



XGBoost

学习视频



Table of Contents

▼ Table of Contents

[XGBoost基本介绍](#)

[一些重要基本概念](#)

[回归树](#)

[GPT解释](#)

[视频解释](#)

[计算叶子节点权重](#)

[加法模型](#)

[视频解释](#)

[目标函数](#)

[模型表达\(加法算法\) \(前向分布算法\)](#)

[目标函数, 优化目标](#)

[正则项处理](#)

[目标项拆解](#)

[泰勒二阶展开](#)

[确定树的结构](#)

[穷举法](#)

[精确贪心算法](#)

[近似算法](#)

[列采样](#)

[加权分位法](#)

[全局策略](#)

[局部策略](#)

[构建树的方法](#)

[缺失值的处理](#)

[学习率 \(shrinkage\)](#)

[系统设计](#)

[核外块运算](#)

[分块并行](#)

[优化缓存命中率低的问题](#)

▼ XGBoost基本介绍

XGBoost (Extreme Gradient Boosting) 是一种高效且灵活的梯度提升 (Gradient Boosting) 框架，广泛应用于机器学习竞赛和实际应用中。它以其高性能和良好的可扩展性著称。以下是对XGBoost的详细介绍：

1. 基本概念

XGBoost是梯度提升决策树（GBDT）的改进版，通过结合多种优化技术来提升计算速度和模型性能。它的核心思想是通过构建一系列的弱学习器（通常是决策树），逐步改进模型的预测性能。

2. 模型原理

2.1. 梯度提升

梯度提升的基本思想是迭代地添加新模型，使新的模型在已有模型的基础上进一步减少预测误差。具体步骤如下：

- 初始模型通常是一个简单的常数模型。
- 每次迭代训练一个新的决策树，拟合当前模型的残差（即真实值与预测值之间的差异）。
- 将新树的预测结果与之前的模型组合在一起，更新模型的预测结果。

2.2. 目标函数

XGBoost的目标函数包含两个部分：损失函数和正则化项。

$$\text{Obj} = \sum_{i=1}^n L(y_i, \hat{y}_i) + \sum k = 1^K \Omega(f_k)$$

- **损失函数** L ：衡量预测值 \hat{y}_i 与真实值 y_i 之间的误差，常用的损失函数包括均方误差 (MSE) 和对数损失 (Log Loss)。
- **正则化项** Ω ：用于控制模型的复杂度，防止过拟合。常见的正则化项包括树的叶节点数量和叶节点权重的平方和。

2.3. 二阶泰勒展开

为了优化目标函数，XGBoost使用二阶泰勒展开来近似损失函数：

$$\text{Obj} \approx \sum_{i=1}^n [g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2] + \Omega(f_t)$$

其中，
 g_i 和 h_i 分别是损失函数的一阶导数和二阶导数。

3. 模型特性

3.1. 正则化

- **L1正则化**：控制特征选择，使模型稀疏。
- **L2正则化**：防止权重过大，增加模型的鲁棒性。

3.2. Shrinkage

每棵树的预测值都会乘以一个缩减系数（learning rate），从而减慢模型学习速度，提高泛化性能。

3.3. 子样本

每次迭代仅使用部分数据进行训练，增加模型的随机性和抗过拟合能力。

3.4. 列抽样

每棵树仅使用部分特征，类似于随机森林，进一步增加模型的随机性。

4. 实现优化

4.1. 并行计算

XGBoost在计算上进行了多种优化，包括：

- **特征按块并行**：利用多线程加速特征分裂点搜索。
- **缓存感知访问**：优化内存访问模式，提高缓存命中率。
- **稀疏感知算法**：有效处理稀疏数据，支持缺失值处理。

4.2. 分布式计算

支持在分布式环境下运行，能够处理大规模数据集。

5. 应用场景

XGBoost广泛应用于各类监督学习任务，包括分类、回归、排序等。在许多机器学习竞赛（如Kaggle）中，XGBoost是获胜者常用的工具。

6. 示例代码

以下是一个简单的XGBoost应用示例，使用Python编写：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据集
boston = load_boston()
X_train, X_test, y_train, y_test = train_test_split(boston.data, boston.target, test_size=0.2, random_state=42)

# 转换为DMatrix格式
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数
params = {
    'booster': 'gbtree',
    'objective': 'reg:squarederror',
    'eta': 0.1,
    'max_depth': 6,
    'subsample': 0.8,
    'colsample_bytree': 0.8
}

# 训练模型
num_round = 100
bst = xgb.train(params, dtrain, num_round)

# 预测
y_pred = bst.predict(dtest)
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

7. 总结

XGBoost通过多种优化技术提升了梯度提升模型的性能和计算效率，其核心特性包括正则化、Shrinkage、子样本和列抽样等，并支持并行和分布式计算。XGBoost在实际应用中表现优异，适用于各种监督学习任务。如果你有更多具体问题或需要进一步的解释，请告诉我。

好的，这里是将XGBoost的详细介绍转换成适合Notion正确渲染的Markdown格式：

一些重要基本概念

L click

g_i, h_i click

G_i, H_i click

w_i click

Obj_{min} click

Process

d : 特征维度

回归树

▼ GPT解释

回归树（Regression Tree）是一种决策树模型，主要用于回归问题，即预测连续数值型的输出。回归树通过递归地将数据划分成更小的区域，来预测目标变量的值。以下是回归树的详细介绍：

1. 基本概念

- **节点（Node）**：树的每个节点代表一个决策点或一个终点。非叶节点表示决策条件，叶节点表示预测的结果。
- **根节点（Root Node）**：树的起点，包含所有的数据样本。

- **叶节点 (Leaf Node)** : 终端节点，包含预测的目标值。
- **分裂 (Split)** : 在非叶节点上，根据某个特征值的条件，将数据集划分成两个或多个子集。

2. 构建过程

构建回归树的过程可以概括为以下几个步骤：

1. 选择最优分裂点：

- 在每个节点，根据某个特征和特征值，将数据集划分成两个子集。
- 选择能够使得数据集划分后误差（如均方误差MSE）最小的分裂点。

2. 递归分裂：

- 对每个子集，重复上述过程，继续选择最优分裂点，直到达到停止条件（如最大深度、最小样本数、最小误差下降等）。

3. 生成叶节点：

- 当停止分裂时，将当前子集的均值作为叶节点的预测值。

3. 评估与剪枝

- **评估**：使用均方误差 (MSE)、均绝对误差 (MAE) 等指标来评估模型性能。
- **剪枝 (Pruning)**：为了防止过拟合，可以通过剪枝技术对决策树进行简化。常见的剪枝方法有预剪枝和后剪枝。
 - **预剪枝 (Pre-pruning)**：在构建树的过程中，通过设置停止条件提前终止树的生长。
 - **后剪枝 (Post-pruning)**：先生成一棵完整的决策树，然后通过剪枝减少其复杂度。

4. 优点

- **易解释**：回归树的决策过程直观易懂，可以清晰地展示特征对预测结果的影响。
- **处理非线性关系**：回归树能够自动捕捉特征之间的非线性关系。
- **无需特征标准化**：对特征不需要进行标准化或归一化处理。

5. 缺点

- **容易过拟合**：如果不加限制，回归树会对训练数据拟合得过于精细，导致泛化性能差。
- **不稳定**：对训练数据的小变动会导致树结构的较大变化。
- **缺乏平滑性**：回归树的预测值在特征值的变化上会有较大的跳跃，不如线性回归那样平滑。

6. 改进方法

- **集成方法**：通过集成学习的方法，如随机森林（Random Forest）和梯度提升树（Gradient Boosting Trees），可以提升模型的稳定性和预测性能。
 - **随机森林**：通过构建多棵回归树并结合其预测结果，可以减少单一树的过拟合风险。
 - **梯度提升树**：通过迭代地构建回归树，每棵树都在前一棵树的残差上进行拟合，从而提高预测精度。

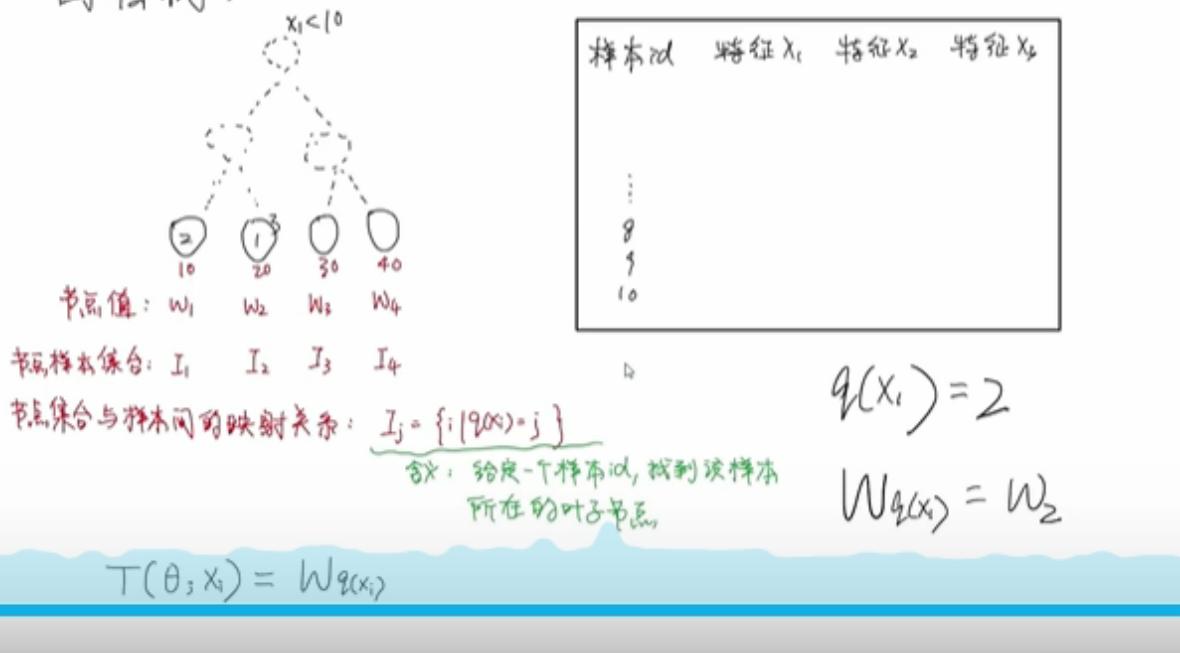
7. 应用

回归树广泛应用于各种需要预测连续值的场景，如房价预测、股票价格预测、气象数据分析等。

希望以上介绍对你理解回归树有所帮助。如果你有具体的疑问或需要详细了解某一方面，请随时告诉我。

▼ 视频解释

回归树：



这张图展示了回归树的一个具体例子，并结合了一些公式和符号解释。以下是对这张图中各个部分的详细讲解：

1. 回归树结构

- **树形结构**：左上角的图展示了一个简单的回归树结构。
 - **根节点**：顶层节点，表示特征 (x_1) 小于 10 的分裂条件。
 - **叶节点**：树的末端节点，其中包含了预测的目标值，如 10、20、30、40 等。
 - **内部节点**：树中非叶子节点，用于进一步分裂数据。

2. 区域划分

- **区域值**：每个叶节点上的值（如 10、20、30、40）表示在该区域内的预测值。这些值通常是该区域内训练数据的均值或中位数。
- **划分依据**：以 $(x_1 < 10)$ 为分裂条件，将数据集划分成不同的子集。

3. 数据集映射

- **数据集分割**：右上角的表格展示了数据集的特征和样本：

- 特征 $\{x_1\}$ 、 $\{x_2\}$ 、 $\{x_3\}$ 等。
- 样本 $\{x_i\}$ ，例如 $\{x_8\}$ 、 $\{x_9\}$ 、 $\{x_{10}\}$ 等。
- 映射函数 $\{g(x_i)\}$ 和 $\{W_{\{g(x_i)\}}\}$ ：图中展示了两个函数：
 - $\{g(x_i)\}$ 表示第 $\{i\}$ 个样本 $\{x_i\}$ 所属的区域索引。在这个例子中， $\{g(x_1) = 2\}$ 表示样本 $\{x_1\}$ 属于第 2 个区域。
 - $\{W_{\{g(x_i)\}}\}$ 表示第 $\{i\}$ 个样本所在区域的预测值，例如 $\{W_{\{g(x_1)\}} = W_2\}$ 表示样本 $\{x_1\}$ 的预测值是区域 2 的值。

4. 公式解释

- 目标函数 $\{T(\theta, x_i) = W_{\{g(x_i)\}}\}$ ：
 - 这表示回归树的预测函数 $\{T\}$ ，对于样本 $\{x_i\}$ ，通过区域索引函数 $\{g(x_i)\}$ 找到对应的区域，然后用区域的预测值 $\{W_{\{g(x_i)\}}\}$ 作为最终预测值。

