数据挖掘 第二次作业 实验报告

*13331231*

*孙圣*

*计应2班*

**一、实验环境**

操作系统：Mac OSX 10.11.4

编程语言：Python3

依赖库：numpy, scipy, sklearn

其他辅助库：Intel MKL

注意：由于数据量巨大，因此提交上去的文件夹中包含了strain.txt和stest.txt，都只是很少量的数据，用来验证程序的可执行性。

**二、实验过程**

由于这次实验的数据量巨大，共有2177020个数据，且每个数据都有11392列，同时外加1列label，共有11393列的数据。假设使用占用1 Byte的int8存储，则总共需要：2177020 \* 11393 \* 1 = 24,802,788,860 Bytes，也就是大约为25G的数据。而且这还没有考虑计算过程中引入的占用大小为4 Bytes甚至是8 Bytes的浮点数。25G的数据对于一个普通的PC机来说已经是巨量数据，对于手提电脑来说更是不可能存储下来。

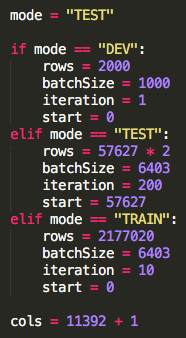
我一开始尝试将这些数据一次读入，使用的是sklearn提供的load\_svmlight\_file()方法：



发现不仅所有的内存被占用，同时Swap的空间也达到了4G，使得整个电脑处于瘫痪状态。因此不得不采用更加灵活的方法来解决这种大规模数据的问题。

于是改用每次读入一部分的数据进行随机梯度下降的方法：

首先，为了编写代码并进行测试，对样本进行部分训练以及对样本整体进行训练，将代码分成3个模式：分别是DEV, TEST, TRAIN。每一个部分所要训练的数据总量都不同，一个batch的大小也不尽相同，迭代的次数和初始的数据点位置也不同：



这样做的好处在于，在不同模式之间切换时，不用频繁的修改参数的值，减少错误与时间开销。

之后就是对数据的读入：

1. 对label的读入：

直接通过numpy提供的genfromtext()方法将所有数据一次性的读入，要注意的是要将reference进行reshape()，不然得到的shape为(row，)而不是(rows, 1)，无法进行训练：



选择一次读入而不是分批读入的原因是：

一次读入总共占用的空间也不大，只有2177020 Bytes，也即2M。但是如果分批读入，不仅造成了多次磁盘I/O浪费时间，而且在训练过程中可能会导致将一部分数据多次读入，同样造成了资源的浪费。

2. 对theta的初始化或者读入：

对于第一次的训练，需要将theta初始化。而如果是之后的训练，就在之前训练的theta的基础上继续训练，因此只需要从文件中读取theta的值即可。

3. 对training data的读入：

首先是一个大的循环：



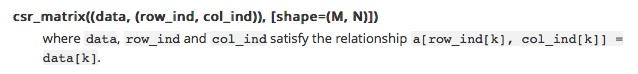
这个循环的作用是：从start位置遍历到rows位置，步长为batchSize，即获得我们要读数据的起始点。

然后就是通过python的文件读写打开文件，之后通过csv.reader对数据进行读入：



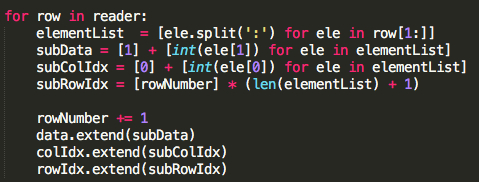
这里其实用到了itertools的islice()方法。使用这个方法的原因是能够快速的读取一个range内的数据，比用python原生的next()跳过每一行要快很多。

对数据的存储主要考虑到了对空间的节省，因此使用了scipy提供的Compressed Sparse Row matrix，即压缩的稀疏行矩阵方法。我所使用的构造函数原型为 csr\_matrix((data, (row\_ind, col\_ind)), [shape=(M, N)])，如图所示：



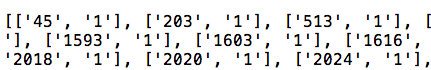
因此要从文件中读取并构造出相应的data，row\_ind，col\_ind的list。

对于每一行的数据，都有以下操作：



1) elementList = [ele.split(':') for ele in row[1:]]

由于每一行的第一个元素都是reference，所以将其忽略，得到row[1:]，然后对于其中的每一个元素，都进行一次split，得到下标和相应的值，保存在一个list中：



