2.10 m个样本的梯度下降

在之前的视频中，已经看到如何计算导数和把梯度下降法应用到logistc回归的一个训练样本上，现在我们想要把他应用在m个训练样本上 。首先，时刻记住有关于成本函数J(w,b)的定义，它是这样的一个平均值，m个样本的损失求和的平均值，

………(1)

…………(2)

是训练样本的预测值，上一节说过了如何对单个训练样本求导数，,,表示相应的值，这些都只用到了一个训练样本。

全局成本函数J是一个求和再求平均的函数，损失函数和的平均，它表明全局成本函数对的导数，也同样是各项损失函数对导数的平均。

J(w,b)=，求和符号后面的即是单个样本的，针对于一个具体的样本，将m个训练样本的都求出来，并且求平均，即得到全局梯度值。可以直接运用到梯度下降算法中。接下来我们得到一个总的算法：

Logistic回归在m个训练样本上。

让我们初始化J=0，，，

再用for循环遍历训练集，同时计算相应的每个训练样本的导数，然后把他们加起来:

这里就是循环的结束了，最终对所有的m个样本都进行了这个计算，还需要除以m，因为我们计算平均值。

随着这些计算，你已经计算了损失函数J对各个参数，，和b的导数，在整个计算过程中，我们使用，，作为累加器，所以在这些计算之后，

，即为全局成本函数对的导数，对于和b也一样，同时注意，，没有上标(i)，因为我们在代码中，把他们作为累加器去求取整个训练集上的和，相反的，这是对应于单个训练样本的，上标i指的是对应第i个训练样本的计算，完成这些计算后，应用一步梯度下降，使得获得更新，同理另外两个参数也是，即

最终所有的参数以及J也会是你成本函数的正确值。

![logistc回归 导数计算](<http://upload-images.jianshu.io/upload_images/5355764-7d38266b40fa5ed5.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

这些公式只应用了一次，梯度下降法，因此你需要重复以上的内容很多次，以应用多次梯度下降，因为整个过程执行一次，说明在梯度下降J图中，参数得到了一次更新，但是要到达最低点，还要继续更新，而更新的内容来自于，将更新后的带回原公式中进行计算，z的值，a的值都会改变，此时J的值也会更新，直到最后参数不再更新，J到达最低点。

虽然细节看起来有点复杂，但是在编程的过程中，思路都会变得很清晰，但它表明计算中有两个缺点当应用在这里的时候，就是说引用此方法到logistc回归你需要编写两个for循环，第一个for循环是遍历m个训练样本的小循环，第二个for循环是遍历所有特征的for循环，在这个例子中，我们只有两个特征，所以n等于2，nx也等于2，但如果你有更多的特征，你开始编写你的，，，所以你很需要一个for循环，遍历所有n个特征，当你应用深度学习算法，你会发现在代码中显示的使用for循环会使算法的效率很低，同时在深度学习领域，会有越来越大的数据集，所以能够应用你的算法完全不显示for循环的话是很有用的，可以帮你处理更大的数据集，有一门向量化技术帮助你的代码，摆脱这些显示的for循环。向量化技术有时用来加速运算，但有时候也未必能够，但是在深度学习时代，用向量化来摆脱for循环已经变得相当重要，因为我们开始处理越来越大的数据集，你的代码需要变得非高效。接下来将了解向量化技术，使得在logistc回归中应用梯度下降法而不需要for循环。

2.11 向量化vectorization

什么是向量化：

在logistic回归中，你需要去计算, w是列向量，x也是列向量，如果你有很多的特征，它们就是非常大的向量，所以w和x都是R内的nx维向量，如果使用费向量化的实现方法：

Non-vectorized：

所以这是一个非向量的实现方法，你会发现这是真的很慢，作为对比，一个向量化的实现方法如下：将会非常直接的计算，在python或者numpy中，使用命令就直接计算出来了，再加上b，将会发现这样计算向量非常快。下面是一个实际的例子：

```

*# \_\*\_ coding: utf-8 \_\*\_* **import** numpy **as** np  
a = np.array([1,2,3,4])  
**print** a  
  
*# 完成向量化的例子 引用time模块是为了计算两次不同的操作花费了多少时间***import** time  
*#它能创建一个数组a吗，通过rand函数随机得到，用随机值创建了一个百万维度的数组,b为另外一个百万大小的数组*a = np.random.rand(1000000)  
b = np.random.rand(1000000)  
  
*#现在tic记录一下当前时间*tic = time.time()  
c=np.dot(a,b)  
toc = time.time()  
**print** c  
**print** (**'这是向量化的版本'**+str(1000\*(toc-tic))+**"ms"**)  
  
c=0  
tic = time.time()  
**for** i **in** range(1000000):  
 c+=a[i]\*b[i]  
toc = time.time()  
**print** c  
**print** (**'这是for loop的版本'**+str(1000\*(toc-tic))+**"ms"**)  
  
*#可以发现向量化和非向量化计算出来的结果是一样的，但是最终花费的时间却相差很大*

