**Lab 3 实验报告**

PB20000296 郑滕飞

**目的/原理：**

**1、实验目的**

以回归树为基学习器，通过XGBoost算法集成为最终的学习器，以对连续属性作出预测。

**2、实验原理-XGBoost**

\*参考自zhuanlan.zhihu.com/p/75217528

XGBoost算法对每个学习器的结果是直接相加关系，而在训练每个学习器时都是优化最终误差+正则化惩罚项。与传统梯度提升[Gradient Boost]相比，XGBoost作出了三项优化：添加了正则化惩罚项避免过拟合、展开到二阶项得出更精确信息、可以处理缺失值。

不过，由于我们的实验中使用的损失函数是平方损失，二阶导数为常数，实际上没有信息，且数据集中并没有缺失值，实际体现的优化方式只有正则化项部分。

一般的XGBoost算法分为两部分，找第一个与有前t-1个时训练第t个。前者为方便起见，直接以全零作为初始预测，而对后者，训练时一般公式是优化

其中分别为泰勒展开中得到的损失函数的一阶、二阶导数。

**3、实验原理-回归树**

由于此为对连续值预测，基学习器采用了回归树。一棵回归树事实上可以看作一个数据到叶结点的映射与叶结点的结果，因此这里的只与叶结点有关。假设树的结构确定，有T个叶结点，每个的结果[在Boosting的视角为权重]为，则上方的公式变为

其中代表映射到第j个叶结点的数据的g与h求和。

由于这对是二次函数，最优时有

而事实上，在实际设计算法时，就是根据新的树的Obj比起原树增加的程度[Gain]来决定在何处进行划分。根据Obj公式容易计算出，划分结点的收益为

其中L，R分别代表划分后左、右的子集。

**实验步骤：**

**1、整体框架**

我的整体框架分为三个部分：学习器类定义[学习器的基本代码]、读取文件，并多次划分、训练、测试考察平均性能。通过

yp = boosttree.predict(X\_test)

rmse = np.sqrt(np.sum((yp-y\_test) \*\* 2) / y\_test.size)

r2 = 1 - np.sum((yp-y\_test)\*\*2) / np.sum((yp-np.average(y\_test))\*\*2)

分别计算出每次的预测结果，对应的均方误差与，并且作出损失曲线进行比较。

**2、XGBoost**

排除去构造树的部分，XGBoost算法较为简单，也即重复执行：预测、构造当前预测对应的g[由于h是常数，不需要构造]、由g训练基学习器。

self.trees.append(self.\_fit\_node(X, 2 \* gdiv2, 0))

gdiv2 = self.predict(X) - y

其中，gdiv2是当前预测值与真实值的差距，对损失函数求导可以发现其也就是g值除以2。

注意到，整个决策树的生成过程中，除了属性集外只用到了g，并不会用到y，因此只需要传入g即可。第三个参数0代表深度，方便在深度达到阈值时停止。

而关于结束策略，通过gdiv2容易算出均方误差，若训练新树导致的均方误差减小不到某个阈值[这里设定为1e-7]，就直接停止。此外，设置了最大树数量的限制，避免不收敛时进入无限循环。不过，在实际测试中，从来没有到达过默认20的最大树数量。

**3、决策树**

生成决策树的部分fit node就较为复杂了。总体思路是，每次选取增益最大的分割点进行分割[注意到分割只涉及顺序，因此没有必要对属性作归一化]，直到达到停止条件：

node = BTree()

feature, split, max\_gain = self.\_get\_best\_split(X, g)

if max\_gain < self.gain\_tol\

or depth >= self.max\_depth\

or length(X) <= self.min\_num:

set node’s property as a leaf

else:

set node’s property as an inner node

split data to l, r

node.lchild = \_fit\_node(x[l],g[l],depth+1)

node.rchild \_fit\_node(x[r],g[r],depth+1)

这里的停止条件综合了增益、深度与结点中最小的数据量，关于停止条件的后续比较会在结果展示中进行。

而对于具体的获得划分的过程，根据

可以计算。不过，注意到对于某个特定属性，只要进行排序，就有L每次增加一个，R每次减少一个，因此**预排序可以大大简化计算增益的过程**。

具体的方式是，对属性进行遍历，在针对某一属性寻找最优划分时，可以先将这个属性连同g一起排序，并将初始的GL设置为0，GR设置为G[也即所有结点都在左侧]，接着进行划分：

for i = 0 to size-2:

Gl += g[i], Gr -= g[i], Hl += 2, Hr -= 2

if feature[i] == feature[i+1]:

continue

score = Gl\*\*2 / (Hl + self.lam) + Gr\*\*2 / (Hr + self.lam) - root\_score

if score > max\_score:

max\_score = score

split\_best = (feature[i] + feature[i+1]) / 2

由于每个属性有大量重复，只需要在排序前一个与下一个有不同的地方进行划分比较，此外，这里计算的score并不是真实的Gain值，Gain值为其除以2后减去gamma。由于获取最优划分点只需要找到最大处，过程并不需要得到Gain，只要返回时操作最大的score即可。