2) subData = [1] + [int(ele[1]) for ele in elementList]

这里是获得非0的所有data，一开始的[1]代表着下标为0的情况，根据logistic regression的公式，默认的是第一项为theta0，所以此处应该为1。之后就把ele的第一个元素取出来。其实这里可以进行一个优化，改成：

subData = [1] \* (len(elementList) + 1)

3) subColIdx = [0] + [int(ele[0]) for ele in elementList]

这个和第二步的同理，就是获得非0的列号，保存起来。

4) subRowIdx = [rowNumber] \* (len(elementList) + 1)

这里存储的是行号，对于每一个行，行号都应该相同，所以直接生成len + 1个即可。

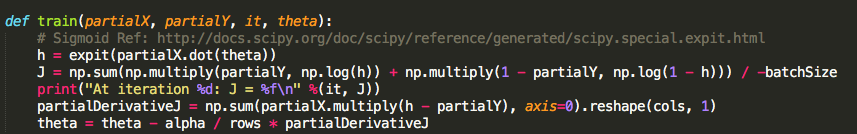
5) 最后将这三组数据extend到整个大的list中。

然后就是调用刚才所说的构造函数，生成一个稀疏矩阵：



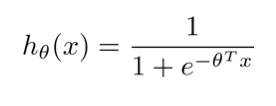
至此，对于一次batch gradient descent所需的数据已经全部读取完毕。

与前面所做的准备工作相比，训练部分就显得十分简单，直接套用LR的公式即可，这里全部采用了矩阵的运算，因此速度比element-wise的计算快很多：



1) h = expit(partialX.dot(theta))

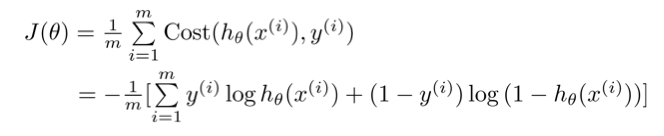
根据公式：



将X与theta点乘，并调用sigmoid函数，计算出h。

2) J = np.sum(np.multiply(partialY, np.log(h)) + np.multiply(1 - partialY, np.log(1 - h))) / -batchSize

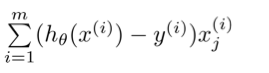
根据公式：



首先，将partialY与log(h)逐元素相乘，再和(1 - partialY)与log(1 - h) 逐元素相乘的结果相加。之后再利用sum()函数将每一行求和，最后除以一个负的batchSize，即可得到J值。

3) partialDerivativeJ = np.sum(partialX.multiply(h - partialY), axis=0).reshape(cols, 1)

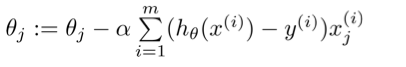
根据公式：



将h与partialY逐元素相减，之后与X的每一列相乘，求和后reshape至cols \* 1的形状。算出对于Xi的梯度：

4) theta = theta - alpha / rows \* partialDerivativeJ

根据公式算出更新的theta：

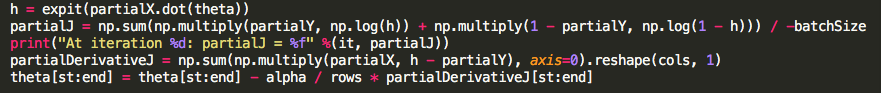


其实这整一部分都和第一次作业的Linear Regression大同小异，所以编码花费的时间不多，唯一要注意的是初始的theta值要设定好，否则会造成第一步求得的h值过于接近0或者1，从而造成第二步取log时溢出的问题。

对于上面的一系列操作，可以使用多线程来做：

将数据分为两个部分，每部分各占一半。因此可以使用两个线程来完成计算。

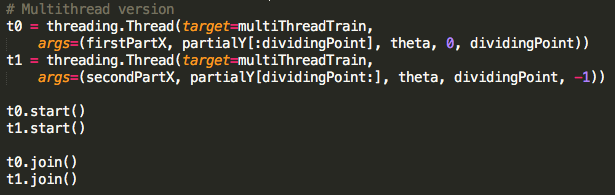
函数部分的代码基本与单线程一致，唯一要注意的是更新theta的时候，只更新start到end之间的那一部分：