5. 符号说明

- **I 表示区域索引**： $\{I_j = \{i \mid g(x_i) = j\}\}$ 表示第 $\{j\}$ 个区域包含的样本索引集合。
- **区域划分示例**：图中展示了如何通过特征 $\{x_1\}$ 的值（例如 $\{x_1 < 10\}$ ）将样本分配到不同的区域。

6. 实际应用

- 这种回归树结构和方法常用于机器学习中的回归问题，通过递归划分数据空间，找到适合每个子区域的最佳预测值，从而提高模型的预测精度。

总结来说，这张图详细展示了回归树的基本结构、数据集划分方法、预测函数的定义以及区域映射关系。通过这种方法，可以直观地理解如何利用回归树来进行数值预测。

▼ 计算叶子节点权重

计算XGBoost中叶子节点的权重涉及到最小化目标函数。目标函数包括损失函数和正则化项：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

为了简化计算，假设损失函数是均方误差（MSE）：

1. 首先，将目标函数在叶子节点的部分展开：

$$L(y_i, \hat{y}_i^{(t)}) = \frac{1}{2}(y_i - \hat{y}_i^{(t)})^2$$

假设有一个叶子节点包含样本集合 I_j ，我们对这个叶子节点的权重 w_j 求导并设为零来找到最优权重：

$$\text{Obj}j = \sum i \in I_j \frac{1}{2}(y_i - \hat{y}_i^{(t-1)} - w_j)^2 + \frac{1}{2}\lambda w_j^2$$

1. 对 w_j 求导：

$$\frac{\partial \text{Obj}j}{\partial w_j} = -\sum i \in I_j (y_i - \hat{y}_i^{(t-1)} - w_j) + \lambda w_j$$

1. 设导数为零：

$$-\sum_{i \in I_j} (y_i - \hat{y}_i^{(t-1)}) + \sum i \in I_j w_j + \lambda w_j = 0$$

1. 整理得到最优权重：

$$w_j = \frac{\sum_{i \in I_j} (y_i - \hat{y}_i^{(t-1)})}{\lambda + |I_j|}$$

其中， $|I_j|$ 表示叶子节点包含的样本数量。

总结

最优权重 w_j 是通过对目标函数求导并设为零，解得的一个简单的闭式解。这确保了每个叶子节点的权重都能使得损失函数最小化，同时正则化项控制模型的复杂度。

加法模型

▼ 视频解释

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

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

参数解释 ◎ β_m 为基函数的系数。

$b(x; \gamma_m)$ 为基函数，

γ_m 为基函数的参数。

类比Adaboost的预测函数：可知道Adaboost就是一个加法模型 ◎

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

$$f(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

这张图展示了加法模型（Additive Model）的数学表达形式，并特别指出了它与 Adaboost 算法的关系。以下是对图中各个知识点的详细讲解：

1. 加法模型的基本形式

加法模型的基本形式表示为：

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

其中：

- $f(x)$ 是模型的预测函数。
- M 是基函数的数量。
- β_m 是第 m 个基函数的系数。
- $b(x; \gamma_m)$ 是第 m 个基函数，参数为 γ_m 。

2. 各项参数解释

- **基函数 $b(x; \gamma_m)$:**

- 这是模型中基础的构建单元，可以是任何形式的函数，如线性函数、非线性函数、决策树等。
- 参数 γ_m 是基函数中的特定参数，用于调节基函数的形状或位置。

- 系数 β_m ：
 - 每个基函数前的权重系数，用于调整该基函数对最终预测结果的影响。
 - 系数 β_m 可以通过优化过程来确定，以最小化预测误差。

3. 图中公式解释

图中给出了两种表示形式：

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

$$f(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

这两种表示是等价的：

- 第一种表示：

- β_m 和 $b(x; \gamma_m)$ 是原始形式的加法模型，其中 $b(x; \gamma_m)$ 是基函数，参数为 γ_m
-

- 第二种表示：

- α_m 和 $G_m(x)$ 是另一种形式的加法模型表示，其中 $G_m(x)$ 表示基模型或弱学习器，通常用于机器学习中的集成方法。

4. 与Adaboost的关系

- Adaboost算法：

- Adaboost (Adaptive Boosting) 是一种提升方法，通过组合多个弱学习器来构建一个强学习器。
- Adaboost的预测函数形式与加法模型类似：

$$f(x) = \sum_{m=1}^M \alpha_m G_m(x)$$

- 其中， α_m 是每个弱学习器的权重， $G_m(x)$ 是第 m 个弱学习器。

图中指出：

| 类比Adaboost的预测函数：可知Adaboost就是一个加法模型。

这表明Adaboost实际上是加法模型的一种具体实现形式，其中基函数 $G_m(x)$ 是一系列弱学习器，权重系数 α_m 是通过加权重采样和迭代优化得到的。

5. 总结

图中的内容通过公式展示了加法模型的基本结构，并与Adaboost算法进行了类比，帮助理解加法模型在机器学习中的实际应用。具体知识点包括：

- 加法模型的基本形式和参数解释。
- 基函数和权重系数的作用。
- 加法模型与Adaboost的关系，说明Adaboost是加法模型的一种具体实现，通过组合弱学习器来提升模型性能。

希望以上讲解能帮助你更好地理解图中的内容和加法模型的相关知识。如果有其他问题或需要进一步解释的地方，请随时告诉我。

目标函数

▼ 模型表达(加法算法) (前向分布算法)

模型表达 (加法模型)

$$\hat{y}_i^{(T)} = \sum_{j=1}^T f_j(x_i) = \underbrace{\sum_{j=1}^{T-1} f_j(x_i)}_{\hat{y}_i^{(T-1)}} + f_T(x_i)$$

前向分步算法：

$$\begin{aligned}\hat{y}_i^{(t)} &= \hat{y}_i^{(t-1)} + \underbrace{f_t(x_i)}_{T(\theta; x_i)} \quad f_t(x_i) = W_{\theta(x_i)} \\ &= \hat{y}_i^{(t-1)} + W_{\theta(x_i)}\end{aligned}$$

这张图展示了加法模型 (Additive Model) 在迭代优化中的表示和前向分步算法的细节。以下是对图中各个知识点的详细讲解：

1. 模型表示 (加法模型)

加法模型的表达式可以表示为：

$$\hat{y}_i^{(T)} = \sum_{j=1}^T f_j(x_i) = \sum_{j=1}^{T-1} f_j(x_i) + f_T(x_i)$$

其中：

- $\hat{y}_i^{(T)}$ 是在第 T 次迭代后的预测值。
- $f_j(x_i)$ 是第 j 次迭代中学到的基函数。
- $\hat{y}_i^{(T-1)}$ 表示在前 $T - 1$ 次迭代后的预测值，新增的基函数 $f_T(x_i)$ 是第 T 次迭代中学到的。
- T 代表有 T 颗回归树

2. 前向分步算法

前向分步算法是一种逐步添加基函数来构建模型的方法。它的核心思想是每一步都添加一个新的基函数，使得当前模型的预测误差最小化。其具体步骤如下：

- **初始模型**：从一个简单的初始模型开始，例如所有预测值为常数的模型。

$$\hat{y}_i^{(0)}$$

- **逐步添加基函数**：

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

其中，

$f_t(x_i)$ 是在第 t 步迭代中学到的新基函数。

- **选择新的基函数 $f_t(x_i)$** ：

选择使得损失函数（如平方误差）下降最多的基函数。

3. 详细推导

图中展示了迭代的具体推导过程：

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

$$f_t(x_i) = W q(x_i)$$

其中：

- $W_{q(x_i)}$ 是当前基函数的权重，与数据点 x_i 所属的区域相关。

4. 图示理解

图中还包含一个小的回归树示例，展示了如何在树结构中添加新的分支或节点以优化模型的预测：

- **初始树（1）**：表示初始模型。
- **新增分支（2）**：在第一步迭代后添加一个新分支，形成新的树结构。
- **进一步分裂（3）**：继续优化树结构，增加更多分支。

5. 总结

总结来说，这张图通过公式和图示详细描述了加法模型的构建过程及其在前向分步算法中的应用。其关键点包括：

- **加法模型的逐步构建**：每一步迭代通过添加新的基函数来改进模型。
- **前向分步算法**：逐步添加基函数，每一步优化当前的损失函数。
- **基函数的选择**：选择能使损失最小化的基函数，并结合树结构展示如何具体实现这种优化。

通过这种逐步优化的方法，加法模型能够有效捕捉数据的复杂模式，提高预测精度。如果你有更多具体的问题或需要进一步的解释，请告诉我。

▼ 目标函数，优化目标

目标函数：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

优化目标：

$$(w_1^*, w_2^*, \dots, w_m^*) = \arg \min \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

这张图展示了加法模型在机器学习中的目标函数和优化目标。以下是对图中各个知识点的详细讲解：

1. 目标函数

目标函数的表达式为：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

目标函数中的组成部分：

- **损失函数** $L(y_i, \hat{y}_i^{(t)})$ ：
 - 这是预测值 $\hat{y}_i^{(t)}$ 和真实值 y_i 之间的差异，表示为损失 L 。
 - 常见的损失函数包括均方误差（MSE）、绝对误差（MAE）等。

- 正则化项 $\Omega(f_j)$ ：
 - 正则化项 Ω 用于控制模型的复杂度，防止过拟合。
 - 它通常是基函数 f_j 的复杂度的某种度量，例如树模型的叶节点数或模型参数的范数。

2. 优化目标

优化目标的表达式为：

$$(\omega^1, \omega^2, \dots, \omega^m) = \arg \min \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

优化目标的具体内容：

- **参数集合** $(\omega^1, \omega^2, \dots, \omega^m)$ ：
 - 这些是需要优化的参数，通常是模型的权重或基函数的参数。
- **最小化损失和正则化项的和**：
 - 目标是找到一组参数，使得损失函数和正则化项之和最小。
 - 这涉及到在模型的拟合程度和复杂度之间进行权衡。
 - m 是叶子节点的个数

3. 详细解读

- **损失函数 L :**
 - 在每次迭代中，损失函数度量了模型的预测值与真实值之间的误差。
 - 通过最小化损失函数，可以逐步提高模型的预测精度。
- **正则化项 Ω :**
 - 正则化项防止模型过度复杂，防止过拟合。
 - 通过增加正则化项，可以约束模型的复杂度，使其在训练数据和新数据上都有良好的表现。
- **优化过程 :**
 - 在每次迭代中，更新模型参数 ω ，使目标函数的值逐步减小。
 - 使用梯度下降或其他优化算法，找到使目标函数最小化的参数值。

4. 应用举例

- 梯度提升树（Gradient Boosting Trees）：
 - 在梯度提升树中，损失函数通常是均方误差（MSE），正则化项是树的复杂度。
 - 每次迭代，通过拟合当前残差，逐步提升模型性能。
- 线性回归：
 - 在线性回归中，损失函数是均方误差（MSE），正则化项可以是L1正则化（Lasso）或L2正则化（Ridge）。

5. 总结

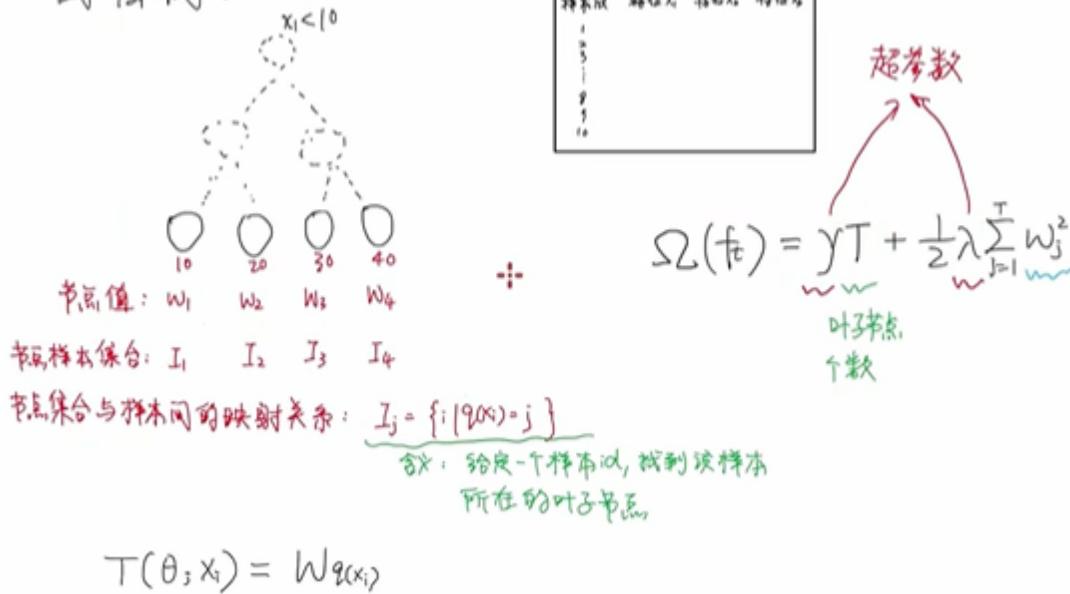
这张图展示了加法模型的核心目标函数和优化过程，通过最小化损失函数和正则化项的和，找到最优的模型参数。关键点包括：