```

![向量化效果截图：](<http://upload-images.jianshu.io/upload_images/5355764-1349cb926f3a05a4.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

当你正在实现深度学习算法，使用向量化技术真的可以更快得到结果，向量化之后，运行速度会大幅提升，你可能听说过，可扩展深度学习实现是在GPU上做的，GPU也叫图像处理单元，graphics processing unit。我们普通写的代码都是在cpu上运行的，cpu和gpu都有并行化的指令，有时候叫做SIMD指令，意思是指单指令流多数据流，这个词的意思是如果你使用了这样的内置函数，如np.function或者其他能让你显示去掉for循环的函数，这样python的numpy能够充分利用并行化去更快的计算，这点对gpu和cpu上面计算都是成立的。Gpu更加擅长SIMD计算，但是cpu事实上也不是太差，只是没有gpu那么擅长。

经验法则是：只要有其他可能，就不要显式使用for循环。

2.12 向量化vectorization的一些其他例子

（1）如果你想计算一个矩阵和一个向量的乘积，即矩阵乘以列向量，按照矩阵的乘法一样算，得到的仍然是一个列向量。

矩阵乘法的定义，矩阵的一行乘以列向量，对应元素乘积之和

非向量化版计算：

```

u = np.zeros((n,1))

np.zeros这个函数意思是构造相应数据类型的数据，这里是一个n维一列的向量，用0填充  
**for** i **in** ...  
 **for** j **in** ...  
 u[i] += A[i][j]\*v[j]

#这是一个双重for循环，对指标i和j

#而向量化实现：

u = np.dot(A,v)

```

（2）假设你内存中已经有一个向量v，如果你想做指数运算，作用到向量v的每一个元素，那么非向量的做法，先初始化一个全0向量u，然后再用for循环，对v中每一个元素做指数运算再放入u中，一次计算一个元素。但事实上numpy模块中有很多内置函数，可以完成运算，只需要调用一个函数，u=np.exp(v)则完成了。

Numpy库有很多向量值函数，例如np.log(v)会逐个元素计算log值，np.Abs(v)会计算绝对值，np.maximum(v,0)计算所有元素中的最大值，求出v中所有元素和0相比的最大值，v\*\*2就是计算v中每个元素的平方，1/v就是每个元素求倒数等等。

所以每当你想写一个for循环时，应该看看可不可以调用numpy，用内置函数计算，而不是用for循环，接下来看看如何把这些技巧应用到logistc回归梯度下降算法实现中来，看看是否可以去掉两个for循环中的一个，

![logistc回归，两个循环简化为一个循环过程](<http://upload-images.jianshu.io/upload_images/5355764-ceade39b946cb5c4.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

首先初始化的时候，不再对x特征每个分量w都计算对应的导数，而是直接使用,对x特征的每个分量的使用变成直接使用x特征向量。

现在对单个分量使用for循环已经简化了，剩下一个依次处理m个训练样本的循环，还可以写代码处理整个训练样本，基本上同时处理。

2.13 向量化logistc回归

这一节我们将谈及向量化是如何实现在logistc回归上面的，这样就能同时处理整个训练集来实现梯度下降法的一步迭代，针对整个训练集的一步迭代不需要使用任何显式for循环。

首先关于logistc回归的正向传播步骤，如果你有m个训练样本，那么对一个样本进行预测，使用计算z和激活函数两个公式，计算得到第一个，然后继续对第二个训练样本做一个预测，第三个，，，第m个，可以看出，为了执行正向传播步骤，需要对m个训练样本，都计算出预测结果，有一个办法可以不用任何一个显式for循环。方法如下：

![m个样本计算预测值](<http://upload-images.jianshu.io/upload_images/5355764-b97176b76e214fab.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

曾经定义过一个矩阵X来作为你的训练输入，即m个样本的特征向量组成的，在不同的列中堆叠而成的一个矩阵，,这是一个的矩阵，关于python中numpy的形式为（）。首先我们探究如何计算,将这些计算全都包含在一个步骤中，事实上仅用了一行代码，首先要构造一个1\*m的矩阵，实际上是一个行向量，用来保存计算出来的z的值，，都是在同一时间计算出，可以发现：

