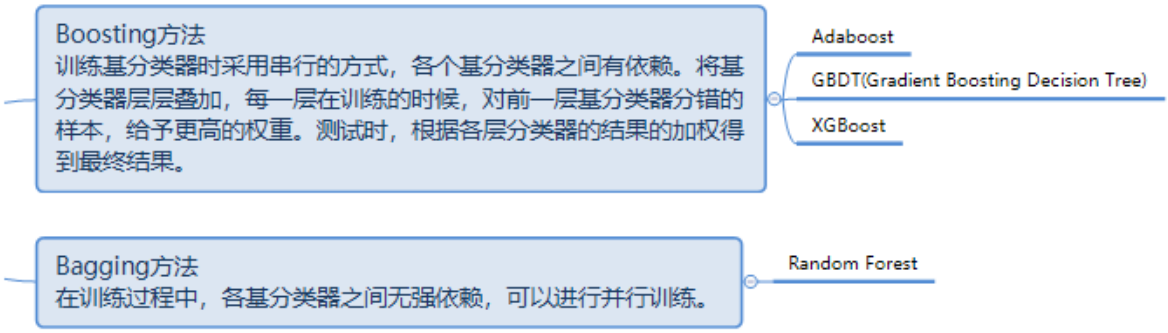


XGBoost属于boosting方法之一。

一、XGBoost的算法思想：

残差：A的实际值 - A的预测值 = A的残差

1. **训练阶段不断地添加树，去拟合之前预测的残差。树中的每一个节点都进行特征分裂，直到满足特定的条件（树的最大深度、增益小于阈值等）；**
2. **训练过程结束得到t棵树，对于样本的预测，就是根据样本的特征在每棵树中归到一个叶子结点，每个叶子结点输出一个预测值；**
3. **将每棵树对应的预测值求和就是该样本的预测值。**

二、最小化目标函数：

我们的目标就是使得建立的树群对样本的预测值$\hat{y}_i$ 尽量接近真实值y_i 。

损失函数：

$$L = \sum_{i=1}^{n} l(y_i, \hat{y}_i) \tag{1}$$

模型的预测精度由模型的偏差和方差共同决定，损失函数代表了模型的偏差，想要方差小则需要在目标函数中添加正则项，用于防止过拟合。所以目标函数由模型的损失函数L 与抑制模型复杂度的正则项Ω 组成，最终的目标函数定义如下：

$$Obj = \sum_{i=1}^{n} l(y_i, \hat{y}_i) + \sum_{i=1}^{t} \Omega(f_i) \tag{2}$$

后半部分是正则化项，将t棵树的复杂度求和（值越小复杂度越低，泛化能力越强）。

接下来是公式推导的部分：

t棵树的预测结果与t-1棵树的预测结果有关：其中$\hat{y}_i^{(t-1)}$ 是t-1步给出的预测值，常量；$f_t(x_i)$是第t棵树加入的预测值。

$$\hat{y}_i^{(t)} = \hat{y}_i^{(t-1)} + f_t(x_i) \tag{3}$$

将正则化项进行拆分，由于前t-1棵树的结构已经确定，因此前t-1棵树的复杂度之和可以用一个常量

表示，如下所示：

$$\begin{aligned}\sum_{i=1}^{t}\Omega(f_i) &= \Omega(f_t)+\sum_{i=1}^{t-1}\Omega(f_i)\\ &= \Omega(f_t)+constant\end{aligned} \tag{4}$$

将（3）和（4）带入（2）中：

$$\begin{aligned}Obj^{(t)} &= \sum_{i=1}^{n} l(y_i,\hat{y}_i^{(t)})+\sum_{i=1}^{t}\Omega(f_i)\\ &= \sum_{i=1}^{n} l(y_i,\hat{y}_i^{(t-1)}+f_t(x_i))+\sum_{i=1}^{t}\Omega(f_i)\\ &= \sum_{i=1}^{n} l(y_i,\hat{y}_i^{(t-1)}+f_t(x_i))+\Omega(f_t)+constant\end{aligned} \tag{5}$$