- **损失函数**：度量预测误差。
- **正则化项**：控制模型复杂度，防止过拟合。
- **优化目标**：在每次迭代中，通过更新参数，最小化目标函数。

这种方法在机器学习中被广泛应用于各种模型的训练和优化中。如果你有更多具体的问题或需要进一步的解释，请告诉我。

▼ 正则项处理

回归树：



$$Obj^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \underbrace{\sum_{j=1}^t \Omega(f_j)}_{\text{超参数}}$$

样本id	特征 x_1	预测 \hat{y}_i	特征 x_2
1	1	10	1
2	2	20	2
3	3	30	3
4	4	40	4

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

回归树和XGBoost详细介绍

第一张图：回归树及其正则化项

1. 回归树结构

- **树结构**：展示了一个简单的回归树，其中根节点依据特征 x_1 是否小于 10 进行划分。
 - **根节点**：树的顶端节点，决定数据的第一个划分条件。
 - **叶节点**：树的末端节点，表示不同区域内的预测值（如10、20、30、40）。

- 内部节点：非叶节点，用于进一步划分数据。

2. 数据集映射

- 数据集分割：右上角的表格展示了样本和特征的分布情况。
- 区域索引：展示了如何依据特征 x_1 将数据分配到不同区域，如 I_1, I_2, I_3, I_4 。

3. 模型函数

- 预测函数 $T(\theta, x_i)$ ：
 - 表示对样本 x_i 的预测值，通过模型参数 θ 和样本的特征计算得到。

4. 正则化项

- 正则化项 $\Omega(f_t)$ ：
 - 公式： $\Omega(f_t) = \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$
 - γT ：叶节点的数量，控制树的复杂度。
 - $\frac{1}{2} \lambda \sum_{j=1}^T w_j^2$ ：叶节点权重的平方和，防止权重过大，提高模型的鲁棒性。

第二张图：XGBoost中的目标函数

1. 目标函数

- 目标函数 $\text{Obj}^{(t)}$ ：
 - 公式： $\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$
 - 损失函数 L ：衡量预测值 \hat{y}_i 和真实值 y_i 之间的误差。
 - 正则化项 $\Omega(f_j)$ ：控制模型复杂度，防止过拟合。

2. 损失函数展开

- 二阶泰勒展开：为了优化目标函数，使用二阶泰勒展开近似损失函数：
 - 公式： $\sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) \approx \sum_{i=1}^N [g_i f_t(x_i) + \frac{1}{2} h_i f_t(x_i)^2]$
 - g_i ：损失函数的一阶导数。
 - h_i ：损失函数的二阶导数。

结合两张图的内容：

1. 目标函数和正则化项的结合

- XGBoost通过最小化目标函数（包括损失函数和正则化项）来构建模型，使得模型在训练数据和新数据上的表现都较好。

2. 模型优化过程

- 每次迭代通过增加一个新的基函数 $f_t(x_i)$ 来改进模型。
- 通过控制正则化项中的叶节点数量和叶节点权重的平方和，来防止模型过拟合。

具体步骤总结：

1. **初始模型**：开始时，模型是一个简单的常数模型。
2. **迭代训练**：在每次迭代中，训练一个新的基函数来拟合当前的残差。
3. **更新模型**：将新基函数加入到模型中，更新预测结果。
4. **正则化**：在每次迭代中，计算正则化项，控制模型复杂度。
5. **优化**：使用二阶泰勒展开来近似损失函数，进行优化。

通过以上步骤，XGBoost在保证高性能的同时，控制了模型的复杂度，防止过拟合，从而提高了模型的泛化能力。如果你有任何具体问题或需要进一步解释的地方，请随时告诉我。

你可以将上述内容复制到Notion中进行查看和使用。如果有更多具体问题或需要进一步的解释，请告诉我。

▼ 目标项拆解

目标函数：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

优化目标：

$$(w_1^*, w_2^*, \dots, w_T^*) = \arg \min \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

A 文本模式 插入空格 | ▲ ▼ ▶ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷ ▷

特征 x_1 特征 x_2 y $\hat{y}=w$ $L(y, \hat{y})$

x_1	x_2	y	$\hat{y}=w$	$L(y, \hat{y})$
y_1		w_3	L_1	$L_1 = (y_1 - w_3)^2$
y_2			w_3	L_2
:				$L_2 = (y_2 - w_3)^2$
y_5		w_1	w_1	L_5
				$L_5 = (y_5 - w_1)^2$

$L_3 = (y_3 - w_3)^2$

$L_4 = (y_4 - w_4)^2$

$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$

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

结点集合： I_1, I_2, I_3, I_4

结点集合与样本间的映射关系： $I_j = \{i | g(x_i) = j\}$

含义：给定一个样本

3x: 给定
所在

$$L_{\text{结点}} = L_3 + L_5$$

$$(y_3 - w_1)^2 + (y_5 - w_1)^2 \rightarrow R$$

$$= y_3^2 - 2y_3w_1 + w_1^2 + y_5^2 - 2y_5w_1 + w_1^2$$

$$= (y_3^2 + y_5^2) - 2(y_3 + y_5)w_1 + w_1^2$$

$$w = -\frac{b}{2a} = y_3 + y_5$$

目标函数:

$$Obj^{(t)} = \underbrace{\sum_{i=1}^N L(y_i, \hat{y}_i^{(t)})}_{YT} + YT + \frac{1}{2}\lambda \sum_{j=1}^J w_j^2$$

$$YT + \sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \underbrace{\hat{y}_i^{(t)}}_{\hat{y}_i^{(t-1)} + w_j}) \right] + \frac{1}{2}\lambda w_j^2$$

目标函数:

$$Obj^{(t)} = YT + \sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \underbrace{\hat{y}_i^{(t-1)} + w_j}_{\hat{y}_i^{(t)}}) \right] + \frac{1}{2}\lambda w_j^2$$

1. 目标函数

图片1和图片5展示了XGBoost的目标函数：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \sum_{j=1}^t \Omega(f_j)$$

其中：

- L 是损失函数，用于衡量预测值与真实值之间的误差。
- $\Omega(f_j)$ 是正则项，用于控制模型复杂度，防止过拟合。

2. 回归树的工作流程

图片2介绍了回归树的工作流程：

1. 给定数据集，包括样本 (x_i, y_i) 。
2. 构建树结构，每个叶子节点对应一个预测值 w_j 。
3. 计算每个样本的损失，累加得到总损失。

3. 叶子节点的损失计算

图片3详细说明了叶子节点的损失计算过程。对于每个叶子节点的损失，我们需要最小化平方误差损失：

$$L_{\text{叶节点1}} = (y_3 - w_1)^2 + (y_5 - w_1)^2$$

通过对损失函数求导并设为零，可以得到最优的叶子节点预测值 w_1 ：

$$w_1 = \frac{y_3 + y_5}{2}$$

4. 优化目标

图片1和图片4展示了如何重新定义目标函数以适应叶子节点的遍历：

$$\text{Obj}^{(t)} = \sum_{i=1}^N L(y_i, \hat{y}_i^{(t)}) + \gamma T + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

其中：

- γT 是叶子节点数量的惩罚项。
- $\lambda \sum_{j=1}^T w_j^2$ 是叶子节点权重的L2正则化项。

5. 样本集合与叶子节点的映射

图片2中的表格和树结构图显示了样本如何被划分到不同的叶子节点中，并计算对应的损失。

6. 目标函数解释

图片中的目标函数是：

$$\text{Obj}^{(t)} = \gamma T + \sum_{j=1}^T \left[\left(\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j) \right) + \frac{1}{2} \lambda \sum_{j=1}^T w_j^2 \right]$$

这个等式将之前的大目标 $\sum_{i=1}^N L(y_i, \hat{y}_i^{(t)})$ 拆分成了小目标 $\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j)$ ，并且每个小目标就只有一个变量 w_i 让我们逐步解析这个目标函数：

1. 正则化项

$$\gamma T$$

- 这里的 γ 是一个常数，它用于控制叶子节点的数量 T 。这个项起到了正则化的作用，防止模型过于复杂。

2. 损失函数

$$\sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j) \right]$$

- 这个部分表示的是损失函数的累加。
- $L(y_i, \hat{y}_i^{(t-1)} + w_j)$ 表示第 i 个样本的损失，其中 $\hat{y}_i^{(t-1)}$ 是前一轮迭代的预测值， w_j 是当前树中第 j 个叶子节点的权重。
- I_j 表示第 j 个叶子节点（可以说是叶节点）包含的样本索引集合（第 j 颗叶子节点）。

3. 正则化项（叶子节点权重）

$$\frac{1}{2} \lambda \sum_{j=1}^T w_j^2$$

- 这个部分是对叶子节点权重的正则化项，其中 λ 是正则化参数， w_j 是第 j 个叶子节点的权重。
- 该项的引入是为了防止权重过大，起到平滑模型的作用。

总结

- 正则化项 γT 控制叶子节点数量，防止模型过于复杂。
- 损失函数累加项 $\sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j) \right]$ 表示所有叶子节点上的样本损失之和。
- 权重正则化项 $\frac{1}{2} \lambda \sum_{j=1}^T w_j^2$ 控制叶子节点权重的大小，防止过拟合。

这一目标函数的设置帮助XGBoost在训练过程中通过优化叶子节点的分布和权重，使得整体模型的预测误差和复杂度都能得到有效控制。

总结

XGBoost模型通过构建多个加法模型（树），逐步减小损失函数，以提高预测精度。它通过引入正则项控制模型复杂度，并采用贪心算法优化每一步的目标函数。理解这些图片中的公式和流程，可以帮助我们更好地理解XGBoost的工作原理和应用。

▼ 泰勒二阶展开

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

$\underbrace{f(x_0)}$ 定值 $\underbrace{f'(x_0)(x - x_0)}$ 定值 $\underbrace{\frac{1}{2} f''(x_0)(x - x_0)^2}$ 定值

$$L(y_i, \hat{y}_i^{(t-1)} + w_j) \approx g_i w_j + \frac{1}{2} h_i w_j^2$$

$\underbrace{x_0}_{x}$ $\underbrace{x - x_0}_{\hat{y}_i^{(t-1)} + w_j}$

$L'(y_i, \hat{y}_i^{(t-1)})$
常量 g_i

$L''(y_i, \hat{y}_i^{(t-1)})$
常量 h_i

$$j^{(t)} = \gamma T + \sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j) \right] + \frac{1}{2} \lambda w_j^2$$

GBDT - 37
XGBoost = 87

$$\approx \gamma T + \sum_{j=1}^T \left(w_j \left(\sum_{i \in I_j} g_i \right) + \frac{1}{2} w_j^2 \left(\lambda + \sum_{i \in I_j} h_i \right) \right)$$

G_j H_j

$$Obj^{(t)} = \gamma T + \sum_{j=1}^T \left[\sum_{i \in I_j} L(y_i, \hat{y}_i^{(t-1)} + w_j) \right] + \frac{1}{2} \lambda w_j^2$$

GBDT - 37
XGBoost = 87

$$\approx \gamma T + \sum_{j=1}^T \left(w_j G_j + \frac{1}{2} w_j^2 (\lambda + H_j) \right)$$

$L'(y_i, \hat{y}_i^{(t)})$ $L''(y_i, \hat{y}_i^{(t)})$
 g_i h_i

$\sum_{i \in I_j} g_i$
 G_j

$\sum_{i \in I_j} h_i$
 H_j

下面是对你提供的图片中内容的详细讲解。

泰勒展开与目标函数近似

在XGBoost中，目标是最小化目标函数。为了便于计算，对目标函数进行二阶泰勒展开：

1. 二阶泰勒展开

函数 $f(x)$ 在 x_0 处的二阶泰勒展开式为：

$$f(x) \approx f(x_0) + f'(x_0)(x - x_0) + \frac{1}{2} f''(x_0)(x - x_0)^2$$

对应到我们的损失函数 $L(y_i, \hat{y}_i^{(t)} + w_j)$ ，记

$$g_i = L'(y_i, \hat{y}_i^{(t)})$$

和

$$h_i = L''(y_i, \hat{y}_i^{(t)})$$

▼ 具体解释

计算梯度 g_i 和Hessian h_i

在XGBoost中，梯度和Hessian是通过对损失函数的导数进行计算得到的。具体计算方法如下：

梯度 g_i

梯度 g_i 是损失函数 L 对预测值 \hat{y}_i 的一阶导数：

$$g_i = \frac{\partial L(y_i, \hat{y}_i)}{\partial \hat{y}_i}$$

Hessian h_i

Hessian h_i 是损失函数 L 对预测值 \hat{y}_i 的二阶导数：

$$h_i = \frac{\partial^2 L(y_i, \hat{y}_i)}{\partial \hat{y}_i^2}$$

具体步骤

选择损失函数：常见的损失函数包括均方误差（MSE）和逻辑回归的对数损失函数。

计算梯度和Hessian：

- 对于均方误差（MSE）：

- 损失函数：

$$L(y_i, \hat{y}_i) = \frac{1}{2}(y_i - \hat{y}_i)^2$$

- 梯度：

$$g_i = \hat{y}_i - y_i$$

- Hessian：

$$h_i = 1$$

- 对于对数损失函数（逻辑回归）：

- 损失函数：

$$L(y_i, \hat{y}_i) = y_i \log(\sigma(\hat{y}_i)) + (1 - y_i) \log(1 - \sigma(\hat{y}_i))$$