其中就是一个1\*m的矩阵，或者说是一个m维的行向量，其中w是一个维的列向量，转置成行向量以后可以和X做乘积运算，乘以X的每一列最后加上b得到一个一维的行向量即Z。

X是把所有的训练样本堆叠起来得到的，一个挨着一个，横向堆叠，Z则是一个一维行向量，使用每个计算出来的横向排在一起，最后为了计算Z，直接使用numpy的指令：

在这里b是一个实数，或者说是一个1\*1的矩阵，就是一个普通的实数，但是当向量加上这个实数时，pyhton会自动的把b这个实数扩展成一个1\*m的向量，在python中这叫做广播。

接下来就是使用激活函数，得到,把所有的堆叠起来，就得到了， sigmoid函数将Z作为参数输入，输出结果为A。

总的来说，不需要for循环就可以从m个训练样本一次性计算出所有样本的z值和a预测值，只需要运行两行代码就可以高效计算出结果，以上就是正向传播一步迭代的向量化实现，同时处理m个训练样本，接下来你会发现使用向量化也可以高效的计算反向传播过程，计算出梯度。

2.14 向量化logistc回归的梯度输出

如何使用向量化计算m个训练数据的梯度，注意是同时计算，最后得到一个非常高效的logistc回归的实现。

在计算梯度时，我们需要计算：

…

我们现在可以定义一个新的变量，,所有的dz横向排列，最后的也会是一个1\*m的矩阵，并且在这之前我们定义了

基于这些定义，可以得到，仅需要一行代码就可以同时完成以上的所有dz的计算，

同时，在前面我们已经解决了遍历的一个训练样本中的每个分量,来计算dw，直接使用,对x特征的每个分量的使用变成直接使用x特征向量，但是最后累加m个样本的，仍然还存在整体样本的循环，并且的计算也一样，将m个样本的db累加，我们已经去掉了一个for循环，至少现在是个向量了，接下来我们如何使用向量化将在m个训练集中的for循环去掉。

其中的计算，就是将每个样本的加起来除以m，因此，所有的组成了一个行向量，所以在python里只需要，关于的计算，

…

使用向量的计算的话，

根据矩阵乘法的定义，最后得到的结果X的每一行乘以一列,最后得到一个一维列向量，列向量每一个元素都是针对不同分量，最后再除以m，则得到。从一开始到最后都是关于参数的向量，最后也仍然是一个一维列向量，x的第一个分量与dz的乘积，再对m个训练样本的第一个分量计算出来的结果求和得到最后，所以刚好就是X矩阵的第一行与dz相乘构成的每个分量。

因此最后使用红色的两个公式便能计算出反向传播过程的参数导数，得到和。注意我们并没有在训练集上使用for循环，现在就可以计算参数的更新了。

现在回顾之前的计算logistc回归的整个过程，没有向量化非常低效，而没有使用任何for循环，最后代码计算只需要几步就完成了：

这就完成了正向传播和反向传播，确实实现了对所有样本进行预测和求导，而且没有使用任何一个for循环，然后梯度下降更新参数：

有了这些我们就实现了logistc回归的梯度下降一次迭代，虽然说过尽量不要使用for循环，但是如果想要实现多次迭代，仍然需要使用for循环，在上面整个一次迭代过程中加上循环次数，应该没有方式能把这个for循环去掉。

高度向量化的非常高效的logistc回归的梯度下降法。

2.15 python中的广播

广播是一种手段，可以让你的python代码段执行的更快，我们将继续深入研究python中的广播是如何实际运作的。

```

*# \_\*\_ coding: utf-8 \_\*\_***import** numpy **as** np  
  
A = np.array([[56.0,0.0,4.4,68.0],  
 [1.2,104.0,52.0,8.0],  
 [1.8,135.0,99.0,0.9]])  
**print** A  
  
cal = A.sum(axis=0)  
*# axis=0意味着竖直相加，水平轴是axis=1***print** cal  
  
percentage = 100\* A/cal.reshape(1,4)  
*#使用矩阵A除以这个1\*4的矩阵，然后得到了百分比矩阵，***print** percentage  
  
*#这个例子就是一个广播的例子，在求percentage时，用3\*4的矩阵除以一个1\*4的矩阵，这里在技术上使用reshape是多余的，但是通常使用reshape来确保参与运算的是  
#我们想要的形状的矩阵。这里是o(1)操作，成本很低。所以不要害怕使用reshape命令来确保你的矩阵形状是你想要的。  
#这种运算是怎么执行的？3\*4矩阵是如何除1\*4矩阵的呢。*