注意（5）式子中，变量就是$f_t(x_i)$和 Ω，也就是第t棵树加入的预测值和第t棵树的正则化项。

泰勒展开

$$f(x+\Delta x)\approx f(x)+f'(x)\Delta x+\frac{1}{2}f''(x)\Delta x^2 \tag{6}$$

对损失函数进行在 x=$\hat{y}_i^{(t-1)}$ 处进行**二阶泰勒展开，**$f_t(x_i)$作为 Δx ，损失函数作为函数体：

$$l(y_i,\hat{y}_i^{(t-1)}+f_t(x_i))=l(y_i,\hat{y}_i^{(t-1)})+g_i f_t(x_i)+\frac{1}{2}h_i f_t^2(x_i) \tag{7}$$

g_i 为损失函数一阶导，h_i 是损失函数二阶导，由于前一步和两步的$\hat{y}^{(t-1)}$**的一阶二阶导都是已知的，因此这两个数也是常量。**。

将展开后的损失函数（7）带入目标函数（5）中：

$$Obj^{(t)}\simeq\sum_{i=1}^{n}[l(y_i,\hat{y}_i^{t-1})+g_i f_t(x_i)+\frac{1}{2}h_i f_t^2(x_i)]+\Omega(f_t)+constant \tag{8}$$

其中$l(y_i,\hat{y}_i^{(t-1)})$ 也是常数项，因此去掉后剩余的目标函数为：

$$Obj^{(t)}\simeq\sum_{i=1}^{n}[g_i f_t(x_i)+\frac{1}{2}h_i f_t^2(x_i)]+\Omega(f_t) \tag{9}$$

正则项由生成的第t棵决策树的叶子结点数量和所有结点权重所组成的向量的L_2 范式共同决定，**有两**

个可调系数， γ 可以控制叶子结点的个数，λ 可以控制叶子节点的权重不会过大，防止过拟合：

$$\Omega(f_t) = \gamma T + \frac{1}{2}\lambda \sum_{j=1}^{T} w_j^2 \tag{10}$$

Number of leaves

L2 norm of leaf scores

叶子结点权重向量的L_2范数

带入（9）中，**将原本遍历n个样本在第t棵树叶子结点上的预测之和变为遍历所有叶子结点后再遍历落在结点上的样本集**（因为每个样本最终都会落到叶子结点上，这样转换得以合并）：

$$\begin{aligned} Obj^{(t)} &\simeq \sum_{i=1}^{n}\left[g_i f_t(x_i) + \frac{1}{2}h_i f_t^2(x_i)\right] + \Omega(f_t) \\ &= \sum_{i=1}^{n}\left[g_i w_{q(x_i)} + \frac{1}{2}h_i w_{q(x_i)}^2\right] + \gamma T + \frac{1}{2}\lambda\sum_{j=1}^{T} w_j^2 \\ &= \sum_{j=1}^{T}\left[(\sum_{i\in I_j} g_i)w_j + \frac{1}{2}(\sum_{i\in I_j} h_i + \lambda)w_j^2\right] + \gamma T \end{aligned} \tag{11}$$

定义 $G_j = \sum_{i\in I_j} g_i$ ：叶子结点 j 所包含样本的一阶偏导数累加之和，常量；

定义 $H_j = \sum_{i\in I_j} h_i$ ：叶子结点 j 所包含样本的二阶偏导数累加之和，常量；

因此最终的目标函数为：

$$Obj^{(t)} = \sum_{j=1}^{T}\left[G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2\right] + \gamma T \tag{12}$$

其中真正的变量只有包含最后一棵树所有叶子结点权重的向量 w_j 不确定。

目标函数求极值

对于最后一棵树的每一个叶子结点 j，单独的都有：

$$G_j w_j + \frac{1}{2}(H_j + \lambda)w_j^2 \tag{13}$$

这个式子是只包含一个变量**叶子结点权重** w_j 的一元二次函数，可以直接求其极值点。

而叶子结点之间相互独立，因此，每个叶子结点子式都达到极值时，目标函数才达到极值点。可以求得叶子结点 j 对应的权值：

$$w_j^* = -\frac{G_j}{H_j + \lambda} \tag{14}$$

所以目标函数的极值可以求出，越小越好：

$$Obj = -\frac{1}{2}\sum_{j=1}^{T}\frac{G_j^2}{H_j + \lambda} + \gamma T \tag{15}$$