- 梯度：

$$g_i = \sigma(\hat{y}_i) - y_i, \text{ 其中 } \sigma(\hat{y}_i) = \frac{1}{1+e^{-\hat{y}_i}}$$

- Hessian：

$$h_i = \sigma(\hat{y}_i)(1 - \sigma(\hat{y}_i))$$

示例

假设我们有以下数据：

- 样本 y_i : [0, 1, 0, 1]
- 预测值 \hat{y}_i : [0.1, 0.6, 0.2, 0.8]

对于均方误差 (MSE)，计算梯度和Hessian如下：

- 梯度：

$$g_1 = 0.1 - 0 = 0.1$$

$$g_2 = 0.6 - 1 = -0.4$$

$$g_3 = 0.2 - 0 = 0.2$$

$$g_4 = 0.8 - 1 = -0.2$$

- Hessian：对于MSE，每个样本的Hessian都为1：

$$h_1 = 1, h_2 = 1, h_3 = 1, h_4 = 1$$

对于逻辑回归，计算梯度和Hessian如下：

- 梯度：

$$g_1 = \sigma(0.1) - 0 = 0.525 - 0 = 0.525$$

$$g_2 = \sigma(0.6) - 1 = 0.645 - 1 = -0.355$$

$$g_3 = \sigma(0.2) - 0 = 0.55 - 0 = 0.55$$

$$g_4 = \sigma(0.8) - 1 = 0.689 - 1 = -0.311$$

- Hessian :

$$h_1 = \sigma(0.1)(1 - \sigma(0.1)) = 0.525 \times (1 - 0.525) = 0.249$$

$$h_2 = \sigma(0.6)(1 - \sigma(0.6)) = 0.645 \times (1 - 0.645) = 0.229$$

$$h_3 = \sigma(0.2)(1 - \sigma(0.2)) = 0.55 \times (1 - 0.55) = 0.248$$

$$h_4 = \sigma(0.8)(1 - \sigma(0.8)) = 0.689 \times (1 - 0.689) = 0.214$$

通过以上方法，可以在Notion中正确渲染数学公式。

通过计算每个样本的梯度和Hessian，我们可以用来进行树的构建和优化。

我们有：

$$L(y_i, \hat{y}_i^{(t)} + w_j) \approx g_i w_j + \frac{1}{2} h_i w_j^2$$

2. 目标函数的近似

目标函数的近似形式：

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

记：

$$G_j = \sum_{i \in I_j} g_i$$

和

$$H_j = \sum_{i \in I_j} h_i$$

▼ 详细解释

在XGBoost中， $\langle G_j \rangle$ 和 $\langle H_j \rangle$ 是由损失函数的一阶导数和二阶导数计算得到的，它们在优化模型时起着关键作用。

详细解释 G_j 和 H_j :

G_j - 一阶导数（梯度）

- G_j 是损失函数对所有属于第 j 个叶子节点的实例的一阶导数（梯度）的总和。
- 它衡量了损失函数的最陡上升方向和速率，表明预测变化对模型改进或恶化的影响。

数学表示：

$$G_j = \sum_{i \in I_j} g_i$$

其中，

$$g_i = \frac{\partial L(y_i, \hat{y}_i^{(t)})}{\partial \hat{y}_i^{(t)}}$$

H_j - 二阶导数 (Hessian)

- H_j 是损失函数对所有属于第 j 个叶子节点的实例的二阶导数 (Hessian) 的总和。
- 它衡量了损失函数的曲率，表明梯度方向的可信度。 H_j 值越高，表明对梯度方向的确定性越高。

数学表示：

$$H_j = \sum_{i \in I_j} h_i$$

其中，

$$h_i = \frac{\partial^2 L(y_i, \hat{y}_i^{(t)})}{\partial (\hat{y}_i^{(t)})^2}$$

在目标函数中的作用

- 这些导数帮助使用二阶泰勒级数展开来近似损失函数的变化，从而简化复杂的优化问题。
- 考虑这些成分的目标函数可以写成：

$$\text{Obj}^{(t)} \approx \gamma T + \sum_{j=1}^T [w_j G_j + \frac{1}{2} w_j^2 (\lambda + H_j)]$$

优化

- 通过对目标函数关于 w_j 求导并设为零，可以求得最优叶子节点权重：

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

- 更新后的目标函数为：

$$\text{Obj}^{(t)} \approx \gamma T - \frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{\lambda + H_j}$$

这个过程确保模型找到叶子节点的最优权重，从而提高预测性能，同时控制模型的复杂度。

目标函数可以简化为：

$$\text{Obj}^{(t)} \approx \gamma T + \sum_{j=1}^T [w_j G_j + \frac{1}{2} w_j^2 (\lambda + H_j)]$$

其中 T 是叶子节点数

3. 求解最优权重

为了最小化目标函数，对每个叶子节点的权重 w_j 求导(将 Obj_t 对 w_1 求导)并设为零：

$$\frac{\partial \text{Obj}^{(t)}}{\partial w_j} = G_j + w_j(\lambda + H_j) = 0$$

解得最优权重：

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

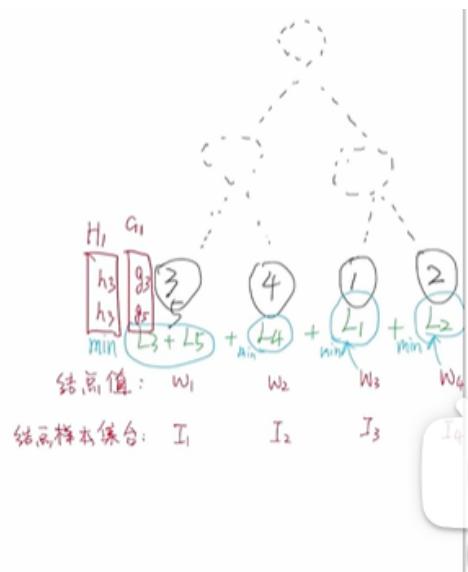
4. 更新后的目标函数

将最优权重代入目标函数，得到更新后的目标函数值：

$$\text{Obj}^{(t)} \approx \gamma T - \frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{\lambda + H_j}$$

样本id	特征 x_1	特征 x_2	y	$\hat{y} = w$	$L(y, \hat{y})$	g_i	h_i
1			y_1	w_3	L_1	g_1	h_1
2			y_2	\vdots	L_2	\vdots	
3			\vdots	\vdots	$L_3 = (y_3 - w)^2$	\vdots	
4			\vdots	\vdots			
5			y_5	w_1	$L_5 = (y_5 - w_1)^2$	g_5	h_5

$$\left\{ \begin{array}{l} w_j^* = -\frac{b}{2\alpha} = -\frac{G_j}{H_j + \lambda} \\ \text{Obj}^{(t)*} = \min \text{Obj}^{(t)} = \gamma T - \frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{H_j + \lambda} \end{array} \right.$$



拿到数据集，先计算出 g_i, h_i ，再算出 G_i, H_i ，再算出最优权重 w_i ，最终算出最小损失 Obj_{min}

总结

通过二阶泰勒展开，我们将复杂的目标函数简化为关于叶子节点权重的二次函数，并通过求导得到最优的叶子节点权重。这样的方法使得我们能够高效地优化XGBoost模型的结构和参数。

这张图片详细说明了这一过程，包括损失函数的泰勒展开、目标函数的近似和权重的优化步骤。理解这些步骤是掌握XGBoost优化过程的关键。

确定树的结构

▼ 穷举法

好的，以下是对逐字稿内容的详细解释，重点在于如何通过选择最佳的划分条件来优化树结构。

1. 确定树结构的重要性

在XGBoost中，我们已经学会了如何通过最小化目标函数来确定叶子节点的权重 w_j 和相应的损失 Obj 。然而，这些计算是基于固定的树结构进行的。实际应用中，不同的划分条件会导致不同的树结构，从而影响叶子节点中的样本分布，最终影响梯度 G_j 和Hessian H_j ，进而改变 w_j 和 Obj 。

2. 穷举法

一种直接但不实用的方法是穷举法，即计算所有可能的树结构的 Obj ，然后选择最小的 Obj 对应的树作为最佳树。然而，这种方法计算量巨大，尤其是在特征数量多和取值范围大的情况下。

3. 贪心算法

贪心算法是一种更为高效的方法，它在每一步选择当前最优的划分条件，而不是考虑所有可能的划分组合。这大大减少了计算量。

4. 预排序

通过对特征进行预排序，可以在寻找最佳划分点时快速定位，从而提高计算效率。

5. 剪枝

剪枝技术允许提前停止不必要的计算。例如，当某个划分条件不能显著减少损失时，可以提前放弃该划分。

数学公式推导

1. 目标函数近似：

$$text{Obj}^{(t)} \approx \gamma T + \sum_{j=1}^T [w_j G_j + \frac{1}{2} w_j^2 (\lambda + H_j)]$$

2. 求解最优权重：

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

3. 更新后的目标函数：

$$\text{Obj}^{(t)} \approx \gamma T - \frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{\lambda + H_j}$$

总结

通过上述优化策略，XGBoost能够高效地找到最优的树结构，避免了计算量庞大的穷举法，从而在保证模型性能的同时，显著提高了训练速度和效率。这些优化策略使得XGBoost成为一个强大且高效的机器学习工具。

▼ 精确贪心算法

▼ GPT explanation

精确贪心算法在XGBoost中的应用

XGBoost中的精确贪心算法旨在高效地找到每一步的最佳划分点，以最小化目标函数。这一算法通过逐步计算每个特征的所有可能划分，并选择使目标函数下降最多的划分点。

具体步骤

1. 初始化

- 计算每个特征的所有可能取值，并将数据按该特征的值排序。
- 初始化梯度(g_i)和Hessian(h_i)的累积和。

2. 计算每个划分点的损失

- 对每个特征的每个可能划分点，计算左右子节点的梯度和Hessian累积和。
- 使用以下公式计算损失：

对于左子节点：

$$\text{Obj}_{\text{left}} = \frac{(\sum g_{\text{left}})^2}{\sum h_{\text{left}} + \lambda}$$

对于右子节点：

$$\text{Obj}_{\text{right}} = \frac{(\sum g_{\text{right}})^2}{\sum h_{\text{right}} + \lambda}$$

总损失：

$$\text{Obj} = \text{Obj}_{\text{left}} + \text{Obj}_{\text{right}}$$

3. 选择最佳划分点

- 比较每个特征的每个可能划分点的损失，选择使总损失最小的划分点作为最佳划分。

公式解释

梯度和Hessian

- 梯度 G 和 Hessian H 的计算如下：

$$G = \sum_{i=1}^N g_i$$

$$H = \sum_{i=1}^N h_i$$

- 目标函数包含正则化项，以防止过拟合：

$$\text{Obj}^{(t)} \approx \gamma T + \sum_{j=1}^T [w_j G_j + \frac{1}{2} w_j^2 (\lambda + H_j)]$$

- 最优权重 w_j 的求解：

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

- 更新后的目标函数值：

$$\text{Obj}^{(t)} \approx \gamma T - \frac{1}{2} \sum_{j=1}^T \frac{G_j^2}{\lambda + H_j}$$