最后，通过每个属性的最优划分Gain进行比较，即得到了整体的最优划分，结合上方就完成了算法。

**展示/分析：**

\*默认参数[经过寻找后比较好的参数效果]：

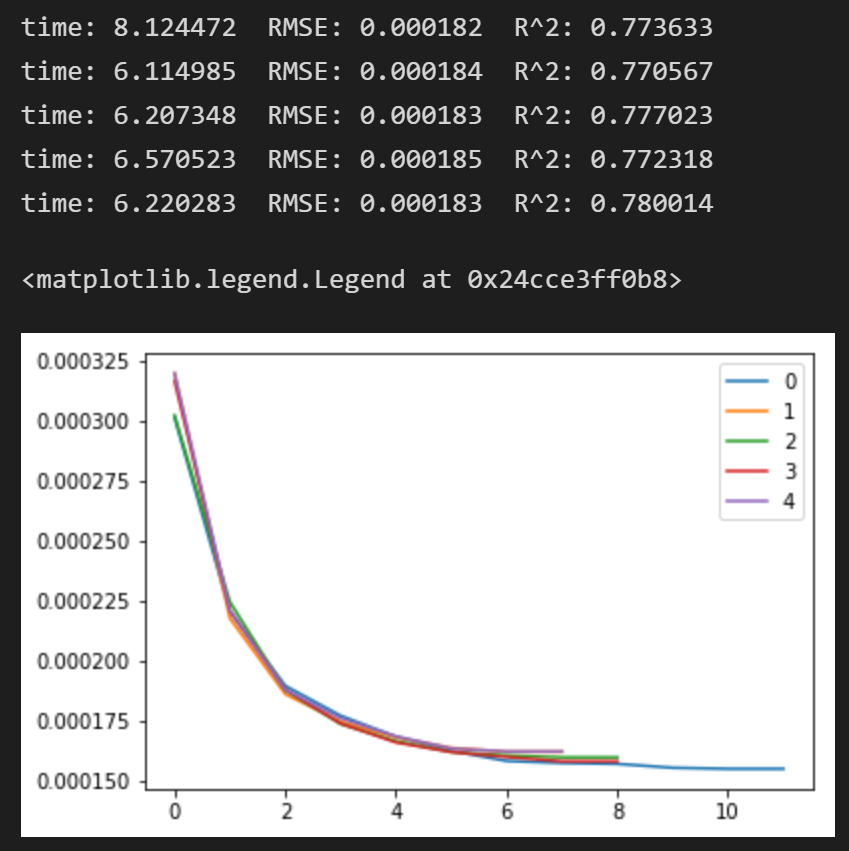
lam = 150, gamma = 1e-7, tol = 1e-7

gain\_tol = 1e-7, max\_depth = 10, min\_num = 20

[默认max\_tree = 20，但以下的讨论中没有出现达到最大数量的情况]

划分比例7:3

**1、输出结果示例**



这是用默认参数进行五次训练、测试的结果，可以发现平均在7秒以内就已经达到了0.00018的均方误差与0.77的，足见XGBoost结合回归树算法的效果之好。

损失曲线绘制的是每训练出一棵树后在训练集上的均方误差，通过横轴可以看出每次训练得到的回归树个数。

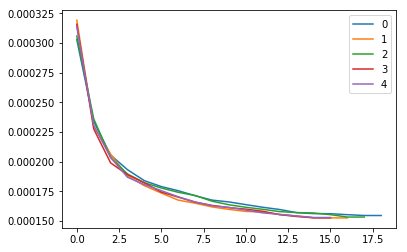
**2、停止策略对比**

具体的过程见ipynb文件，这里给出停止策略的结果[测试五次]：

|  |  |  |  |
| --- | --- | --- | --- |
| **策略** | **中位时间** | **中位RMSE** | **中位** |
| 增益小于1e-6 | 3.24s | 1.94e-4 | 0.729 |
| 增益小于1e-7 | 7.31s | 1.84e-4 | 0.762 |
| 增益小于1e-8 | 9.61s | 1.85e-4 | 0.771 |
| 深度高于5 | 14.23s | 1.83e-4 | 0.775 |
| 深度高于7 | 14.19s | 1.82e-4 | 0.784 |
| 深度高于10 | 18.16s | 1.86e-4 | 0.762 |
| 集合低于200 | 18.89s | 1.86e-4 | 0.771 |
| 集合低于300 | 17.77s | 1.84e-4 | 0.773 |
| 集合低于500 | 21.79s | 1.81e-4 | 0.786 |

\*集合低于若干的测试额外添加了深度高于10的终止条件，否则会导致划分过多

从结果可以发现，总体来说深度结合集合低于值的策略性能最稳定，也相对最好。值得注意的是，单棵树结束划分快并不代表总时间短，因为可能会导致划分的树更多。表现最好的集合低于500+深度高于10的终止策略，平均划分了16棵树，损失函数如下：



可以分析，这种情况下收敛慢意味着某种“**好而不同**”，因此在集成学习中最后表现出了更好的效果。不过，它的时间开销也显著大。

**3、参数对比**

下面在默认停止策略的情况下对比参数：

|  |  |  |  |
| --- | --- | --- | --- |
| **策略** | **中位时间** | **中位RMSE** | **中位** |
| lambda = 50 | 5.64s | 1.90e-4 | 0.755 |
| lambda = 100 | 6.31s | 1.84e-4 | 0.777 |
| lambda = 200 | 7.94s | 1.80e-4 | 0.774 |
| 默认参数 | 6.46s | 1.90e-4 | 0.751 |
| gamma = 1e-5 | 1.75s | 2.32e-4 | 0.564 |
| gamma = 1e-6 | 3.75s | 1.94e-4 | 0.738 |
| gamma = 0 | 10.13s | 1.85e-4 | 0.766 |

同样可以发现，时间开销较大的时候意味着倾向于训练出的树更多才收敛，结果一般更好。此外，由于总树基本不会超过20，过拟合可能性不大，不加入正则化项或许效果更好。

**总结：**

这次实验的任务难度主要在于两部拆分的构造，尤其是回归树部分如何优化。此外，在梯度提升每次优化误差的思想改进下，XGBoost能得到更加快且精确的结果。根据最后的对比部分，个体学习器少进行一些划分、增加学习器的数量会对集成产生较明显的正面效果。