至此公式推导结束

有了目标函数，根据求极值的方法就能够对叶子结点权重进行赋值。当我们训练完成得到kt棵树，要预测一个样本的分数，其实就是根据这个样本的特征，在每棵树中会落到对应的一个叶子节点，每个叶子节点就对应一个分数，最后只需要将每棵树对应的分数加起来就是该样本的预测值。

三、树的生成

每棵树是如何分裂的，选取什么特征作为切点很关键。

贪心算法

从树的深度为0开始：

1. 对每个叶节点枚举所有的可用特征；
2. 针对每个特征，把属于该节点的训练样本根据该特征值进行升序排列，通过线性扫描的方式来决定该特征的最佳分裂点，并**记录该特征的分裂收益**；
3. 选择收益最大的特征作为分裂特征，用该特征的最佳分裂点作为分裂位置，在该节点上分裂出左右两个新的叶节点，并为每个新节点关联对应的样本集；
4. 回到第1步，递归执行直到满足特定条件为止；

计算每个特征的分裂收益

分裂前目标函数：

$$Obj_1 = -\frac{1}{2}[\frac{(G_L+G_R)^2}{H_L+H_R+\lambda}]+\gamma$$

分裂后目标函数：

$$Obj_2 = -\frac{1}{2}[\frac{G_L^2}{H_L+\lambda}+\frac{G_R^2}{H_R+\lambda}]+2\gamma$$

分裂后收益：

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

计算每次分裂的分割方案

线性扫描，计算所有分割方案对应的特征分裂的Gain，哪种最大选取哪种划分方法。对于所有的特征，我们只要做一遍从左到右的扫描就可以枚举出所有分割的梯度和GL和GR。然后用计算Gain的公式计算每个分割方案的分数就可以了。

限制树的生成和分裂

1. 对于引入分割后的复杂度过高，带来的增益小于阈值γ（正则项中叶子结点树T的稀疏）时，则忽

略分割（类似于预剪枝）；
2. 树达到最大深度max_depth，停止建树，避免树太深导致学习局部样本从而过拟合；
3. 一个叶子结点的样本数太少了，防止过拟合；
4. 树的最大数量

四、AutoScheduler中使用

数据转换

在AutoScheduler中，特征通过C++相关代码获取并打包，在Python进行解包。

```
features, normalized_throughputs, task_ids =
    get_per_store_features_from_measure_pairs(
    self.inputs, self.results, skip_first_n_feature_extraction=n_cac
```

即输入数据包含“特征”，“标准化吞吐量”和“任务id”。
然后将这几种数据打包为xgb.DMatrix格式，并按照task id按顺序排列：

```
dtrain = pack_sum_xgbmatrix(
    features, normalized_throughputs, task_ids, normalized_throughput
)
```

> 在pack_sum_xgbmatrix（）中，通过一系列子任务排序等操作，最终将(feature, label)也就是特征和标签以xgb.DMatrix格式返回

