**数据科学竞赛：递增特征构建的简单实现**

**一、引言**

在智能风控或者其他的数据科学竞赛当中，我们经常可以从用户的基础信息表中发现类似这样的特征：

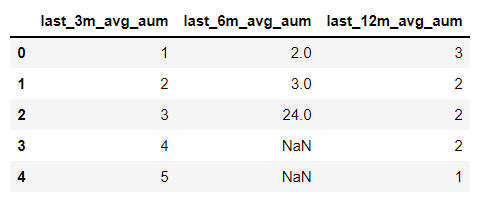
|  |  |
| --- | --- |
| **字段英文名** | **字段含义** |
| last\_3m\_avg\_aum | 近3个月均aum |
| last\_6m\_avg\_aum | 近6个月均aum |
| last\_12m\_avg\_aum | 近12个月均aum |
| last\_3m\_avg\_times | 近3个月取款平均值 |
| last\_3m\_avg\_times | 近6个月取款平均值 |
| last\_3m\_avg\_times | 近12个月取款平均值 |

这些原始的特征在建模中有着自己的“贡献力”，很显然基于这些字段我们可以生成很多的特征。有一种特征就是关于这些变量的递增变量，什么意思呢？就是3个月均aum之间的关系：如果是递增的就将新生成的特征记录为1，反之记录为0

**二、数据准备**

在进行实验之前我们进行数据的准备，我们设置的实验数据如下：

data = [  
 [1,2,3,],  
 [2,3,2],  
 [3,24,2],  
 [4,np.nan,2],  
 [5,np.nan,1]  
 ]  
columns = ['last\_3m\_avg\_aum','last\_6m\_avg\_aum','last\_12m\_avg\_aum']  
data\_df = pd.DataFrame(data,columns=columns)  
data\_df



**三、列递增**

假设我们现在的需求是判断某一列数据是否是递增的，这个怎么实现呢？我们可以遍历某一列数据进行下一个值与当前值的比较。显然这个办法比较蠢，还好pandas中实现了一个方法我们可以直接的调用，比如以下几个例子（代码使用jupyter notebook）：

data\_df['last\_3m\_avg\_aum'].is\_monotonic  
data\_df['last\_3m\_avg\_aum'].is\_monotonic\_increasing  
data\_df['last\_12m\_avg\_aum'].is\_monotonic  
data\_df['last\_12m\_avg\_aum'].is\_monotonic\_decreasing  
  
依次输出：  
True  
True  
False  
True

可以看出：

**is\_monotonic**用来判断严格单调的（不论是单调递增还是单调递减）

**is\_monotonic\_increasing**用来判断是否递增（不是严格的也行）

**is\_monotonic\_decreasing**用来判断是否递减（不是严格的也行）

那么关于这几个方法的实现细节，大家可以转官方文档进行查看。

这是关于列递增的方式，使用Pandas自带的方法就可以完成。

**四、行递增**

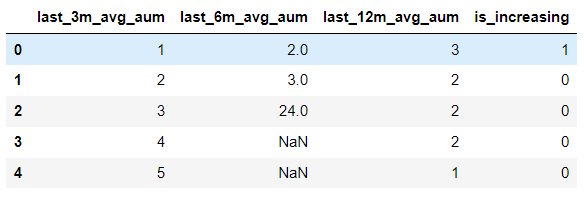
上述方式判断是列递增，那么怎么实现行数据的递增判断呢？接下来我们提供几种思路和方法供大家考虑，也欢迎小伙伴提出新的方法：

（1）第1种方法就是简单的使用for循环取数进行判断，由于过于简单就不在演示。

1. 第2种方法是对目标dataframe进行转置，再使用自带的方法进行判断，接下来我将写一个函数，用来判断每一行数据是否都是递增的，并新增一列来存储判断的结果：

import gc  
def progressive\_increasing(df,columns,newcolname):  
 df\_T = df[columns].T  
 df[newcolname] = df\_T.apply(lambda x:x.is\_monotonic)  
 df[newcolname] = df[newcolname].astype(int)  
 del df\_T  
 gc.collect()  
 return df  
columns = ['last\_3m\_avg\_aum','last\_6m\_avg\_aum','last\_12m\_avg\_aum']  
new\_df = progressive\_increasing(data\_df,columns,'is\_increasing')

打印new\_df的输出为：



is\_increasing列存储了我们需要的特征，上述函数还是比较好用的，传入三个参数就可以实现我们的需求。

是不是觉得自己特别的秀呢？但是在实际使用的时候会出现一个问题，当我们的数据量很大的时候，比如说我当时处理**100W**个样本的时候，此时这个函数就跑不动了。那么出现的问题在哪呢？大家自己找答案。

