是叶子节点的分数（leaf score），所以  $w_{q(\mathbf{x})}$  表示回归树对样本的预测值

# 模型函数形式

- 例子：预测一个人是否喜欢电脑游戏



回归树的预测输出是实数分数，可以用于回归、分类、排序等任务中。对于回归问题，可以直接作为目标值，对于分类问题，需要映射成概率，比如采用逻辑函数  $\sigma(z) = \frac{1}{1 + e^{-z}}$

# 目标函数

- 参数空间中的目标函数：

$$Obj(\Theta) = L(\Theta) + \Omega(\Theta)$$

误差函数：我们的模型有多拟合数据。

正则化项：惩罚复杂模型

误差函数可以是square loss, logloss等，正则项可以是L1正则，L2正则等。

Ridge Regression（岭回归）：
$$\sum_{i=1}^n (y_i - \theta^T x_i)^2 + \lambda \|\theta\|^2$$

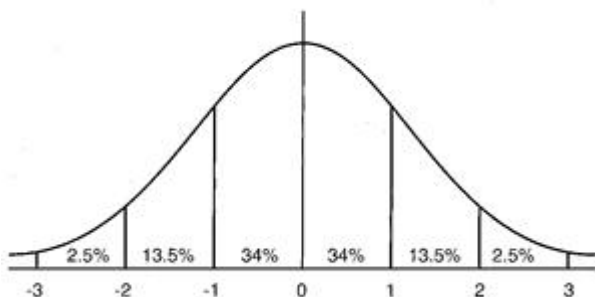
LASSO：
$$\sum_{i=1}^n (y_i - \theta^T x_i)^2 + \lambda \|\theta\|_1$$

# 正则项

- 正则项的作用，可以从几个角度去解释：
  - 通过偏差方差分解去解释
  - PAC-learning泛化界解释
  - Bayes先验解释，把正则当成先验

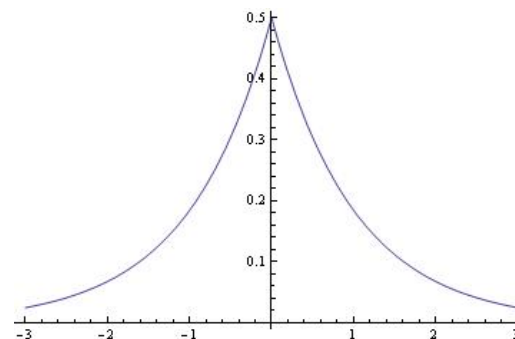
# 正则项

- 正则项的作用，可以从几个角度去解释：
  - 通过偏差方差分解去解释
  - PAC-learning泛化界解释
  - Bayes先验解释，把正则当成先验
- 从Bayes角度来看，正则相当于对模型参数引入先验分布：



L2正则，模型参数服从高斯分布  $\theta \sim N(0, \sigma^2)$

对参数加了分布约束，大部分绝对值很小



L1正则，模型参数服从拉普拉斯分布

对参数加了分布约束，大部分取值为0

# 正则项

- XGBoost的目标函数（函数空间）

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

正则项对每棵回归树的复杂度进行了惩罚

# 正则项

- XGBoost的目标函数（函数空间）

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

正则项对每棵回归树的复杂度进行了惩罚

- 相比原始的GBDT，XGBoost的目标函数多了正则项，使得学习出来的模型更加不容易过拟合。



# 正则项

- XGBoost的目标函数（函数空间）

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

正则项对每棵回归树的复杂度进行了惩罚

- 相比原始的GBDT，XGBoost的目标函数多了正则项，使得学习出来的模型更加不容易过拟合。
- 有哪些指标可以衡量树的复杂度？

树的深度，内部节点个数，叶子节点个数( $T$ )，叶节点分数( $w$ )...