多线程主要是用了python提供的threading库，首先将partialX分为两部分，并转换为普通的ndarray：



之后调用Thread()函数，其中target是指所调用的函数，而args是相应的参数：



利用start()函数启动执行各个线程，join()函数保证两个线程都结束运行之后才进入下一个循环，否则会引起theta更新错误，造成训练失败。

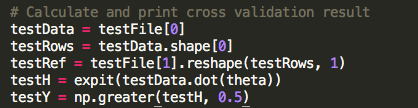
完成了训练之后，就要把theta值写入到文件中，便于下一次利用已有的theta值进行训练，而不用重新训练，这样就能够使得训练过程离散化，不用一直开着电脑。

最后一步就是对结果进行预测，我又将其分为两个部分：

1) 利用已有的数据进行cross validation，检验模型的预测的准确程度，这样就能大概知道模型的表现，而不用一直提交的kaggle的系统上。

首先利用UNIX命令head –numberOfLines test.txt > stest.txt生成cross validation的数据。使用这个命令的好处是：如果直接打开test.txt需要话费一定的时间，而这个命令刹那间就能生成相应的数据。

之后，用load\_svmlight\_file()方法读进cross validation的数据，与前面一样，这里是用稀疏矩阵进行存储的，因此节省了空间开销。然后获取相应的维度信息，例如行数，Reference的值等等：



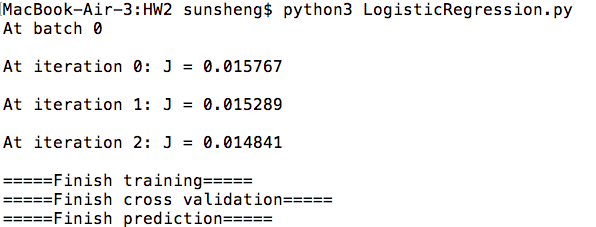
然后将训练数据和之前训练得出的theta进行点乘，再用sigmoid函数求出概率。最后利用numpy中提供的greater()方法，和0.5作为比较，如果所得概率大于0.5，则判定为1，否则，则判定为0。

最后，使用np.count\_nonzero(testRef == testY)，算出正确预测的个数，并输出相应的结果和正确率到testResult.csv中。

2) 输出测试结果。

步骤基本与1相同，唯一要注意的是，此处并没有reference，因此将相应的部分删去即可。

**三、程序的运行截图与优化分析**

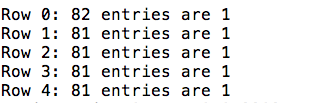
****

**对于内存的优化：**

1. 使用了Batch gradient descent的方法，避免了一次读入所有的训练数据，而是将数据以一个batch读入，避免了对内存的大量占用，同时节省了一开始读取数据的时间。这是因为，让I/O burst和CPU burst交替出现可以使操作系统更好的调度，不至于浪费时间进行忙等待；

2. 使用了compressed sparse row matrix(csr\_matrix)，压缩的稀疏行矩阵，不仅减少了大量的存储空间（只保存为1的列），而且使用行矩阵的方式进行存储而不是列、对角线等其它方式，对于运算速度也有所保证。

如下图：



由此可知，每一行大约有81个元素为1，假设保存所有这些信息需要3 Bytes（行号、列号、数据），那么每一行总共需要81 \* 3 Bytes = 243 Bytes。总共共有2177020行，因此一共需要243 \* 2177020 = 529,015,860 Bytes，约等于530M。这和原来的25G相比，只是原来的百分之二，因此节省了大量的空间。

但是，对于内存的节约总是有一定的代价的。在数据的读取过程中，有许多需要操作Python原生List的部分。众所周知，直接对List的操作其实是很慢的，同时对于稀疏矩阵的运算操作也比numpy中普通的ndarray来的慢很多，毕竟缺少了矩阵的并行化运算。因此是以牺牲部分运算效率的代价来换得内存开销的大幅减少的。