```

运行结果：

[[ 56. 0. 4.4 68. ]

[ 1.2 104. 52. 8. ]

[ 1.8 135. 99. 0.9]]

[ 59. 239. 155.4 76.9]

[[ 94.91525424 0. 2.83140283 88.42652796]

[ 2.03389831 43.51464435 33.46203346 10.40312094]

[ 3.05084746 56.48535565 63.70656371 1.17035111]]

关于广播，一个(m,n)矩阵和一个(1,n)矩阵相加，首先pyhton会把(1,n)的矩阵复制m次，编程(m,n)矩阵，然后两个矩阵相加。同理，若一个(m,n)与一个(m,1),则会复制n次列向量。

广播通用规则，对于一个(m,n)矩阵，加减乘除一个(1,n)的矩阵，后者都会复制m次变成(m,n)矩阵，同理，若为(m,1)的矩阵，则会复制n次列。注意这里的加减乘除都是逐个元素对应操作，不是根据矩阵的乘法原则。还有另一种广播操作，就是一维向量与实数的操作，同理。以上是在神经网络中用到的广播形式，还有更多参看numpy文档。

2.16 关于python/numpy向量的说明

Python让你能够使用广播运算，一般来说，pyhton numpy程序语言给你提供了很高的灵活性，这算是一门编程语言的优势，同时也是劣势，优势是因为它让语言的表现力更强，语言的灵活性很大，也就是说，你可以用一行代码完成很多运算，弱点就是因为广播和这么大的灵活性，有时可能会引入非常细微的错误，非常奇怪的bug，如果你不熟悉所有复杂的广播运作方式，比如你想用列向量把它加到一个横向量上，你可能会预计它会报错，说维度不匹配，或者类型错误之类的，但事实上你会得到一个行向量和一个列向量求和后的矩阵。

Python的这些奇怪的效果有其内在逻辑，但是如果你不熟悉的话，可能会产生很多奇怪的难以调试的错误。

```

*# \_\*\_ coding: utf-8 \_\*\_***import** numpy **as** np  
  
a = np.random.randn(5)  
*#生成5个随机的高斯变量，存储在数组a中，  
#这创建了一个数据结构，a.shape=（5，）就可以查看到这个数据结构，叫做秩为1的数组，这个数据结构与行向量和列向量并不一样  
#***print** a  
**print** a.shape  
*#这就是所谓的pyhton秩为1的数组，它既不是行向量也不是列向量***print** a.T  
**print** np.dot(a,a.T)  
  
  
a=np.random.randn(5,1)  
**print** a  
**print** a.T  
**print** np.dot(a,a.T)

```

![秩为1数组和列向量区别：运行结果](<http://upload-images.jianshu.io/upload_images/5355764-d8f220a8f2c660fc.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

当你进行编程联系时，或者实现神经网络的logistc回归时，就不要使用这些秩为1的数组，相反每次创建数组时，你要把它定义成列向量，或者变成一个行向量，那么你向量的行为就更容易理解一些。代码如下：

```

a = np.random.randn(5) *#创建了一个秩为1的数组，不使用这个*a = np.random.randn(5,1) *#列向量，5行1列 a.shape = (5,1)*a = np.random.randn(1,5) *#创建一个行向量，5行1列*

```

另外，当你在代码中做了很多，不清楚一个向量具体的维度是多少，经常使用assert（）这样一个声明，确保这是一个向量，这些assert()语句执行起来很快，也可以看成是代码的文档。最后，如果你因为一些原因，得到了秩为1 的数组，你可以使用reshape命令，a=a.reshape转化成一个向量。那么它的行为更好预测，就是列向量或者行向量的行为。

有时候会看到一些很难调试的错误，都来自秩为1的反直觉行为，通过消除代码中秩为1的数组，可以让代码更简单。所以在代码中，尽量使用n\*1矩阵，基本上是列向量，或1\*n矩阵，基本是行向量，随意插入assert（）声明，要仔细检查你的矩阵和数组的维度。不要害怕调用reshape来保证你的数组或向量是你想要的维度。

2.18 logistc回归损失函数解释

这节中，将给出关于成本函数的简洁的证明。

公式集合：



我们约定，这里的即是算法输出的预测值。表示的是在给定训练样本x的条件下，y为1（在二分分类问题里，即表示是的概率）的概率。换句话说：

总结一下，就是

即在给定x情况下，这个训练样本的结果可能为1，可能为0，本来是有类标的，y表示类标，根据类标可以决定计算出来这个训练样本预测值为类别的概率。如果概率越大，说明预测结果越准确，概率越小，则说明预测结果偏差大。