XGBoost采用的：

$$\Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

对叶子节点个数进行惩罚，相当于在训练过程中做了剪枝

# 误差函数的二阶泰勒展开

- 第 $t$ 次迭代后，模型的预测等于前 $t-1$ 次的模型预测加上第 $t$ 棵树的预测：

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$$

# 误差函数的二阶泰勒展开

- 第 $t$ 次迭代后，模型的预测等于前 $t-1$ 次的模型预测加上第 $t$ 棵树的预测：

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$$

- 此时目标函数可写作：

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

公式中  $y_i, \tilde{y}_i^{(t-1)}$  都已知，模型要学习的只有第 $t$ 棵树  $f_t$

# 误差函数的二阶泰勒展开

- 第 $t$ 次迭代后，模型的预测等于前 $t-1$ 次的模型预测加上第 $t$ 棵树的预测：

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i)$$

- 此时目标函数可写作：

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

公式中  $y_i, \tilde{y}_i^{(t-1)}$  都已知，模型要学习的只有第 $t$ 棵树  $f_t$

- 将误差函数在  $\tilde{y}_i^{(t-1)}$  处进行二阶泰勒展开：

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^n [l(y_i, \hat{y}_i^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

公式中， $g_i = \partial_{\hat{y}^{(t-1)}} l(y_i, \hat{y}_i^{(t-1)})$      $h_i = \partial_{\hat{y}^{(t-1)}}^2 l(y_i, \hat{y}_i^{(t-1)})$

## 误差函数的二阶泰勒展开

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^n [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

- 将公式中的常数项去掉，得到：

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

## 误差函数的二阶泰勒展开

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^n [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

- 将公式中的常数项去掉，得到：

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

- 把  $f_t, \Omega(f_t)$  写成树结构的形式，即把下式代入目标函数中

$$f(\mathbf{x}) = w_{q(\mathbf{x})} \quad \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

## 误差函数的二阶泰勒展开

$$\mathcal{L}^{(t)} \simeq \sum_{i=1}^n [l(y_i, \hat{y}^{(t-1)}) + g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i)] + \Omega(f_t)$$

- 将公式中的常数项去掉，得到：

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

- 把  $f_t, \Omega(f_t)$  写成树结构的形式，即把下式代入目标函数中



$$f(\mathbf{x}) = w_{q(\mathbf{x})} \quad \Omega(f) = \gamma T + \frac{1}{2} \lambda \|w\|^2$$

- 得到：

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

# 误差函数的二阶泰勒展开

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

 对样本累加                       对叶节点累加

- 怎么统一起来？

有人可能看不懂下一页的公式转换，补充解释如下：

先根据j的定义，举例现在有4个样本。两类办法可以求最小误差——

1-1、那么本来的误差求和只需要对4个样本的真实值和预测值分别计算误差并求和。

假设根据树的分类，现在将4个样本分别分配到了2个叶节点，那么新的计算误差的方式就变成如下：

2-1、先对每个叶子节点里面的所有样本进行误差计算并求和。

2-2、对每个叶子节点的误差再次求和。这样计算的结果实际上和1-1类的计算方式完全一致。



# 误差函数的二阶泰勒展开

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

对样本累加

对叶节点累加

- 怎么统一起来？

定义每个叶节点j上的样本集合为  $I_j = \{i | q(x_i) = j\}$

则目标函数可以写成按叶节点累加的形式：

$$\begin{aligned}\tilde{L}^{(t)} &= \sum_{j=1}^T \left[ \left( \sum_{i \in I_j} g_i \right) w_j + \frac{1}{2} \left( \sum_{i \in I_j} h_i + \lambda \right) w_j^2 \right] + \gamma T \\ &= \sum_{j=1}^T \left[ \mathbf{G}_j w_j + \frac{1}{2} (\mathbf{H}_j + \lambda) w_j^2 \right] + \gamma T\end{aligned}$$

# 误差函数的二阶泰勒展开

若树的结构确定，那么后面的  $\gamma T$  就可以直接视为常数项，常数项的导数为0。

$$\tilde{L}^{(t)} = \sum_{j=1}^T \left[ G_j w_j + \frac{1}{2} (H_j + \lambda) w_j^2 \right] + \gamma T$$