优点

- **高效性**：通过对特征进行预排序，可以快速计算每个划分点的损失。
- **精确性**：每一步都选择当前最优的划分，保证局部最优解。

例子

假设我们有一个特征 x 和相应的标签 y ，我们首先计算该特征的所有可能划分点。

假设 x 有以下值：

$$x = [1, 2, 3, 4, 5]$$

对应的梯度 g 和 Hessian h 为：

$$g_i = [0.1, -0.2, 0.3, -0.4, 0.5]$$

$$h_i = [1, 1, 1, 1, 1]$$

我们逐步计算每个可能划分点的损失，并选择损失最小的划分点。例如：

- 划分点为 2 时：

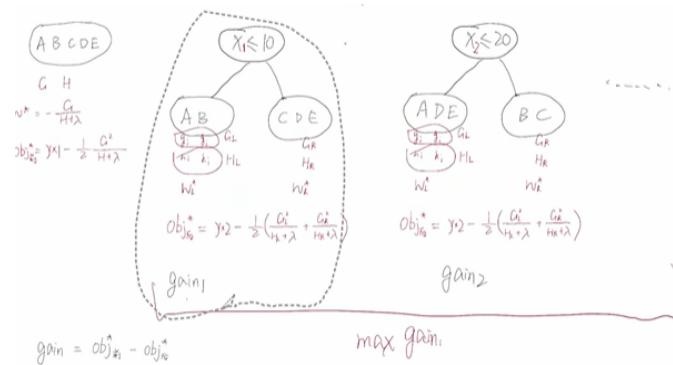
- 左子节点： $G_{\text{left}} = 0.1 - 0.2 = -0.1$, $H_{\text{left}} = 2$
- 右子节点： $G_{\text{right}} = 0.3 - 0.4 + 0.5 = 0.4$, $H_{\text{right}} = 3$
- 总损失：

$$\text{Obj} = \frac{(-0.1)^2}{2+\lambda} + \frac{(0.4)^2}{3+\lambda}$$

通过这种方式计算所有可能的划分点，选择损失最小的作为最佳划分点。

通过上述步骤和公式，我们可以高效地使用精确贪心算法来优化XGBoost模型的树结构，从而提高模型的预测性能。

▼ 视频方法



$$\begin{aligned}
 \text{gain} &= \text{obj}_{j_1}^* - \text{obj}_{j_2}^* \\
 &= \frac{1}{2} \left[\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{(G_L + G_R)^2}{H_L + H_R + \lambda} \right] - y
 \end{aligned}$$

1. $\text{min gain} \leq 10^{-5}$

2. 叶子节点包含样本个数 ≤ 1

3. 层级、叶子节点数 ...

1. 精确贪心算法的基本思想

在XGBoost的学习过程中，树的生成可以看作是一个节点逐步分裂的过程。每一步的目标是选择一个最佳划分，使得目标函数的损失最小。精确贪心算法通过计算每个可能的划分点的增益，选择增益最大的划分点。

2. 增益计算

增益 (gain) 的计算公式如下：

$$\text{gain} = \text{Obj}_{\text{split}(\text{前})} - \text{Obj}_{\text{no split}(\text{后})}$$

具体计算过程如下：

分裂前的目标函数：

$$\text{Obj}_{\text{no split}} = \frac{G^2}{H + \lambda}$$

分裂后的目标函数：

$$\text{Obj}_{\text{split}} = \frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda}$$

增益公式：

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

其中：

- G 和 H 分别是梯度和Hessian的累加和。
- G_L 和 G_R 分别是左子节点和右子节点的梯度累加和。
- H_L 和 H_R 分别是左子节点和右子节点的Hessian累加和。
- λ 是正则化参数。
- γ 是叶子节点的惩罚项。

3. 节点的分裂过程

1. 计算当前节点的梯度和Hessian累加和。
2. 遍历每个可能的划分点，计算左右子节点的梯度和Hessian累加和。
3. 计算每个划分点的增益，选择增益最大的划分点作为当前节点的最优划分。

4. 树的生成和停止条件

- 树的生成是一个递归的过程，每次选择增益最大的划分点进行分裂。
- 停止条件可以包括：
 1. 最大增益小于某个阈值。
 2. 叶子节点包含的样本数少于某个阈值。
 3. 树的最大深度达到预设的阈值。

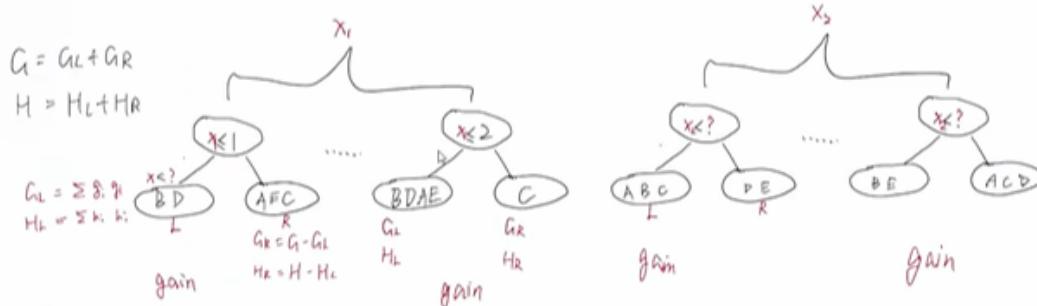
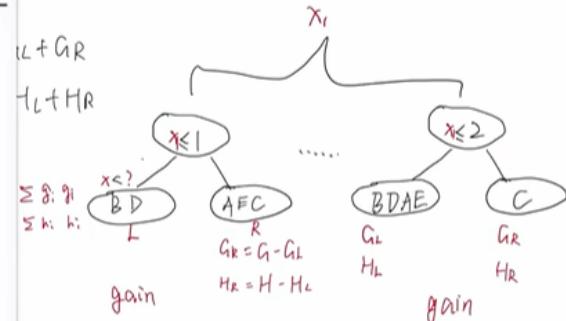
▼ 算法实现

Algorithm 1: Exact Greedy Algorithm for Split Finding

```

Input:  $I$ , instance set of current node
Input:  $d$ , feature dimension
 $gain \leftarrow 0$  特征维度为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  $x_{jk}$ ) 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

```



特征 X_1	样本ID	g_i	h_i	G_L	G_R	H_L	H_R	Gain
1	B							
2	A							
3	C							

特征 X_2	样本ID	g_i	h_i	G_L	G_R	H_L	H_R	Gain
10	E							
20	B							
30	C							

精准贪心算法概述

精准贪心算法的目标是在每个节点上找到最佳的分裂点，使得目标函数的增益最大化。这一过程是通过对所有可能的分裂点进行评估并选择使得增益最大的那个点来实现的。

算法步骤

输入

- I ：当前节点的实例集合
- d ：特征维度

▼ 特征维度

在XGBoost中，特征维度（feature dimension）指的是数据集中用于训练模型的特征（属性、变量）的数量。在机器学习中，特征是用来描述和表示样本的一组可测量的属性。每个样本（数据点）可以有多个特征，这些特征共同定义了样本的特征空间。

特征维度的重要性

- 1. 模型性能**：特征的数量和质量对模型的性能有直接影响。更多的特征可以提供更多的信息，有助于模型更好地理解数据，但也可能导致维度灾难（curse of dimensionality），即在高维空间中数据变得稀疏，模型难以泛化。
- 2. 训练时间**：特征维度的增加会导致模型训练时间的增加，因为更多的特征意味着更多的计算。
- 3. 复杂度**：模型的复杂度随着特征维度的增加而增加，过多的特征可能导致模型过拟合（overfitting）。

特征维度的选择与处理

- 1. 特征选择**：通过选择最相关的特征，可以减少特征维度，提高模型的性能和训练速度。常用的方法有过滤法（Filter）、包裹法（Wrapper）、嵌入法（Embedded）。
- 2. 特征工程**：包括特征提取和特征变换。可以通过特征组合、特征缩放、特征编码等方法来创建更有意义的特征。
- 3. 降维**：通过PCA（主成分分析）等方法将高维数据降到低维空间，减少特征维度，同时尽量保留数据的主要信息。

在XGBoost中的应用

在XGBoost中，特征维度的处理主要体现在分裂节点时对每个特征进行评估和选择最佳的分裂点。以下是一些具体步骤：

- 1. 遍历特征**：对于每个特征，计算所有可能的分裂点。
- 2. 评估分裂点**：计算每个分裂点的增益，增益的公式为：

$$gain = G_L^2 / (H_L + \lambda) + G_R^2 / (H_R + \lambda) - G^2 / (H + \lambda)$$

其中， G 和 H 分别是梯度和二阶梯度的总和， G_L 和 H_L 是左子节点的梯度和二阶梯度的总和， G_R 和 H_R 是右子节点的梯度和二阶梯度的总和， λ

是正则化参数。

3. **选择最优特征和分裂点**：选择增益最大的特征及其对应的分裂点进行节点分裂。

代码示例

以下是一个简单的示例，展示如何在XGBoost中处理特征维度：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(
    X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)
```

```

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

```

在这个示例中，我们使用波士顿房价数据集进行训练。特征维度为13，因为数据集中有13个特征。在训练过程中，XGBoost会自动评估每个特征及其可能的分裂点，选择增益最大的特征进行节点分裂，从而构建决策树。

希望这些信息能帮助你更好地理解XGBoost中的特征维度。如果有任何进一步的问题或需要更详细的解释，请随时告诉我。

输出

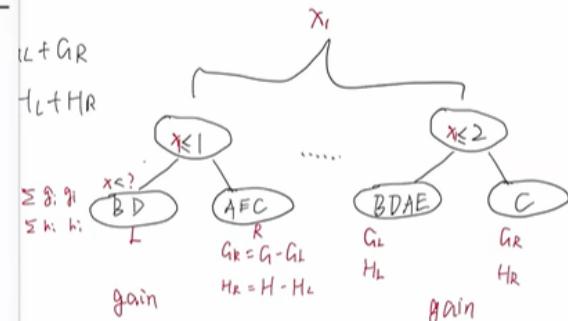
- 使得增益最大的分裂点

伪代码