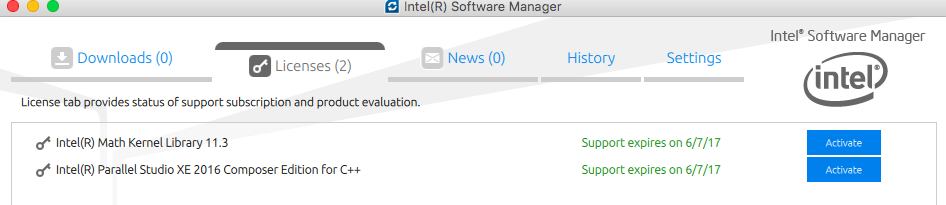
**对于运算速度的优化：**

1. 使用了python库中的itertools.islice()方法，快速的定位所要的数据区间，比直接使用next()方法跳过每一行提高了许多；

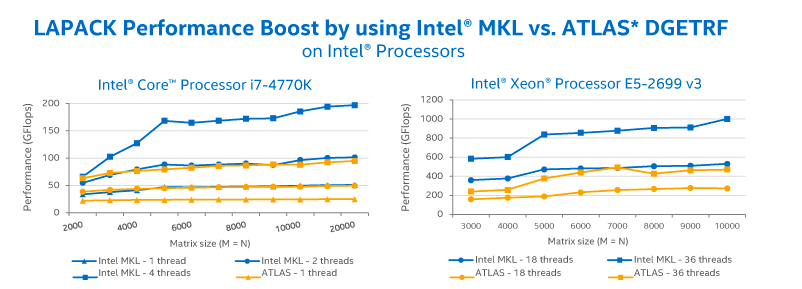
2. 使用numpy中的multiply(), sum(), dot(), log()，以及scipy中的expit()方法（计算sigmoid函数），而不是简单的利用循环进行遍历。因为这些库函数都是向量化的计算，而CPU又对这方面的计算有很多优化，因此速度比非向量化的计算提升不少；

3. 使用了Python中的threading多线程库，将数据分为两个部分，分别来计算J值和更新theta值，这样就利用了现代计算机的多处理器的优势，提升计算效率；

4. 使用Intel MKL(Math Kernal Library)取代了Mac自带的ATLAS作为BLAS库。由于MKL对Intel的处理器有许多计算上的优化，因此矩阵运算速度会快很多：



借鉴网上的benchmark比较[7]，由于有大量I/0时间，因此用系统time命令统计的时间不太准确：



由于MKL需要License（学生可以免费领取一年），而且占用空间较大，约为2G硬盘空间。更加好的方法是使用OpenBLAS，一个开源的矩阵运算库，虽然速度比MKL略低，但是空间占用少，安装也方便，因此更为推荐。

**四、改进空间**

其实如果采用了Batch Gradient Descent，就不必过分拘泥于使用稀疏矩阵来存储数据，因为在读入时的预处理和之后的计算都会带来速度上的减缓。

更好的方法是先用scipy提供的load\_svmlight\_file()方法读入数据，之后立刻用toarray()方法将稀疏矩阵转换为正常的ndarray。然后利用线程池实现的多线程进行并行化运算，得到相应的结果。这样就在内存消耗和计算速度的trade-off之间找到了一个平衡。

同时，还可以尝试使用cuda对运算进行加速。由于我的电脑显卡不是Nvidia的，因此不能使用cuda。不过在和另一个同学合作完成作业3时发现，如果使用CPU\_ONLY的运算，CPU占用率会达到200%（多核心），内存占用会达到4G。而使用CUDA加速后，CPU占用率会下降到75%左右，同时内存开销会降到2G，而且运算速度提升至原来的7倍。因此cuda加速应该是一个相当不错的选择。

参考资料：

[1] Python csv

<https://docs.python.org/3/library/csv.html>

[2] sklearn load\_svmlight\_file

<http://scikit-learn.org/stable/modules/generated/sklearn.datasets.load_svmlight_file.html>

[3] loop through specific range

<http://stackoverflow.com/questions/19031423/how-to-loop-through-specific-range-of-rows-with-python-csv-reader>

[4] itertools

<https://docs.python.org/3.1/library/itertools.html>

[5] scipy csr\_matrix

<http://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csr_matrix.html#scipy.sparse.csr_matrix>

[6] scipy sigmoid function

<http://docs.scipy.org/doc/scipy/reference/generated/scipy.special.expit.html>

[7] Intel MKL benchmark

<https://software.intel.com/en-us/intel-mkl/benchmarks#close>

[8] Python multithread

<https://pymotw.com/2/threading/>

[9] Python multithread waiting all finish

<http://stackoverflow.com/questions/11968689/python-multithreading-wait-till-all-threads-finished>

[10] csr\_matrix slicing

<http://stackoverflow.com/questions/7609108/slicing-sparse-scipy-matrix>

附：

除了这个LR算法之外，我和殷家康，张羽怡，汤子扬同学一起讨论，互相帮助理解之后完成了一个RandomForest算法，我主要负责了其中的数据读取与多线程部分。