- 如果确定了树的结构（即 $q(x)$ 确定），为了使目标函数最小，可以令其导数为0，解得每个叶节点的最优预测分数为：

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

注：在向量求导的时候，可以忽略下标，那么求导公式完毕后再加回下标，是一种较好的理解方式。

代入目标函数，得到最小损失为：

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

模型实际上都是在求参数向量，一般方式为先对目标函数置导数为0求极值，然后将求得的参数向量 $w_j$ 代回目标函数，那么就可以求得最小损失。

# 回归树的学习策略

- 当回归树的结构确定时，我们前面已经推导出其最优的叶节点分数以及对应的最小损失值，问题是怎么确定树的结构？

暴力枚举所有可能的树结构，选择损失值最小的 – NP难问题

贪心法，每次尝试分裂一个叶节点，计算分裂前后的增益，选择增益最大的

# 回归树的学习策略

- 当回归树的结构确定时，我们前面已经推导出其最优的叶节点分数以及对应的最小损失值，问题是怎么确定树的结构？

暴力枚举所有可能的树结构，选择损失值最小的 – NP难问题

贪心法，每次尝试分裂一个叶节点，计算分裂前后的增益，选择增益最大的

- 分裂前后的增益怎么计算？

ID3算法采用信息增益

C4.5算法采用信息增益比

CART采用Gini系数

XGBoost呢？

# XGBoost的打分函数

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

标红部分衡量了每个叶子节点对总体损失的贡献，我们希望损失越小越好，则标红部分的值越大越好。

这个公式很好理解，要求一个函数的最小值，那么可以拆分上述函数的两部分，对前半部分求最小值，对后半部分也求最小值，那么相加的值即为整个函数的最小值。

注意，前面部分是负号，所以红色部分是越大越好。

分类树是 CART 中用来分类的了，不同于 ID3 与 C4.5，分类树采用基尼指数来选择最优的切分特征，而且每次都是二分。

基尼指数是一个类似与熵的概念，对于一个有  $K$  种状态对应的概率为  $p_1, p_2, \dots, p_K$  的随机变量  $X$ ，其  $Gini$  定义如下：

$$Gini(X) = \sum_k p_k(1 - p_k) = 1 - \sum_k p_k^2$$

根据公式可得到伯努利分布  $X \sim Bernoulli(p)$  的基尼系数为：

$$Gini(X) = \sum_k p_k(1 - p_k) = 2p(1 - p)$$

对于训练数据集  $D$ ，假设共有  $K$  个类别， $C_k$  代表第  $k$  类的样本子集， $|C_k|$  为  $C_k$  的大小， $|D|$  为  $D$  的大小，则集合  $D$  的基尼系数为：

$$Gini(D) = \sum_k \frac{|C_k|}{|D|} \left(1 - \frac{|C_k|}{|D|}\right) = 1 - \sum_k \left(\frac{|C_k|}{|D|}\right)^2$$

类似于 ID3 中的信息增益，假设现在用特征  $A$  对数据进行分割，若特征  $A$  为离散特征，则根据  $A$  的某一可能取值  $a$  将  $D$  分为  $D_1$  与  $D_2$

$$D_1 = \{D|A = a\} \quad D_2 = \{D|A \neq a\}$$

其实就跟回归树一样，对于连续特征，也类似于回归树。接下来便得到类似于条件熵的一个量  $Gini(D, A)$ ，即在已知特征  $A$  的条件下集合  $D$  的基尼指数：

$$Gini(D, A) = \frac{|D_1|}{|D|} Gini(D_1) + \frac{|D_2|}{|D|} Gini(D_2)$$

下部分的公式定义：

# XGBoost的打分函数

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

$$\begin{aligned} Obj_{split} &= -\frac{1}{2} \left[ \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} \right] + \gamma T_{split} \\ Obj_{noSplit} &= -\frac{1}{2} \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} + \gamma T_{noSplit} \\ Gain &= Obj_{noSplit} - Obj_{split} \\ &= \frac{1}{2} \left[ \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - \gamma (T_{split} - T_{noSplit}) \end{aligned}$$