```

Algorithm 1: Exact Greedy Algorithm for Split Finding
Input:  $I$ , instance set of current node
Input:  $d$ , feature dimension
 $gain \leftarrow 0$  // 增益操作没有写错了
 $G \leftarrow \sum_{i \in I} g_i, H \leftarrow \sum_{i \in I} h_i$ 
for  $k = 1$  to  $m$  do
     $G_L \leftarrow 0, H_L \leftarrow 0$ 
    for  $j$  in  $sorted(I, by x_{jk})$  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

```



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 d do #这个是遍历每个特征，也就是每个 x_i)

$G_L \leftarrow 0, H_L \leftarrow 0$

for j in sorted(I , by $x_{j,k}$) do #这是遍历各个特征的多种可能，求最大收益

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

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

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

end

end

Output: Split with max score

具体步骤解析

1. **初始化**：增益gain初始化为0。计算当前节点所有实例的梯度和G和二阶梯度和H
2. **遍历特征**：对每一个特征进行处理：
 - 初始化左子节点的梯度和 G_L 和二阶梯度和 H_L 为0
 - 将实例按该特征的值进行排序
 - 遍历排序后的实例，逐个将实例从当前节点移动到左子节点，更新 G_L 和 H_L
 - 计算右子节点的梯度和 G_R 和二阶梯度和 H_R
 - 计算当前分裂点的增益，并更新最大增益score

计算过程示例

假设我们有一个特征 x 和若干个实例，实例的梯度和二阶梯度分别为 g_i 和 h_i ，我们需要对特征 x 进行分裂：

1. 初始化：

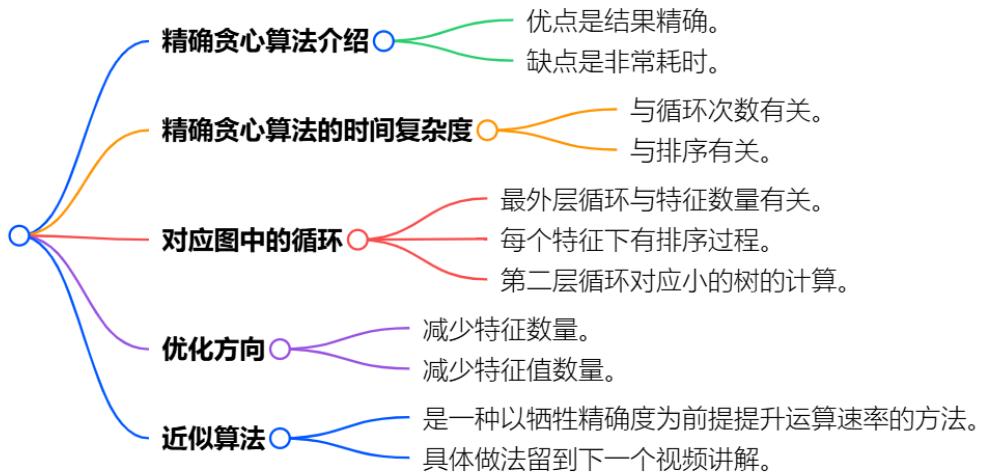
- 计算节点上所有实例的梯度和 $G = \sum g_i$ 和二阶梯度和 $H = \sum h_i$

2. 遍历特征：

- 对每个可能的分裂点，计算左子节点和右子节点的梯度和二阶梯度：
 - G_L 和 H_L 分别是左子节点的梯度和二阶梯度的总和
 - $G_R = G - G_L$ 和 $H_R = H - H_L$ 分别是右子节点的梯度和二阶梯度的总和
- 计算增益： $gain = G_L^2/(H_L + \lambda) + G_R^2/(H_R + \lambda) - G^2/(H + \lambda)$
- 选择增益最大的分裂点

这个算法通过遍历所有特征的所有可能分裂点，计算每个分裂点的增益，最终选择增益最大的分裂点作为最优分裂点。

▼ 优化算法

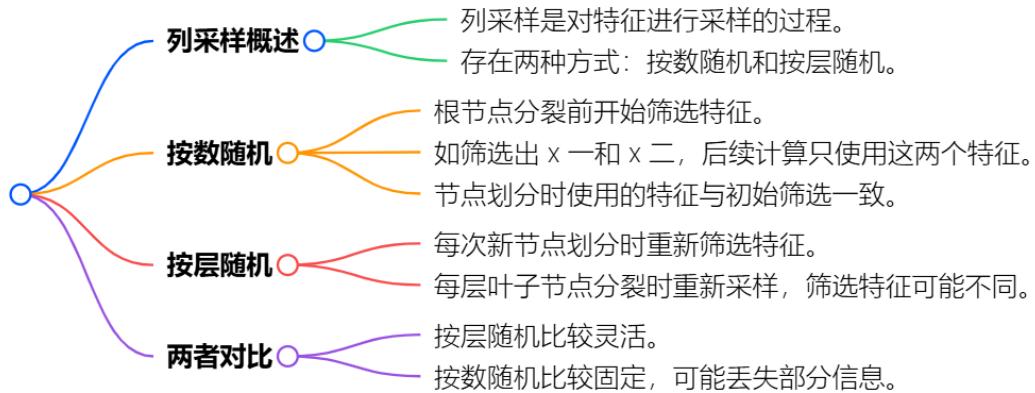


豆包

你的 AI 助手，助力每日工作学习

近似算法

▼ 列采样



豆包

你的 AI 助手，助力每日工作学习

列采样有按数随机和按层随机两种方式。按数随机在根节点分裂前筛选特征，后续节点沿用；按层随机每层节点分裂时重新筛选。按层随机灵活，按数随机固定，后者可能丢失信息。

近似算法中的列采样

列采样是XGBoost中的一个重要技术，用于提升模型的泛化能力和计算效率。列采样涉及对特征进行随机选择，而不是使用全部特征进行训练和分裂。这样可以减少计算量，并且增加模型的多样性，防止过拟合。

列采样概述

列采样是对特征进行采样的过程。它存在两种方式：按数随机和按层随机。

按数随机

- 在节点分裂前开始筛选特征。
- 假如选出特征 x_1 和 x_2 ，后续计算只使用这两个特征。
- 节点划分时使用的特征与初始筛选一致。

按层随机

- 每次新节点划分时重新筛选特征。
- 每层叶子节点分裂时重新采样，筛选特征可能不同。

两者对比

- 按层随机比较灵活。
- 按数随机比较固定，可能丢失部分信息。

近似算法中的列采样

在XGBoost中，列采样与近似算法结合使用，通过随机选择特征的子集进行分裂点的近似计算。以下是详细步骤：

1. **初始化**：根据给定的列采样比例，随机选择一部分特征进行使用。例如，如果列采样比例设置为0.8，则每次只使用80%的特征。
2. **特征排序**：对选中的特征进行排序，准备进行分裂点的查找。
3. **候选分裂点**：基于排序后的特征，生成候选分裂点的集合。
4. **增益计算**：对每个候选分裂点，计算其增益。增益的计算公式为：

$$gain = G_L^2 / (H_L + \lambda) + G_R^2 / (H_R + \lambda) - G^2 / (H + \lambda)$$

其中， G 和 H 分别是梯度和二阶梯度的总和， G_L 和 H_L 是左子节点的梯度和二阶梯度的总和， G_R 和 H_R 是右子节点的梯度和二阶梯度的总和， λ 是正则化参数。

5. **选择最佳分裂点**：选择增益最大的分裂点进行节点分裂。

代码示例

以下是一个简单的代码示例，展示如何在XGBoost中设置列采样参数：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
```

```

X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数，包括列采样
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'colsample_bytree': 0.8, # 列采样比例
    'colsample_bylevel': 0.8 # 每层的列采样比例
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

```

在这个示例中，`colsample_bytree` 和 `colsample_bylevel` 参数用于控制列采样的比例。`colsample_bytree` 表示在构建每棵树时使用的特征比例，`colsample_bylevel` 表示在每层使用的特征比例。通过设置这些参数，XGBoost可以在训练过程中随机选择部分特征，从而提高模型的多样性和泛化能力。

希望这些信息能帮助你更好地理解XGBoost中的列采样。如果有进一步的问题或需要更详细的解释，请随时告诉我。

▼ 加权分位法

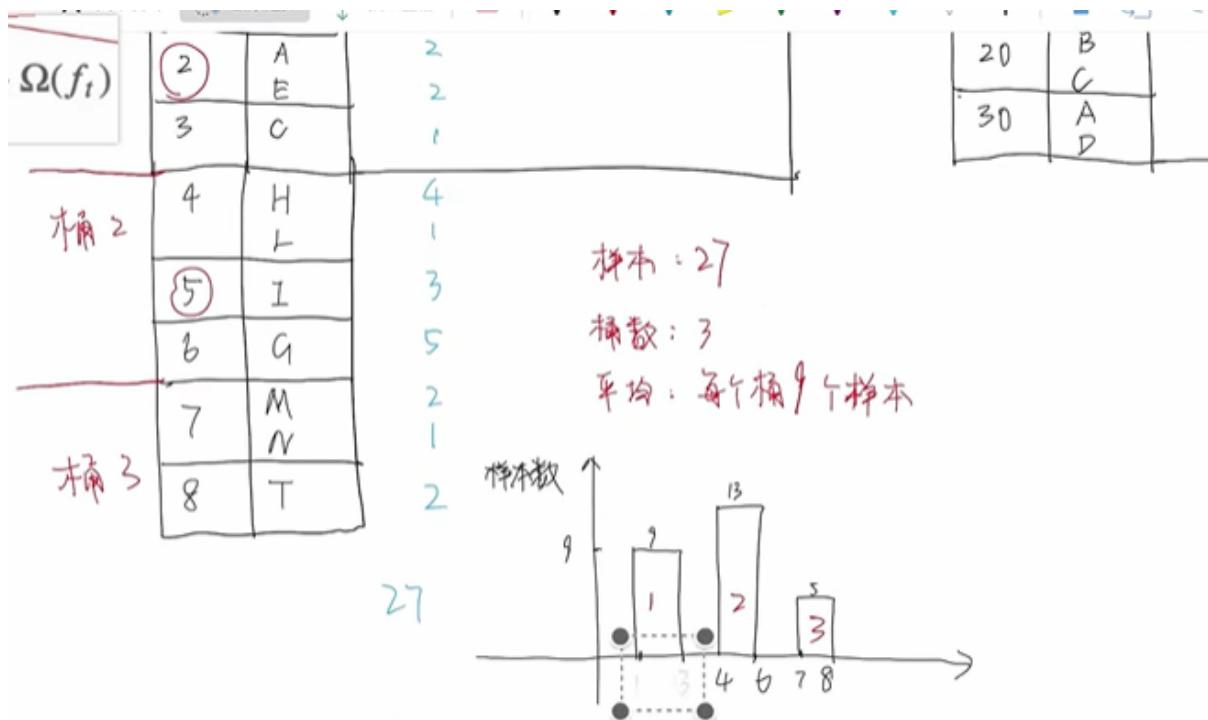


$$\begin{aligned}
 & \sum_{i=1}^N \left(g_i f_t(\mathbf{x}_i) + \frac{1}{2} h_i f_t^2(\mathbf{x}_i) \right) + \Omega(f_t) \\
 &= \sum_{i=1}^N \frac{1}{2} h_i \left(2 \frac{g_i}{h_i} f_t(\mathbf{x}_i) + f_t^2(\mathbf{x}_i) \right) + \Omega(f_t) \\
 &= \sum_{i=1}^N \frac{1}{2} h_i \left(\frac{g_i^2}{h_i^2} + 2 \frac{g_i}{h_i} f_t(\mathbf{x}_i) + f_t^2(\mathbf{x}_i) \right) + \Omega(f_t) \\
 &= \sum_{i=1}^N \frac{1}{2} h_i \left(f_t(\mathbf{x}_i) - \left(-\frac{g_i}{h_i} \right)^2 \right)^2 + \Omega(f_t)
 \end{aligned}$$

常量，不影响目标函数的优



从上式中可看出，目标函数是真实值为 $-g_i/h_i$ ，权重为 h_i 的二阶梯度加权损失，因此，使用二阶梯度加权。



切分点 $\{s_{k1}, s_{k2}, \dots, s_{kl}\}$ 应满足:

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \varepsilon, \quad s_{k1} = \min_i x_{ik}, \quad s_{kl} = \max_i x_{ik}$$

$$r_k(z) = \frac{1}{\sum_{(x,h) \in D_k} h} \sum_{(x,h) \in D_k, x < z} h$$

加权分位法 (Weighted Quantile Sketch) 介绍

加权分位法是一种用于处理特征值分布不均匀的算法，通过对数据进行加权分桶，计算每个桶内的加权分位数来确定分裂点。这一方法可以减少计算量并保证分裂点选择的准确性。

目标函数推导

XGBoost 使用二阶泰勒展开的目标函数来近似损失函数，从而进行优化。目标函数包含了损失函数和正则项：

$$L(\theta) = \sum_{i=1}^N l(y_i, f_t(x_i)) + \Omega(f_t)$$

其中，

- $l(y_i, f_t(x_i))$ 是样本 i 的损失
- $\Omega(f_t)$ 是正则项，用于控制模型复杂度

二阶泰勒展开

我们对目标函数进行二阶泰勒展开，可以得到：

$$L(\theta) \approx \sum_{i=1}^N (g_i f_t(x_i) + 1/2 h_i f_t^2(x_i)) + \Omega(f_t)$$

其中，

- $g_i = \partial l(y_i, f(x_i))/\partial f(x_i)$ 是一阶梯度
- $h_i = \partial^2 l(y_i, f(x_i))/\partial^2 f(x_i)$ 是二阶梯度

分桶与计算过程

分桶步骤

1. **初始化**：根据给定的分桶数量和加权分位法，初始化分桶参数。
2. **特征排序**：对每个特征值进行排序。
3. **加权分桶**：根据样本的权重 h_i ，将特征值分配到不同的桶中。（假如一个值的 h_i 为3，那么等价于他出现了三次。分桶等分时要算三次）

公式推导

通过对上述公式进行简化和变形，我们可以得到优化目标的计算公式。以下是详细推导过程：

1. 目标函数展开：

$$\sum_{i=1}^N (g_i f_t(x_i) + 1/2 h_i f_t^2(x_i)) + \Omega(f_t)$$

2. 对样本进行加权(同时提取出 $0.5h_i$)：

$$= \sum_{i=1}^N (1/2 h_i (2g_i/h_i f_t(x_i) + f_t^2(x_i)) + \Omega(f_t))$$

3. 重新组合公式(加上一个常量 $2g_i/h_i f_t(x_i)$ ，不影响计算)：

$$= \sum_{i=1}^N (1/2 h_i (g_i^2/h_i^2 + 2g_i/h_i f_t(x_i) + f_t^2(x_i)) + \Omega(f_t))$$

4. 最终化简公式：

$$= \sum_{i=1}^N (1/2h_i(f_t(x_i) - (-g_i/h_i))^2) + \Omega(f_t)$$

这表明，通过加权分位数计算，可以用 h_i 作为样本的权重来进行分裂点的选择。

加权分位法的分裂点选择

在实际操作中，XGBoost 使用加权分位法将数据进行分桶，然后通过计算每个桶的增益来选择最优的分裂点。增益的计算公式为：

$$gain = G_L^2/(H_L + \lambda) + G_R^2/(H_R + \lambda) - G^2/(H + \lambda)$$

其中，

- G_L 和 H_L 是左子节点的梯度和二阶梯度的总和
- G_R 和 H_R 是右子节点的梯度和二阶梯度的总和
- G 和 H 是当前节点的梯度和二阶梯度的总和
- λ 是正则化参数

代码示例

以下是一个简单的代码示例，展示如何在XGBoost中应用加权分位法：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
```