模型训练

完成数据转换后，在xgboost更新函数中完成模型的更新训练 （XGBoost不支持增量训练，所以每次都重新训练一个新模型）：

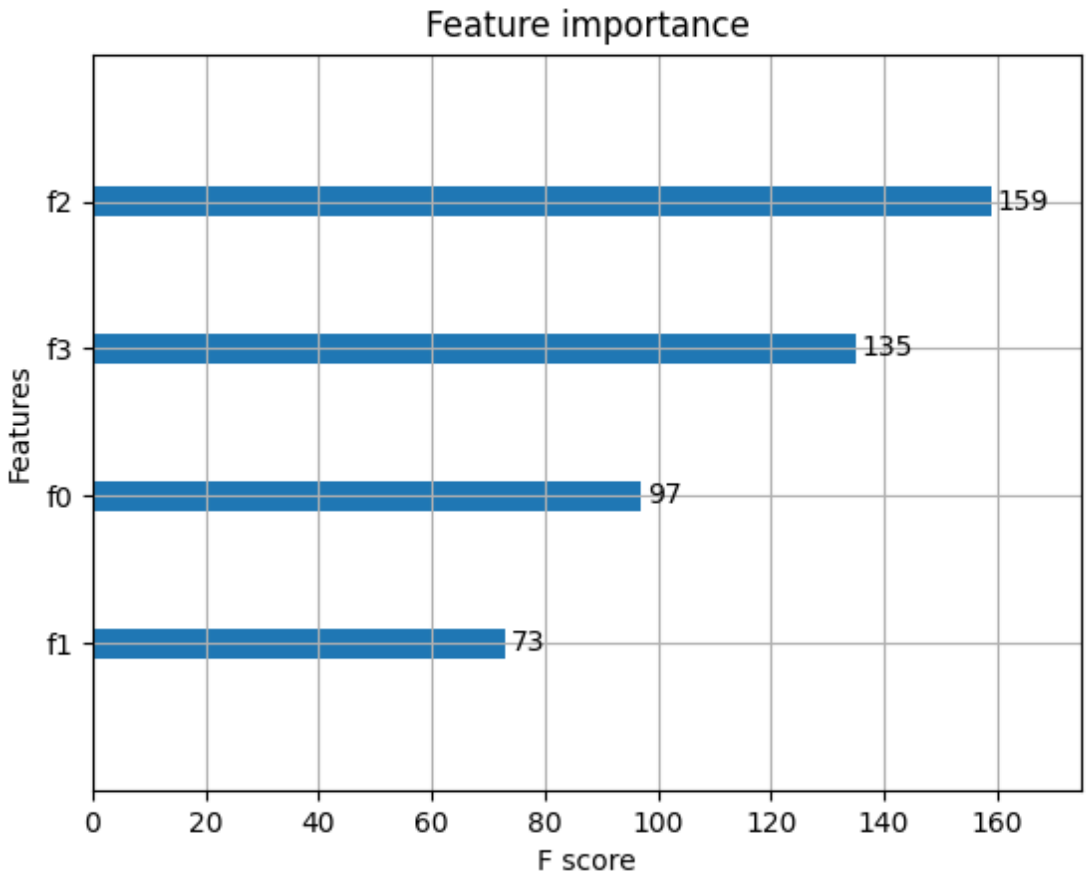
```
self.bst = xgb.train(
            self.xgb_params,
            dtrain,
            num_boost_round=10000,
            obj=pack_sum_square_error,
            callbacks=[
                custom_callback(
                    stopping_rounds=50,
                    metric="tr-p-rmse",
                    fevals=[#函数宏指令；执行由串指定的函数
                        pack_sum_rmse,
                        pack_sum_average_peak_score(self.plan_size),
                    ],
                    evals=[(dtrain, "tr")],
                    maximize=False,
                    verbose_eval=self.verbose_eval,
                )
            ],
        )
```

预测

```
raw_preds = self.bst.predict(dtest)
```

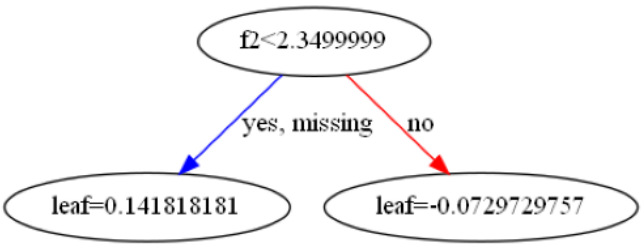
五、调用结果

XGBoost可以用于解决回归和分类问题。我们的项目是解决回归问题。
下面是XGBoost完成训练后，对不同的特征的重要性排序（值越大说明特征越重要，其所在树的节点也越上层）：