标红部分衡量了每个叶子节点对总体损失的贡献，我们希望损失越小越好，则标红部分的值越大越好。

因此，对一个叶子节点进行分裂，分裂前后的增益定义为：

$$Gain = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} - \gamma$$

Gain的值越大，分裂后 L 减小越多。所以当对一个叶节点分割时，计算所有候选(feature, value)对应的gain，选取gain最大的进行分割

以上公式是根据标红部分而定义的一个基尼函数，此处可以和CART的基尼系数进行对比。具体请参考上页。

特别需要说明一点，为何基尼值越大，分裂后的L减小就越多呢？

实际上，切分前的Gain值是固定的。假设Gain\_nosplit = 10,现在有几个分裂节点供我们选择：

1.设Gain = 2,则Gain\_spilt = 10 - 2 = 8

2.设Gain = 1,则Gain\_spilt = 10 - 1 = 9

3.设Gain = 7,则Gain\_spilt = 10 - 7 = 3。至此，说明Gain值越大，则分类后的L减小越多，那么这个叶子节点就比其他叶子节点更适合分割。

# 树节点分裂方法 (Split Finding)

- 精确算法

遍历所有特征的所有可能的分割点，计算gain值，选取值最大的 (feature, value) 去分割

---

**Algorithm 1:** Exact Greedy Algorithm for Split Finding

---

**Input:**  $I$ , instance set of current node

**Input:**  $d$ , feature dimension

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

**for**  $k = 1$  **to**  $m$  **do**

$G_L \leftarrow 0, H_L \leftarrow 0$

**for**  $j$  in sorted( $I$ , by  $\mathbf{x}_{jk}$ ) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

**end**

**end**

**Output:** Split with max score

---

# 树节点分裂方法 (Split Finding)

- 近似算法

对于每个特征，只考察分位点，减少计算复杂度

- Global: 学习每棵树前，提出候选切分点
- Local: 每次分裂前，重新提出候选切分点

---

**Algorithm 2:** Approximate Algorithm for Split Finding

---

**for**  $k = 1$  **to**  $m$  **do**

    | Propose  $S_k = \{s_{k1}, s_{k2}, \dots, s_{kl}\}$  by percentiles on feature  $k$ .  
    | Proposal can be done per tree (global), or per split(local).

**end**

**for**  $k = 1$  **to**  $m$  **do**

    |  $G_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} g_j$

    |  $H_{kv} \leftarrow \sum_{j \in \{j | s_{k,v} \geq \mathbf{x}_{jk} > s_{k,v-1}\}} h_j$

**end**

Follow same step as in previous section to find max score only among proposed splits.

---



# 树节点分裂方法（Split Finding）

- 近似算法举例：三分位数

|          |            |       |                |            |       |                |            |       |       |
|----------|------------|-------|----------------|------------|-------|----------------|------------|-------|-------|
|          |            |       | 1/3 percentile |            |       | 2/3 percentile |            |       |       |
| features | 1          | 1     | 3              | 4          | 5     | 12             | 45         | 50    | 99    |
| labels   | 1          | 0     | 0              | 1          | 1     | 0              | 0          | 0     | 0     |
| $g_i$    | $g_1$      | $g_2$ | $g_3$          | $g_4$      | $g_5$ | $g_6$          | $g_7$      | $g_8$ | $g_9$ |
| $h_i$    | $h_1$      | $h_2$ | $h_3$          | $h_4$      | $h_5$ | $h_6$          | $h_7$      | $h_8$ | $h_9$ |
|          | $G_1, H_1$ |       |                | $G_2, H_2$ |       |                | $G_3, H_3$ |       |       |

$$Gain = \max\left\{ Gain, \frac{G_1^2}{H_1 + \lambda} + \frac{G_{23}^2}{H_{23} + \lambda} - \frac{G_{123}^2}{H_{123} + \lambda} - \gamma, \right. \\ \left. \frac{G_{12}^2}{H_{12} + \lambda} + \frac{G_3^2}{H_3 + \lambda} - \frac{G_{123}^2}{H_{123} + \lambda} - \gamma \right\}$$

此公式实际还可以补充一个 $H_2$ 的分位点，其他分类数据为 $H_{13}$

# 树节点分裂方法 (Split Finding)

- 实际上XGBoost不是简单地按照样本个数进行分位，而是以二阶导数值作为权重 (Weighted Quantile Sketch)，比如：

|          |     |     |     |     |     |     |     |     |     |
|----------|-----|-----|-----|-----|-----|-----|-----|-----|-----|
| features | 1   | 1   | 3   | 4   | 5   | 12  | 45  | 50  | 99  |
| hi       | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.1 | 0.4 | 0.2 | 0.6 |