```

dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数，包括加权分位法的参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'max_bin': 256, # 分桶数
    'subsample': 0.8, # 样本采样比例
    'colsample_bytree': 0.8, # 特征采样比例
    'tree_method': 'approx' # 使用近似算法
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

```

通过这个过程，我们可以看到如何使用加权分位法来高效地进行特征分裂点的选择，提高模型的计算效率和准确性。

加权分位法中的切分点和超参数

加权分位法中的切分点和超参数在优化和控制分裂点选择中起着关键作用。以下是详细的解释，包括公式推导和超参数的定义及其作用。

切分点选择

切分点公式

切分点 $\{s_{k1}, s_{k2}, \dots, s_{kl}\}$ 应满足：

$$|r_k(s_{k,j}) - r_k(s_{k,j+1})| < \varepsilon, \quad s_{k1} = \min_i x_{ik}, \quad s_{kl} = \max_i x_{ik}$$

$$r_k(z) = \frac{1}{\sum_{(x,h) \in D_k} h} \sum_{(x,h) \in D_k, x < z} h$$

切分点 $s_{k1}, s_{k2}, \dots, s_{kl}$ 应满足：

$$|r_k(s_{kj}) - r_k(s_{kj} + 1)| < \varepsilon,$$

其中，

- $s_{k1} = \min_i x_{ik}$,
- $s_{kl} = \max_i x_{ik}$,

$$r_k(z) = 1 / (\sum_{(x,h) \in D_k} h) \sum_{(x,h) \in D_k, x < z} h$$

这个公式描述了加权分位法中切分点的计算方法。 $r_k(z)$ 表示权重的累积分布函数，通过计算样本权重的累积分布来确定切分点的位置。 ε 是一个阈值，用于控制切分点之间的间隔。

超参数

超参数定义及作用

在加权分位法中，有几个关键的超参数影响分裂点的选择和模型的性能：

1. **max_bin**: 最大分桶数。这个参数控制特征值被分成的桶的数量，默认值为256。分桶数量越多，分裂点选择越精细，但计算复杂度也越高。
2. **subsample**: 样本采样比例。这个参数控制在每次树构建过程中使用的数据比例，取值范围在0到1之间。较小的值可以防止过拟合，但可能导致欠拟合。
3. **colsample_bytree**: 列采样比例。这个参数控制每次树构建过程中使用的特征比例，取值范围在0到1之间。较小的值可以增加模型的多样性，防止过拟合。

4. **colsample_bytree**：每层列采样比例。这个参数控制在每一层的节点分裂过程中使用的特征比例，取值范围在0到1之间。

5. **lambda (λ)**：正则化参数。这个参数用于控制模型复杂度，防止过拟合。值越大，模型的正则化程度越高。

示例

以下是一个示例，展示如何在XGBoost中设置和使用这些超参数：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数，包括加权分位法的参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'max_bin': 256,  # 分桶数
    'subsample': 0.8,  # 样本采样比例
    'colsample_bytree': 0.8,  # 特征采样比例
    'colsample_bylevel': 0.8,  # 每层的列采样比例
```

```

'lambda': 1, # 正则化参数
'tree_method': 'approx' # 使用近似算法
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

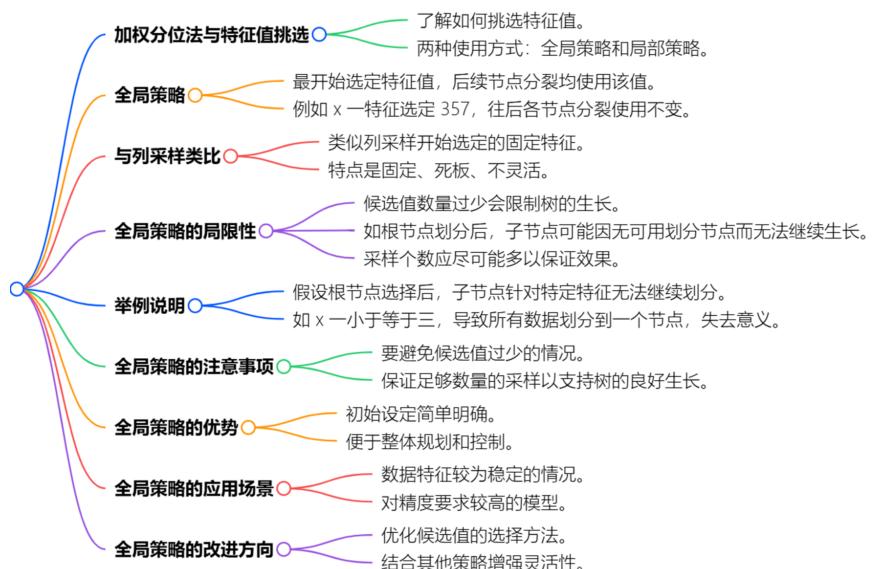
# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')

```

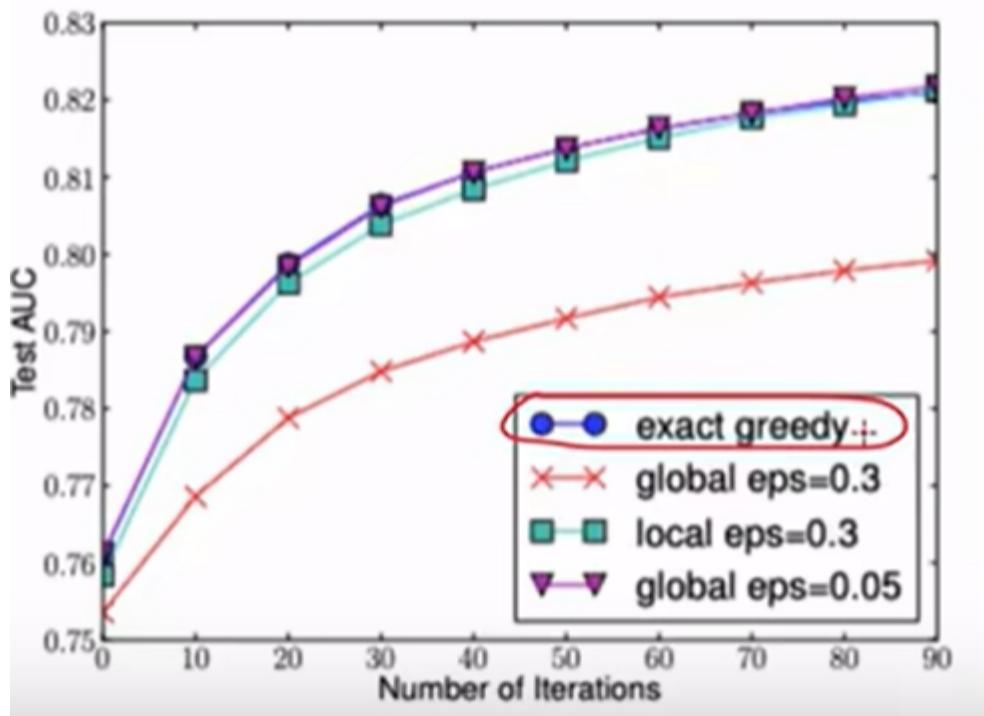
总结

加权分位法通过对数据进行加权分桶，计算每个桶内的加权分位数来确定分裂点，从而提高模型的计算效率和准确性。超参数在这一过程中起着重要作用，控制分裂点选择的精度和模型的复杂度。

▼ 全局策略



▼ 局部策略

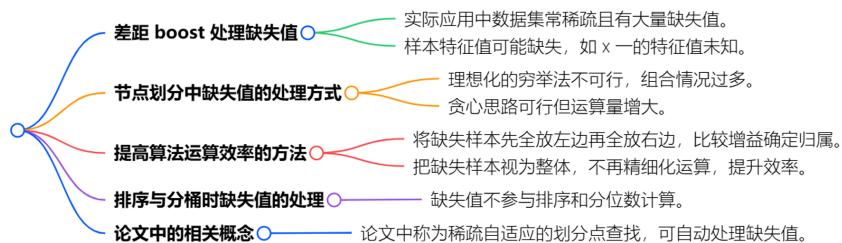
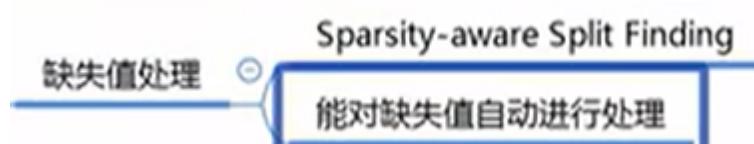


构建树的方法

精确贪心算法

近似算法

▼ 缺失值的处理



这段文字主要讲述了XGBoost 对缺失值的处理方法。实际应用中的数据集常很稀疏存在大量缺失值。最初提到穷举所有情况来划分样本到叶子节点的方法因组合情况多不可行。也可挨个用贪心思路，但运算量大。为提高算法效率，作者将缺失样本整体先放左边或右边，比较所得增益，大的一边就是样本最终位置。在排序和分桶执行分位数算法时，缺失值不参与排序和计算。这种处理方法在论文中叫稀疏自适应的划分点查找，能自动处理缺失值。

XGBoost中缺失值的处理

在实际应用中，数据集往往包含大量缺失值。XGBoost在处理缺失值时采用了一种名为"稀疏自适应的切分点查找"（Sparsity-aware Split Finding）的算法，能够自动处理缺失值。以下是详细介绍和相关方法。

稀疏自适应的切分点查找

缺失值处理概述

缺失值在特征选择和节点分裂时可能会带来计算复杂性和不准确性。XGBoost通过以下方式处理缺失值：

1. **缺失值自动处理**：能够对缺失值进行自动处理，而无需进行额外的填补或删除操作。

差距 Boost 处理缺失值

实际应用中的数据集

在实际应用中，数据集常常稀疏且有大量缺失值。样本特征值可能缺失，例如特征 x_1 的特征值未知。

理想化与实际情况

- 理想化的穷举法不可行，组合情况过多。
- 贪心思路可行但运算量增大。

节点划分中缺失值的处理方式

理想化的处理方式

在节点划分过程中，理想情况下应该考虑所有可能的缺失值组合，但这在实际操作中不现实。

XGBoost的处理方式

将缺失样本先全部左边再放右边，比较增益确定归属。缺失样本不参与排序和分位数计算，而是被视为一个整体。

提高算法运行效率的方法

样本视为整体

把缺失样本视为整体，不再精细化运算，从而提高效率。

不精细化运算

不再将缺失值进行排序和分位数计算，从而减少计算量。

排序与分桶时缺失值的处理

排序与分桶时的处理方法

缺失值不参与排序和分位数计算，只是简单地分配到一边。

自动处理缺失值

XGBoost在处理缺失值时，可以自动将其分配到左右子节点中，并通过增益来确定最佳分配方式。

具体公式

公式解释

在处理缺失值时，XGBoost采用了以下公式来计算分裂点：

$$r_k(z) = 1 / (\sum_{(x,h) \in D_k} h) \sum_{(x,h) \in D_k, x < z} h$$

其中，

- $r_k(z)$ 表示加权累积分布函数。
- D_k 是节点 k 中的所有样本。
- h 是样本的权重。

切分点选择

切分点 $s_{k1}, s_{k2}, \dots, s_{kl}$ 应满足：

$$|r_k(s_{kj}) - r_k(s_{kj+1})| < \varepsilon,$$

其中，

- $s_{k1} = \min_i x_{ik}$
- $s_{kl} = \max_i x_{ik}$

代码示例

以下是一个简单的代码示例，展示如何在XGBoost中应用缺失值处理：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error
import numpy as np

# 加载数据
data = load_boston()
X = data.data
y = data.target
```

```
# 人为引入缺失值
np.random.seed(42)
missing_rate = 0.1
n_samples, n_features = X.shape
n_missing_samples = int(np.floor(missing_rate * n_samples * n_features))
missing_samples = np.array([np.random.choice(n_samples, n_missing_samples),
                           np.random.choice(n_features, n_missing_samples)])
X[missing_samples[0], missing_samples[1]] = np.nan

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象，XGBoost自动处理缺失值
dtrain = xgb.DMatrix(X_train, label=y_train, missing=np.nan)
dtest = xgb.DMatrix(X_test, label=y_test, missing=np.nan)

# 设置参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'tree_method': 'hist', # 使用稀疏自适应算法
    'missing': np.nan # 处理缺失值
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)
```