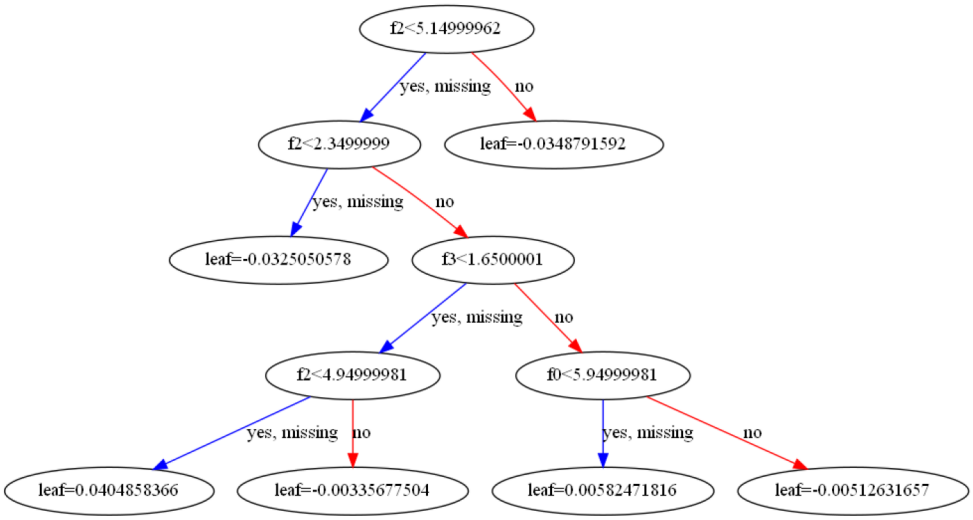


训练结果中的决策树：

num_trees=0第一棵树



num_trees=100



六、C++调用XGBoost模型

由于XGBoost并没有提供C++接口，因此网上关于这方面的解决办法通常是采用C++调用C接口或者C++调用Python函数。

XGBoost训练出来的k棵树保存为文本文件：

```
booster[21]:
0:[f0<5.84803009] yes=1,no=2,missing=1
    1:leaf=0.0851583481
    2:leaf=0.0711555183
booster[22]:
0:[f12<16.1300011] yes=1,no=2,missing=1
    1:leaf=0.0848125666
    2:leaf=0.0719704106
booster[23]:
0:[f10<19.9000015] yes=1,no=2,missing=1
    1:leaf=0.0838522911
    2:leaf=0.0745095685
```

观察输出的XGBoost模型，可以看到这个还是比较容易解析的（从0开始，根据特征进行分裂，yes、no和缺失都有对应的叶子结点号），因此我选择直接对TXT文本进行解析并生成对应的.cpp代码。

```
booster_code += "{0}if (sample[{1}] {2}) {{\n".format("  " * indentation_level, feature_index, comparison)
booster_code += get_single_booster_cpp_code(booster_tree, yes_branch_id, indentation_level + 1)#递归遍历所有的分支
booster_code += "{0}}} else {{\n".format("  " * indentation_level)
booster_code += get_single_booster_cpp_code(booster_tree, no_branch_id, indentation_level + 1)#递归遍历二叉树的另一个分支
booster_code += "{0}}}\n".format("  " * indentation_level)
```

这里用两个递归解析二叉树的两个分支直到访问叶子结点。

```
if 'leaf' in level[0]:#只有叶子结点，作为递归的终止条件
    booster_code += "{0}sum += {1};\n".format("  " * indentation_level, float(level[0].split('=')[1]))
    return booster_code
```

终止条件：只有叶子结点（即叶子结点的leaf在第一行）。

生成的.cpp文件：

```
if (sample[0] <6.88165998) {
  sum += 0.0914210528;
} else {
  sum += 0.0818093494;
}


sum += 0.0897860676;


if (sample[12] <14.9150009) {
  sum += 0.0905708075;
} else {
  sum += 0.0821594968;
}
```

通过这样的方式对输入的样本累积其在每一棵树的叶子结点的权重，从而得到最终的预测值。

七、存在的问题

1. 将C++调用训练好的模型进行预测得到的值与python调用xgboost预测函数得到的结果相差大。

训练数据：波士顿房价数据集

```
1 from sklearn.datasets import load_boston
```

预测数据：

```
1 std::vector<float> test_sample{ 2.6169e-01, 0.0000, 9.9000e+00, 0.00(
```

C++得到的预测结果：3.72584

Python得到的预测结果：20.752954

实际标签：19.4

分析原因：**参考网上对于分类模型的累积值需要做后处理得到概率值，应该是权重积累值和最终的输出存在转换关系，这部分在xgboost源码中实现。**

首先，predict函数无参数，默认输出的是概率值。

当设置output_margin=True时，输出的不是概率值，而是每个样本在xgboost生成的所有树中叶子节点的累加值，因为在训练模型时，objective选择了logistic，所以用这个累加值做sigmoid就是predict函数无参数时的概率值。如何知道每个样本在xgboost生成的所有树中是哪个叶子节点呢?