1/3 percentile      2/3 percentile

- 为什么用hi加权？

把目标函数整理成以下形式，可以看出hi有对loss加权的作用

$$\sum_{i=1}^n \frac{1}{2} h_i (f_t(\mathbf{x}_i) - g_i/h_i)^2 + \Omega(f_t) + constant,$$

# 稀疏值处理

- 稀疏值：  
缺失，类别one-hot编码，大量0值
- 当特征出现缺失值时，XGBoost  
可以学习出默认的分点分裂方向

---

## Algorithm 3: Sparsity-aware Split Finding

---

**Input:**  $I$ , instance set of current node

**Input:**  $I_k = \{i \in I | x_{ik} \neq \text{missing}\}$

**Input:**  $d$ , feature dimension

*Also applies to the approximate setting, only collect statistics of non-missing entries into buckets*

$gain \leftarrow 0$

$G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$

**for**  $k = 1$  **to**  $m$  **do**

*// enumerate missing value goto right*

$G_L \leftarrow 0, H_L \leftarrow 0$

**for**  $j$  *in sorted*( $I_k$ , *ascent order by*  $x_{jk}$ ) **do**

$G_L \leftarrow G_L + g_j, H_L \leftarrow H_L + h_j$

$G_R \leftarrow G - G_L, H_R \leftarrow H - H_L$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

**end**

*// enumerate missing value goto left*

$G_R \leftarrow 0, H_R \leftarrow 0$

**for**  $j$  *in sorted*( $I_k$ , *descent order by*  $x_{jk}$ ) **do**

$G_R \leftarrow G_R + g_j, H_R \leftarrow H_R + h_j$

$G_L \leftarrow G - G_R, H_L \leftarrow H - H_R$

$score \leftarrow \max(score, \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda})$

**end**

**end**

**Output:** Split and default directions with max gain

---

# XGBoost的其它特性

- 行抽样 (row sample)
- 列抽样 (column sample)  
借鉴随机森林
- Shrinkage (缩减), 即学习速率  
将学习速率调小, 迭代次数增多, 有正则化作用
- 支持自定义损失函数 (需二阶可导)

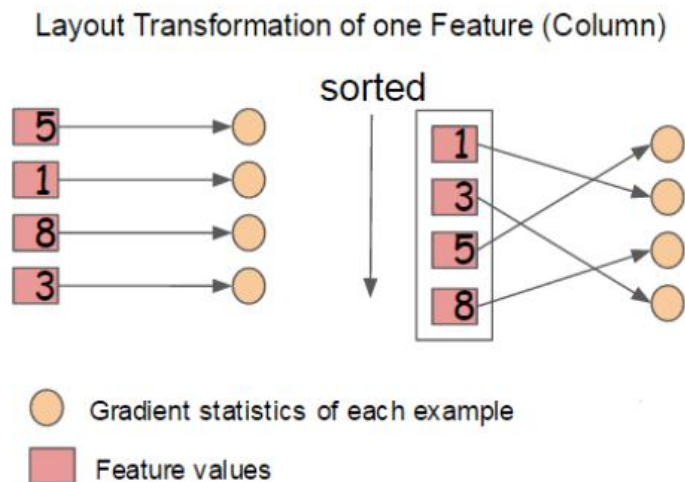
```
def logloss(preds, dtrain):  
    labels = dtrain.get_label()  
    # 将预测的score转化为概率  
    preds = 1.0 / (1.0 + np.exp(-preds))  
    grad = preds - labels  
    hess = preds * (1.0 - preds)  
    return grad, hess
```

# XGBoost的系统设计

- **Column Block**

- 特征预排序，以column block的结构存于内存中
- 存储样本索引（instance indices）
- block中的数据以稀疏格式（CSC）存储

这个结构加速了split finding的过程，只需要在建树前排序一次，后面节点分裂时直接根据索引得到梯度信息



- **Cache Aware Access**

- column block按特征大小顺序存储，相应的样本的梯度信息是分散的，造成内存的不连续访问，降低CPU cache 命中率
- 缓存优化方法
  - 预取数据到buffer中（非连续->连续），再统计梯度信息
  - 调节块的大小