```
# 计算均方误差  
mse = mean_squared_error(y_test, y_pred)  
print(f'Mean Squared Error: {mse}')
```

总结

通过稀疏自适应的切分点查找和自动处理缺失值的方法，XGBoost可以高效且准确地处理缺失值问题。这种方法不仅简化了预处理步骤，还提高了模型的鲁棒性和计算效率。

▼ 学习率 (shrinkage)

在XGBoost中也加入了步长 η ，也叫做收缩率shrinkage：

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

这有助于防止过拟合，步长 η 通常取为0.1.





XGBoost中的学习率 (Shrinkage)

学习率概述

学习率 (Shrinkage) 是每个基学习器加上的步长。其形式类似于常见的学习率，用于控制每个基学习器的影响程度，从而防止模型过拟合。

公式

在XGBoost中，学习率的公式如下：

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

其中，

- $\hat{y}_i^{(t)}$ 是样本 i 在第 t 轮的预测值
- $\hat{y}_i^{(t-1)}$ 是样本 i 在第 $t - 1$ 轮的预测值
- η 是学习率
- $f_t(x_i)$ 是第 t 个基学习器对样本 i 的预测值

学习率的作用和效果

学习率有助于防止过拟合，通过缩小每个基学习器的步长，使模型逐步逼近最优解。常用的学习率为 0.1。

添加学习率的原因

- **防止过拟合**：当模型训练效果过好时，过于精确容易导致过拟合。
- **模型的稳定性**：通过缩小步长，逐步逼近最优解，提高模型的稳定性和可靠性。

学习率的应用示例

假如通过差距 Boost 模型拟合数据，训练第一个基学习器效果佳时，乘以 0.1 进行收缩。训练第二个基学习器效果好时，同样收缩以防止过拟合，依此类推。不断收缩，达到防止过拟合的最终效果。

学习率的重要性

- **避免过拟合**：对避免模型过拟合具有关键作用。
- **提升模型泛化能力**：有助于提升模型的泛化能力，使其在未知数据上的表现更为稳健。

学习率的原理分析

- **基于模型拟合程度的调整**：根据模型在训练数据上的拟合程度，调整学习率，逐步逼近最优解。
- **以逐步收缩的方式优化模型**：通过逐步缩小步长，使模型稳步优化。

学习率的优势

- **有效降低过拟合风险**：通过缩小每个基学习器的步长，防止模型过于复杂。
- **提高模型稳定性和可靠性**：使模型在训练过程中更加稳定，预测结果更可靠。

学习率与其他方法比较

- **区别于其他防止过拟合的策略**：学习率是通过控制基学习器的影响程度来防止过拟合。
- **独特的应用场景和效果**：在许多实际应用中，学习率是调整模型效果的重要超参数。

学习率的未来发展

- **可能会有进一步的优化和改进**：未来可能会有更智能的学习率调整策略。
- **在更多领域得到应用和拓展**：学习率作为控制模型训练的重要参数，可能会在更多领域中得到应用。

代码示例

以下是一个代码示例，展示如何在XGBoost中设置和使用学习率：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数，包括学习率
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1, # 学习率
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'tree_method': 'approx'
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
```

```
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

总结

学习率（Shrinkage）是XGBoost中重要的超参数，通过控制每个基学习器的步长，防止模型过拟合，提高模型的稳定性和可靠性。在实际应用中，常用的学习率为0.1，可以根据需要进行调整。

希望这些解释和示例能帮助你更好地理解XGBoost中的学习率。如果有进一步的问题或需要更详细的解释，请随时告诉我。

系统设计

▼ 核外块运算

核外块运算（Out-of-core Computation）介绍

在处理大规模数据集时，内存（RAM）可能不足以一次性加载所有数据到内存中。XGBoost为了应对这种情况，引入了核外块运算（Out-of-core Computation）功能，使得在有限内存的情况下也能高效处理大规模数据。以下是详细介绍。

核外块运算概述

核外块运算是一种处理数据的方式，允许在数据量超过内存容量时，依然能够进行有效的计算。它通过将数据分成多个块，每次只加载部分数据到内存中进行处理，从而避免内存溢出。

核外块运算的工作原理

- 数据分块**：将大数据集分成多个小块，每个块可以单独加载到内存中进行处理。
- 块的迭代处理**：依次将每个数据块加载到内存中进行计算，然后将结果保存到磁盘或临时存储中。
- 结果合并**：所有数据块处理完毕后，将各个块的结果合并，得到最终模型。

核外块运算的步骤

1. 数据准备：

- 将数据集分割成多个较小的数据块，这些数据块可以单独加载到内存中进行处理。

2. 块迭代处理：

- 加载一个数据块到内存中。
- 对数据块进行处理和计算，例如计算梯度和二阶梯度。
- 将计算结果保存到磁盘或临时存储中。

3. 模型更新：

- 加载下一个数据块并重复上述步骤。
- 所有数据块处理完毕后，进行模型更新。

4. 结果合并：

- 将所有数据块的计算结果进行合并，得到最终的模型。

核外块运算的优势

1. **处理大规模数据**：可以处理超过内存容量的大规模数据集。
2. **资源利用率高**：通过分块处理，充分利用内存和磁盘资源。
3. **高效性**：尽管需要多次读写磁盘，但通过优化的I/O操作，依然可以保持较高的计算效率。

核外块运算的应用场景

1. **超大数据集**：例如，数十GB或TB级别的数据集，内存无法一次性加载。
2. **内存受限环境**：例如，云计算平台或嵌入式系统，内存资源有限。

XGBoost中的核外块运算实现

在XGBoost中，可以通过设置 `xgb.DMatrix` 的 `data` 参数为磁盘文件路径来实现核外块运算。例如：

```
import xgboost as xgb

# 假设 data.csv 是一个非常大的数据集，无法一次性加载到内存中
dtrain = xgb.DMatrix('data.csv?format=csv&label_column=0')

# 设置参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'tree_method': 'hist' # 使用 histogram-based tree method
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
dtest = xgb.DMatrix('test.csv?format=csv&label_column=0')
y_pred = bst.predict(dtest)

# 保存模型
bst.save_model('model.bst')
```

在上述代码中，`data.csv` 和 `test.csv` 是存储在磁盘上的大规模数据集，通过 `xgb.DMatrix` 加载时，XGBoost 会自动处理数据分块和核外运算。

总结

核外块运算是XGBoost处理大规模数据集的重要特性，使得在内存有限的情况下依然能够进行高效的模型训练和预测。通过分块处理和结果合并，XGBoost能够充分利用内存和磁盘资源，解决了大规模数据处理中的内存瓶颈问题。

▼ 分块并行



分块并行（Block Parallelization）介绍

在处理大规模数据集和进行复杂计算时，并行计算是提高效率和缩短运行时间的重要手段。XGBoost 通过分块并行（Block Parallelization）技术，将数据集划分为多个块，并对这些块进行并行处理，从而提升计算性能。

分块并行的工作原理

分块并行的基本思想是将数据集划分为多个较小的块，每个块独立进行处理，多个块可以同时在多个处理器上并行计算。最终，将各个块的计算结果进行合并，得到最终的模型。

分块并行的步骤

- 数据分块**：将数据集划分为多个块，每个块可以独立处理。
- 并行处理**：将每个块分配到不同的处理器或计算节点上进行并行计算。
- 结果合并**：将各个块的计算结果合并，得到最终的模型。

分块并行的实现

在 XGBoost 中，分块并行主要应用于以下几个方面：

- 梯度计算**：在树的构建过程中，需要计算每个节点的梯度和二阶梯度，XGBoost 将数据集划分为多个块，并行计算每个块的梯度和二阶梯度。

2. **分裂点查找**：在找到最佳分裂点时，XGBoost对数据进行排序和扫描，这些操作也可以通过分块并行来加速。
3. **模型训练**：整个模型训练过程，包括树的构建和更新，可以通过分块并行进行加速。

实现分块并行的技术

1. **多线程并行**：利用多线程技术，在单个计算节点上实现分块并行。XGBoost会自动根据硬件配置，合理分配线程数。
2. **分布式并行**：在多个计算节点上实现分块并行，通过分布式计算框架（如Spark）进行数据分发和并行计算。

分块并行的优势

1. **高效利用计算资源**：通过并行计算，能够充分利用多核处理器和分布式计算资源，提高计算效率。
2. **加速模型训练**：大幅缩短模型训练时间，使得在处理大规模数据集时，能够更快地得到结果。
3. **提升模型性能**：通过分块并行，可以更快地进行超参数调优和模型选择，从而提升模型的整体性能。

分块并行的应用场景

1. **大规模数据集**：在处理数百万甚至数十亿条记录的数据集时，分块并行能够显著加速计算过程。
2. **高维数据**：在处理具有数千或数万个特征的高维数据时，分块并行能够提高梯度和分裂点计算的效率。
3. **分布式计算环境**：在云计算平台或高性能计算集群上，分块并行能够充分利用分布式计算资源，实现高效计算。

代码示例

以下是一个示例，展示如何在XGBoost中使用分块并行进行模型训练：

```
import xgboost as xgb
from sklearn.datasets import load_boston
```

```
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)

# 设置参数，包括并行相关参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'tree_method': 'hist', # 使用 histogram-based tree method 支持分块并行
    'nthread': 4 # 设置使用的线程数，调整为你的机器的核数
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

在上述代码中，`tree_method` 参数设置为 `hist`，这是一种支持分块并行的树构建方法。`nthread` 参数设置为 4，表示使用4个线程进行并行计算，可以根据机器的核数进行调整。

总结

分块并行是XGBoost中提升计算效率的重要技术，通过将数据集划分为多个块并行处理，可以显著加速模型训练过程，并充分利用计算资源。这种技术特别适用于大规模数据集和高维数据处理，在实际应用中具有广泛的应用前景。

希望这些解释和示例能帮助你更好地理解XGBoost中的分块并行技术。如果有进一步的问题或需要更详细的解释，请随时告诉我。

▼ 优化缓存命中率低的问题

XGBoost中优化缓存命中率低的问题

在大规模数据处理中，缓存命中率（Cache Hit Rate）是影响计算效率的重要因素。缓存命中率低会导致频繁的内存访问，增加延迟并降低整体性能。XGBoost通过多种策略来优化缓存命中率，从而提升计算效率。

缓存命中率的概述

缓存是处理器与主存之间的高速存储器，用于暂时存储频繁访问的数据。高缓存命中率意味着大部分数据访问都在缓存中完成，从而减少对主存的访问，提高计算效率。缓存命中率低则会导致更多的主存访问，增加延迟和计算成本。

优化缓存命中率的策略

1. 数据分块（Block Processing）：

- 将数据分成多个较小的块，每个块可以独立处理并尽量保持在缓存中。
- 通过分块处理，减少每次计算所需加载的数据量，提高缓存命中率。

2. 列块化（Column Blocked Access）：

- XGBoost使用了一种称为列块化的数据结构，将特征按列存储，使得对同一列的访问局部化，从而提升缓存命中率。
- 在树的构建过程中，频繁访问的是特征值和梯度，列块化可以使这些数据更容易保存在缓存中。

3. 预取技术（Prefetching）：

- 通过硬件或软件预取技术，提前将即将访问的数据加载到缓存中，减少等待时间。
- XGBoost可以利用现代处理器的预取指令，提前加载特定数据块，提高缓存命中率。

4. 稀疏矩阵优化 (Sparse Matrix Optimization) :

- 对于稀疏数据，XGBoost使用稀疏矩阵存储方式，只存储非零元素，减少内存占用和访问开销。
- 在梯度提升树的构建过程中，稀疏矩阵的使用可以减少无用数据的加载，提高缓存利用率。

5. 内存对齐 (Memory Alignment) :

- 确保数据结构在内存中的对齐方式，减少缓存行的分裂，提高访问效率。
- XGBoost优化了内存对齐策略，使得数据访问更加高效。

代码示例：优化缓存命中率

以下是一个示例，展示如何通过XGBoost的参数设置和优化策略来提高缓存命中率：

```
import xgboost as xgb
from sklearn.datasets import load_boston
from sklearn.model_selection import train_test_split
from sklearn.metrics import mean_squared_error

# 加载数据
data = load_boston()
X = data.data
y = data.target

# 划分训练集和测试集
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# 创建DMatrix对象
dtrain = xgb.DMatrix(X_train, label=y_train)
dtest = xgb.DMatrix(X_test, label=y_test)
```

```
# 设置参数，包括缓存优化相关参数
params = {
    'objective': 'reg:squarederror',
    'max_depth': 3,
    'eta': 0.1,
    'tree_method': 'hist', # 使用 histogram-based tree method 支持缓存优化
    'subsample': 0.8,
    'colsample_bytree': 0.8,
    'nthread': 4 # 使用多线程，提高并行处理能力
}

# 训练模型
bst = xgb.train(params, dtrain, num_boost_round=100)

# 预测
y_pred = bst.predict(dtest)

# 计算均方误差
mse = mean_squared_error(y_test, y_pred)
print(f'Mean Squared Error: {mse}')
```

在上述代码中，`tree_method` 参数设置为 `hist`，这是一种支持缓存优化的树构建方法。通过合理设置 `subsample` 和 `colsample_bytree` 参数，可以减少每次计算的数据量，进一步提升缓存命中率。

总结

通过数据分块、列块化、预取技术、稀疏矩阵优化和内存对齐等多种策略，XGBoost 能够有效提高缓存命中率，提升计算效率。这些优化策略在处理大规模数据和高维数据时尤为重要，能够显著减少计算时间和资源消耗。