**找答案的时候我们会发现一个新的问题：大矩阵/大稀疏矩阵的转置问题。**

**感觉又有话题讨论了，不过这次我们不讨论。**

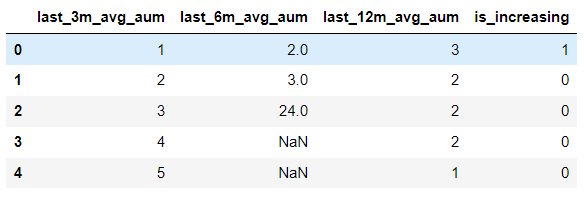
另外上述方法中有一个小技巧，就是使用了gc.collect()，可以帮助我们在大量数据的情况下节省点内存。

为了解决这个函数跑不动的问题，我就又写了一个方法。这个方法最终是我构建特征的方案。

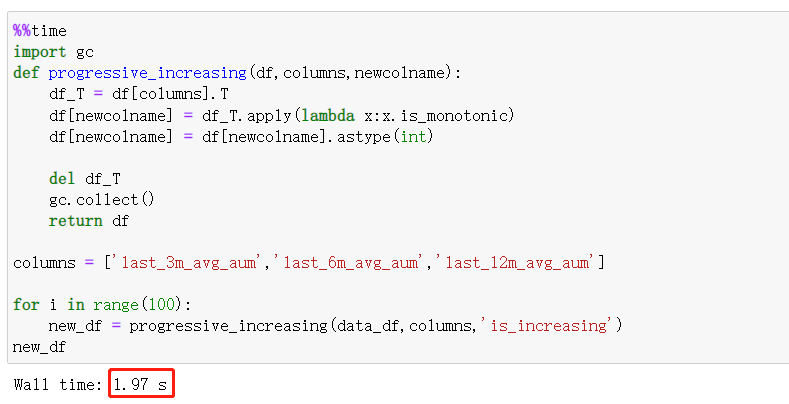
1. 方法3说简单也是非常简单的，talk is cheap, show you the code：

def progressive\_increasing\_use(df,columns,newcolname):  
 df[newcolname] = (df[columns[1]] > df[columns[0]]) & (df[columns[2]] > df[columns[0]])  
 df[newcolname] = df[newcolname].astype(int)  
 return df  
columns = ['last\_3m\_avg\_aum','last\_6m\_avg\_aum','last\_12m\_avg\_aum']  
new\_df = progressive\_increasing\_use(data\_df,columns,'is\_increasing')  
new\_df

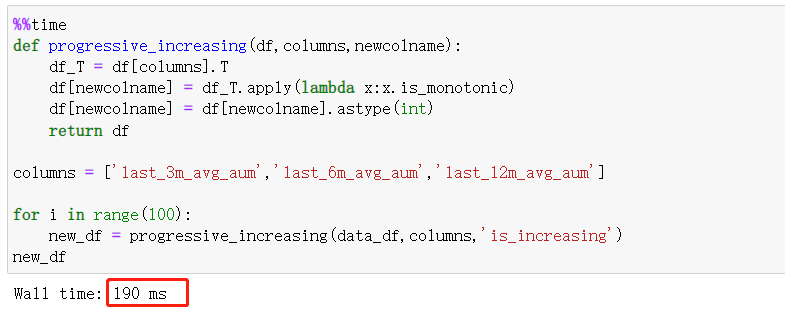
输出如下：



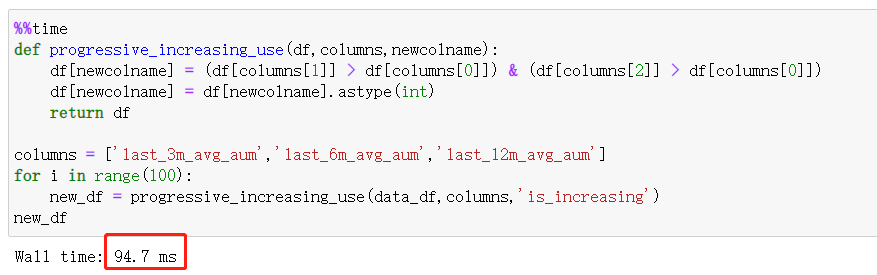
结果与上述第2种方法一模一样，那么该方法与第二个方法相比有运行速度上有什么区别呢？我们使用 %%time来看一下：



100次运行消耗的时间1.97s，如果我们把内存回收的代码出去，我们来看一下时间：



这个时间要省很多了，大概190ms。由于dataframe比较小，所以这个回收一下内存的时间就显得比较大。再看一下第3种方法的时间：



显然第3种方法消耗的时间是最短的。当我们处理的dataframe很大的时候，不同方法之间的时间差距会拉开的更多，大家可以创建一个超大的dataframe进行试验一下。

1. **总结**

本次文章我们以构建特征工程中遇到的一个问题出发，讲解了如何计算一个increasing趋势特征，并引出一个值得思考的问题：大矩阵的转置(存储)。如果有空我们下期推文将研究一下大矩阵的相关问题。另外我们还可以构建其他的特征，如：**每一行大于这行平均值的个数特征等等。**

**以上就是本次文章的全部内容，亲爱的朋友下次再见。**