# 更高效的工具包 LightGBM

- 速度更快

| Data      | xgboost   | xgboost_hist | LightGBM     |
|-----------|-----------|--------------|--------------|
| Higgs     | 3794.34 s | 551.898 s    | 238.505513 s |
| Yahoo LTR | 674.322 s | 265.302 s    | 150.18644 s  |
| MS LTR    | 1251.27 s | 385.201 s    | 215.320316 s |
| Expo      | 1607.35 s | 588.253 s    | 138.504179 s |
| Allstate  | 2867.22 s | 1355.71 s    | 348.084475 s |

- 内存占用更低

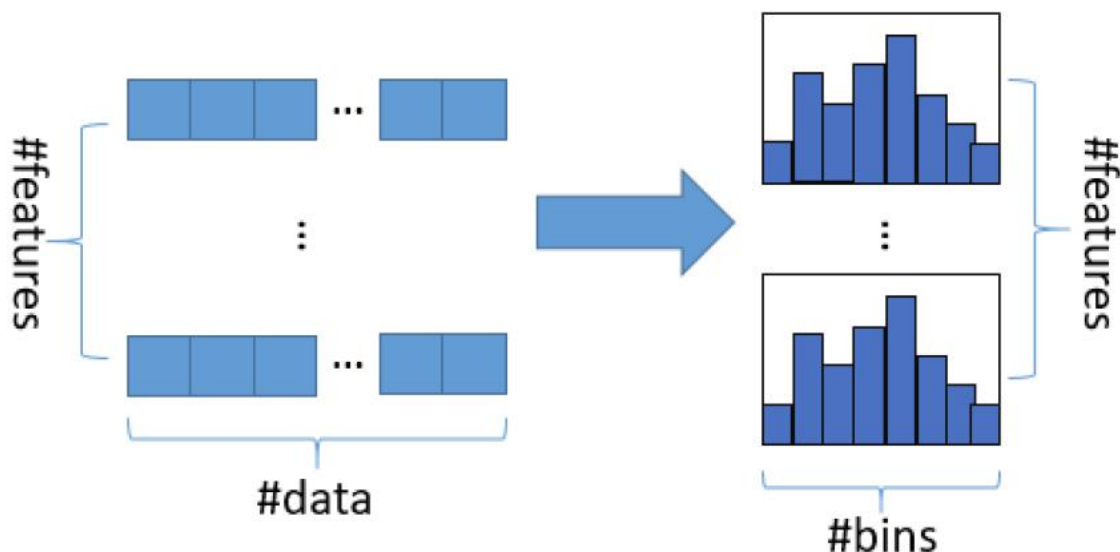
| Data      | xgboost | xgboost_hist | LightGBM |
|-----------|---------|--------------|----------|
| Higgs     | 4.853GB | 3.784GB      | 0.868GB  |
| Yahoo LTR | 1.907GB | 1.468GB      | 0.831GB  |
| MS LTR    | 5.469GB | 3.654GB      | 0.886GB  |
| Expo      | 1.553GB | 1.393GB      | 0.543GB  |
| Allstate  | 6.237GB | 4.990GB      | 1.027GB  |

- 准确率更高（优势不明显，与XGBoost相当）

# LightGBM的改进

- 直方图算法

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



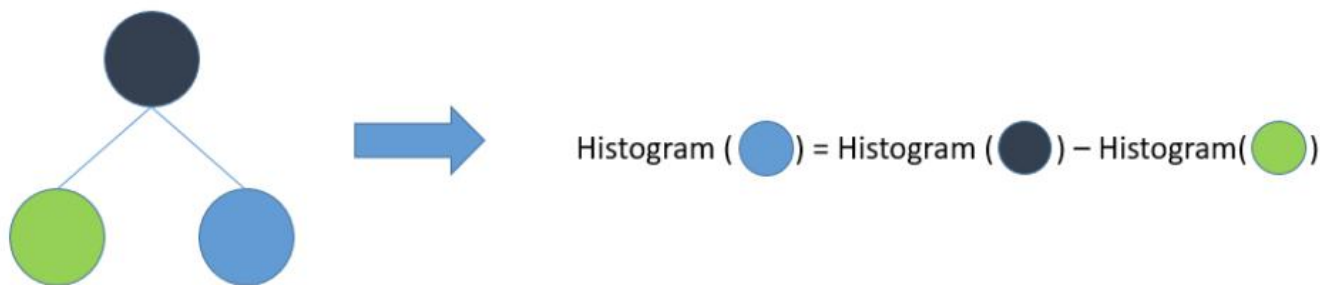
- 减小内存占用，比如离散为256个bin时，只需要8bit，节省7/8
- 减小了split finding时计算增益的计算量，从 $O(\#data)$ 降到 $O(\#bins)$