当设置pred_leaf=True时，会输出每个样本在所有树中的叶子节点

ypred_leaf = bst.predict(dtest, pred_leaf=True)

即测试集有25个样本，xgboost生成了10棵树，则样本在所有树中的叶子节点对应了一个（25, 10）的矩阵。例如，第一个样本在10棵树中的叶子节点都是第一个。第二个样本在10棵树中的叶子节点都是第二个。

再看树结构，共有10个booster，编号0~9，对应10棵树。对于第一个booster，当f3<0.75时，分到leaf1，f3>=0.75时，分到leaf2，缺失该特征，分到leaf1。我们将10棵树中的leaf1的值全部加起来，即：

-0.0787879-0.0823941-0.0743093-0.0591238-0.0662177-0.0670728-0.068963-0.0610322-0.0576446-0.0599772=-0.6755226

-0.6755226即为叶子节点权重的累加值，在这里，所有叶子节点的权重已经乘了学习率，对-0.6755226求sigmoid：

```
1 / float(1 + np.exp(0.6755226)) = 0.337261343368 #这个值跟输出的概率值是对应的。
```

这个问题还在研究（回归树直接对样本在所有叶子节点的权重加和得到的是对数尺度的预测值，因此指数化即可得到回归的预测值）。

预测：src/learn.er.cc

```
void Predict(std::shared_ptr<DMatrix> data, bool output_margin,
             HostDeviceVector<bst_float> *out_preds, unsigned laye
             unsigned layer_end, bool training,
             bool pred_leaf, bool pred_contribs, bool approx_contr
             bool pred_interactions) override {
```

转换：src/objective/aft_obj.cu

```
    void PredTransform(HostDeviceVector<bst_float> *io_preds) const
    // 树返回对数尺度，指数化
    common::Transform<>::Init(
        [] XGBOOST_DEVICE(size_t _idx, common::Span<bst_float> _prec
      _preds[_idx] = exp(_preds[_idx]);
    }, common::Range{0, static_cast<int64_t>(io_preds->Size())},
        io_preds->DeviceIdx())
    .Eval(io_preds);
  }
```

C++得到的预测结果：3.72584

Python得到的预测结果：20.752954

实际标签：19.4

2. 现在的方案是Python完成模型的训练更新，C++调用模型文件进行预测，因此训练和预测分离，虽然简单可行，但是后期在线训练难以实现。**因此可以继续研究在C++中调用其他接口的方法。**

八、问题的解决：

问题1

XGBoost设计了多种目标函数用于不同的场景，不同场景下决策树叶子结点权重累加值的意义不同。例如：

1.逻辑回归 LogisticRegression ，权值累加值经过sigmoid映射到（0,1）之间，用于二分类：

```
static T PredTransform(T x) { return common::Sigmoid(x); }
```

2.加权平方损失函数 SquaredLogError，权值累加值直接输出，用于回归：

```
XGBOOST_DEVICE static bst_float PredTransform(bst_float x) { return x; }//直接返回
```

3.线性回归 LinearSquareLoss，直接输出，回归：

```
XGBOOST_DEVICE static bst_float PredTransform(bst_float x) { return x; }//直接返回
```

对于本项目来说，采用多维多特征的线性回归即可，因此在训练时选择reg:linear作为目标函数参数，得到结果如下：

```
预测值（特征输入）
test_sample{ 0.26169, 0.0000, 9.9000, 0.0000, 0.54400 ,6.0230, 90.40
真实值（房价）
19.4
Python预测
18.663061
C++预测
18.6631
```

```
predict result:18.6631
Starting classification.
Classified 10000 samples in 5861.25 miliseconds.
0.586125ms per sample.
```