将两个公式，合并，用一个公式进行表示，得到结果如下：

将y的可能情况带入，发现符合判断概率的公式。即这个表达式，

包含了上面的两个条件概率公式。

由于log函数是严格递增的函数，最大化即相当于最大化P(y|x)，计算，化简得到：

这个公式就是我们前面提到的损失函数L的负值，前面有一个负号的原因是，当你训练学习算法时，希望算法输出的概率值是最大的，想得到的是最大概率，即最小化损失函数，那么对损失函数而言，最小损失值即最大概率，预测结果最相近，符合逻辑。因此，这就是单个训练样本的损失函数表达式。

那么成本函数呢，m个训练样本的总体成本函数如何表示？

整个训练集中标签的概率，假设所有的训练样本是服从同一分布并且相互独立的，也即是独立且同分布的，所有这些样本的联合概率，就是每个样本概率的乘积。如果你想做最大似然估计，需要寻找一组参数，使得给定样本的观测值的概率最大，令这个概率最大化，等价于令其对数最大化，在两边同时加上log符号，在等式两边取对数，训练集的标签出现概率的对数为

求和符号后面的部分即为-L，即损失函数的负值。在统计学里，有一个方法，叫做最大似然估计，maximum likelihood estimation。即求出一组参数，使这个式子取最大值，那么就是去掉富豪后面部分的最小值，这样我们就推导出了前面给出的logistc回归的成本函数J(w,b)：

由于训练模型时，目标是让成本函数最小化，所以我们不用最大似然估计，去掉这里的负号，并且可以进行相应的缩放，所以前面乘以一个常熟因子。

第一章的补充内容：

一点是，如果你想达到很高的性能水平，有两个条件，一个是需要训练一个规模足够大的神经网络以发挥数据规模量巨大的优点，另外要达到横轴的右端位置，需要很多的数据，因此我们经常说，规模一直在推动深度学习的进步，我们需要有一个有很多隐藏单元的神经网络。有许多的参数，许多的连接，而且还有数据规模，事实上，要在神经网络上获得更好的表现，最可靠的手段要么是训练一个更大的神经网络，要么投入更多的数据。但这只是在一定程度上，因为你的数据达到一定程度，神经网络规模太大，需要的训练时间太久。

第二章作业笔记：

1.4标准化行

另一个在深度学习和机器学习的技巧是标准化数据，因为梯度下降收敛的更快在标准化之后，这使得有更好的性能。这里的标准化通常是指把x变成，每个行向量除以它们的标准值，即所有元素的平方和开方。这里的除法是用到了广播技术的，将分母的2\*1按列复制3列，元素对应进行除法操作。

例如：

np.linalg.norm(a,ord=None,axis = 1)    #计算矩阵a的范数,axis表示按行计算，=0表示按列计算

1.4广播以及softmax函数

sigmoid函数只能分两类，而softmax能分多类，softmax是sigmoid的扩展。

Sigmoid函数对传入的z进行运算，最后得到为0或者为1的概率。Softmax的算法公式为

![softmax函数](<http://upload-images.jianshu.io/upload_images/5355764-8989ec248cd3408b.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

![softmax函数公式详细](<http://upload-images.jianshu.io/upload_images/5355764-3b6e75f382cc3118.png?imageMogr2/auto-orient/strip%7CimageView2/2/w/1240>)

numpy.dot()有两种意思：参数为两个列表时，为求两个列表的点积，即对应相乘再加和

参数为向量与矩阵或者矩阵时，则是做矩阵的乘法

参数一个为m\*n的矩阵，一个为n个元素的列表时，就结果为矩阵的每一行对应乘以列表，最后加和，每一行都是这样，最后得到一个m行的列向量。

```

import numpy as np

x1 = np.array([

[9, 2, 5],

[7, 5, 0]])

x2 = [1,2,3]

toc = np.dot(x1,x2)

print (str(toc))

```

运行结果：[28 17]

关于python的数组，列表，和矩阵之间的联系：

在python基本类型里只有列表，声明方法为，在中括号外面加括号之类的都不正确，并且列表没有shape属性。要将列表转化为向量或者矩阵，都需要调用numpy的方法来完成。例如

```

x = [1,2,3,4]

x = np.array(x)

print (x.shape)

#结果则为：(4,)

但是此时np.dot(x,x)结果仍然是对应元素的乘积加和

```

abs()与np.abs()区别：这两个函数的意思都是取绝对值，但是作用的对象不一样，abs（）可以对实数取绝对值，数组也可以，但是对向量就不可以，而np.abs（）还可以对列表进行处理。