# LightGBM的改进

- 直方图差加速

一个叶子的直方图可以由它的父亲节点的直方图与它兄弟节点的直方图做差得到，提升一倍速度

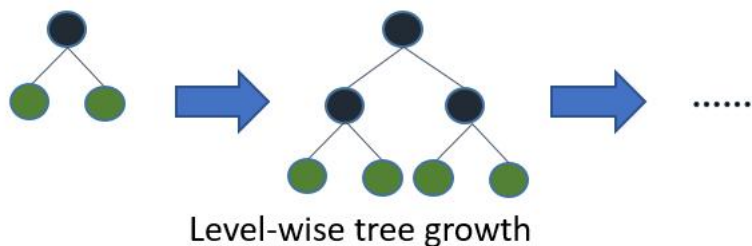


思考：选取哪个子节点统计直方图？



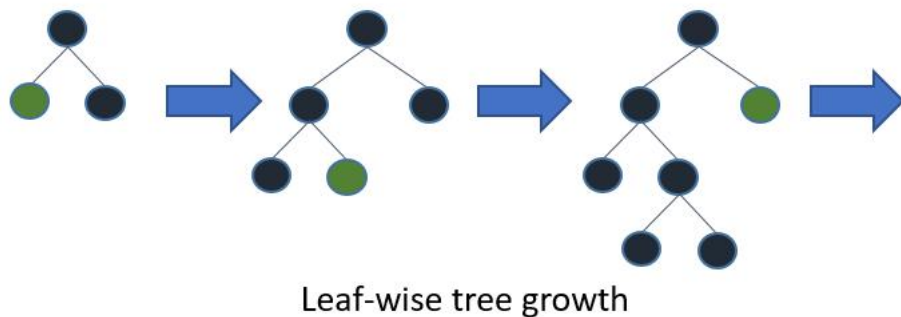
# LightGBM的改进

- 建树过程的两种方法：Level-wise和Leaf-wise



XGBoost

同一层所有节点都做分裂，最后剪枝

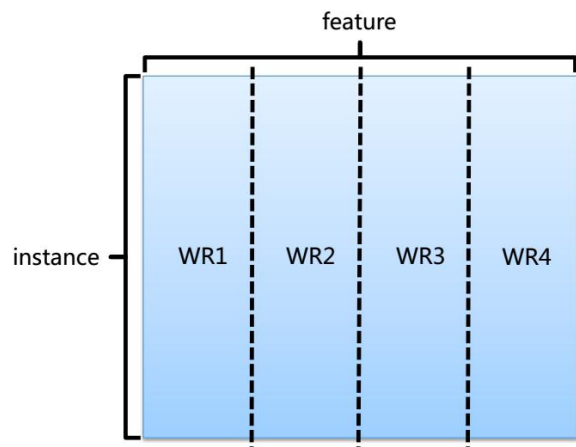


LightGBM

..... 选取具有最大增益的节点分裂  
容易过拟合，通过max\_depth限制

# LightGBM的改进

- 并行优化（**Optimization in parallel learning**）
  - 传统的特征并行
    1. 垂直切分数据，每个worker只有部分特征
    2. 每个worker找到局部最佳切分点（feature, threshold）
    3. worker之间互相通信，找到全局最佳切分点
    4. 具有全局最佳切分特征的worker进行节点分裂，然后广播切分后左右子树的instance indices
    5. 其他worker根据广播的instance indices进行节点分裂



特征并行示意图

缺点：

- split finding计算复杂度 $O(\#data)$ ，当数据量大时会比较慢
- 网络通信代价大，需要广播instance indices

# LightGBM的改进

- 并行优化（**Optimization in parallel learning**）
  - LightGBM的特征并行
    - 每个worker保存所有数据集
    - 1. 每个worker在其特征子集上寻找最佳切分点
    - 2. worker之间互相通信，找到全局最佳切分点
    - 3. 每个worker根据全局最佳切分点进行节点分裂

优点：

避免广播instance indices，减小网络通信量

缺点：

split finding计算复杂度没有减小

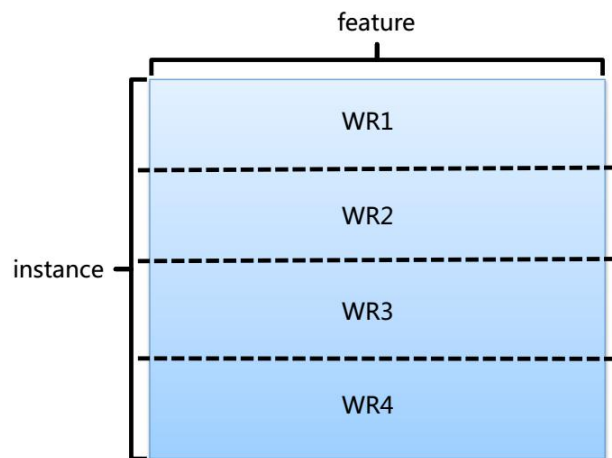
且当数据量比较大时，单个worker存储所有数据代价高

# LightGBM的改进

- 并行优化（**Optimization in parallel learning**）

- 传统的数据并行

1. 水平切分数据，每个worker只有部分数据
2. 每个worker根据本地数据统计局部直方图
3. 合并所有局部直方图得到全局直方图
4. 根据全局直方图进行节点分裂



数据并行示意图

缺点：网络通信代价巨大

采用point-to-point communication algorithm，  
每个worker通信量  $O(\#machine * \#feature * \#bin)$

采用collective communication algorithm，  
每个worker通信量  $O(2 * \#feature * \#bin)$

# LightGBM的改进

- 并行优化（**Optimization in parallel learning**）
  - LightGBM的数据并行
    - 不同的worker合并不同特征的局部直方图
    - 采用直方图做差算法，只需要通信一个节点的直方图

通信量减小到  $O(0.5 * \text{\#feature} * \text{\#bin})$

Voting parallel, 参考论文“A Communication-Efficient Parallel Algorithm for Decision Tree”

# LightGBM的改进

- **Gradient-based One Side Sampling (GOSS)**

在每一次迭代前，利用了GBDT中的样本梯度和误差的关系，对训练样本进行采样：对误差大（梯度绝对值大）的数据保留；对误差小的数据采样一个子集，但给这个子集的数据一个权重，让这个子集可以近似到误差小的数据的全集。这么采样出来的数据，既不损失误差大的样本，又在减少训练数据的同时不改变数据的分布，从而实现了在几乎不影响精度的情况下加速了训练。

- **Exclusive Feature Bundling (EFB)**

在特征维度很大的数据上，特征空间一般是稀疏的。利用这个特征，我们可以无损地降低GBDT算法中需要遍历的特征数量，更确切地说，是降低构造特征直方图（训练GBDT的主要时间消耗）需要遍历的特征数量。

# 参考文献

- Greedy function approximation a gradient boosting machine. *J.H. Friedman(1999)*.
- *Additive logistic regression a statistical view of boosting. Friedman(2000)*.
- XGBoost: A Scalable Tree Boosting System. T. Chen, C. Guestrin (2016).
- A Highly Efficient Gradient Boosting Decision Tree. Guolin Ke (2017).
- Introduction to Boosted Trees. T. Chen
- Tree Boosting With XGBoost. Didrik Nielsen
- 《统计学习方法》李航. 附录梯度下降法与牛顿法
- <https://github.com/Microsoft/LightGBM/wiki/Features>
- [XGBoost 与 Boosted Tree](#)
- [GBDT详解，火光摇曳](#)
- [泰勒公式，维基百科](#)

**谢谢聆听！**