二分分类

我们将数据的标签分为0和1(即有和没有),这样的分类就是二分分类

图像数据在计算机中的储存

图像数据在计算机中通常按照像素大小被储存为三个大小相同的二维矩阵,分别记录图像中每个像素的红,绿,蓝三种颜色的颜色强度.

数据的输入

在输入数据时,通常会将三个矩阵中的矩阵中的所有数据输入到一个特征向量x中,向量的维数是矩阵中的数据总数
数据则是由向量x和其标签y组成的,训练集是由若干个这样的数据组成的
我们将训练集中的所有特征向量x作为列向量或者行向量依次合并为一个大的矩阵 ***X*** ,将所有标签y依次合并为一个行向量 ***Y*** ,(使用函数***X.shape***或者***Y.shape***可以输出X和Y的维度)

logistic回归

logistic回归是一种学习算法,用在监督学习问题中,并且输出的标签为0和1(也就是二元分类)时

函数的选择

输入一张图片,特征向量x则是一张图片,我们需要识别这张图片是否含有猫,则需要一个算法给出预测值y,即这张图含有猫的概率.

~~线性回归函数~~$y = w^T x + b$~~:~~ 实际上并不是一个很好的算法,因为y的范围应当在0到1之间

所以,***sigmoid函数***是一个很好的选择,可以将y的值映射到0到1之间,那么就可以将线性回归函数整个作为自变量带入到**sigmoid函数**中,即$y = f(z) = \frac{1}{1+e^{-z}} (z = w^T x + b)$,当z无限大时，$e^{-z}$无限趋近于零，则y整体会趋近于1，反之则趋近于0，不会有超出0和1范围的风险

参数w与b

在进行神经网络编程时，我们通常会把w与参数b分开。一些人会额外设置一个$x_0 = 1$，所以矩阵***X***是一个n + 1维的矩阵，此时$y = \sigma\ (\theta^T x)$ ，在这种约定中，存在一个向量$\theta = [\theta_0, \theta_1, ...\theta_n]$,$x_0$对应的$\theta_0$就起到了b的作用，而其余的参数则起到了w的作用,所以一般来讲我们会将w与b看做独立的参数

logistic回归损失函数

为了训练logistic回归函数的参数w以及b，需要定义一个成本函数.而回归损失函数，可以用于衡量算法的运行情况

什么是损失

从数学上讲，损失可以定义为$L(\hat{y}, y) = (\hat{y} - y)^2$或者$\frac{1}{2}(\hat{y} - y)^2$
但是在logistic回归中,通常并不会这么做,因为在之后的优化问题中,多数是非凸的,最后得到的结果是很多个局部最优解

误差平方看似合理,但是在梯度下降法中不太好用,所以在logistic回归中,我们会定义一个不同的损失函数,起着与误差平方相似的作用,可以为我们提供一个凸的优化问题,很容易去优化

$$L(\hat{y}, y) = -(y \ln \hat{y}) + (1 - y) \ln (1 - \hat{y})$$

对于这个**logistic回归的损失函数**,我们依然想让它**尽可能地小**

所以接下来会举例存在的两种情况

如果$y = 1$,则会有$L(\hat{y}, y) = -\ln \hat{y}$
也就是说$-\ln \hat{y}$应当**足够小**,也就是$\hat{y}$应当足够大,但是由于**sigmoid函数**的作用,$\hat{y}$不可能大于1,所以$\hat{y}$应当尽可能趋近于1

同理,如果$y = 0$,则会有$L(\hat{y}, y) = -\ln (1 - \hat{y})$
$\hat{y}$就应当尽可能趋近于0

成本函数

损失函数是在**单个训练样本**中定义的,它衡量了在单个训练样本上的表现.但是如果要衡量在**全体训练样本**上的表现,则需要定义一个***成本函数***

$$J(w,b)=\frac{1}{m}\sum_{i=1}^{m}L(\hat{y}^{(i)},y^{(i)})\quad(\hat{y}\text{是利用}w\text{和}b\text{通过}logistic\text{回归算法得出的预测值})$$

所以我们需要找到合适的参数w和b使得$J(w,b)$尽可能小

梯度下降法

目的

找到使得成本函数$J(w,b)$最小的参数w和b

方法

首先我们应当选择一个值对w,b进行初始化

我们很容易看到,使用**上一章所说的成本函数**进行计算的成本函数$J(w,b)$是一个凸函数,所以无论选择那个一点进行初始化实际上都是一样的,最终都会到达某一个特定的点,或者十分接近这个点的附近 通常选取0作为初始值

过程 :从初始点开始,最陡的下坡方向(也就是梯度最大的一个方向走)走一步,称为一次迭代,在若干次迭代后,w,b的值会到达或者无限接近于一个全局最优解

我们可以先从二维视角来观察,取wOJ面,假设一次迭代的结果为

$$w'=w-\alpha\frac{dJ(w)}{dw}\quad(\alpha\text{为学习率,可以控制每一次迭代的步长})$$

在算法收敛前，不断重复地进行这一个过程,最终达到或无限接近全局最优解

但实际上成本函数含有w,b两个参数,所以需要不断地通过

$$w'=w-\alpha\frac{dJ(w,b)}{dw}$$

$$b^{'} = b - \alpha\frac{dJ(w,b)}{db}$$

这个过程来进行更新

计算图

神经网络的计算

一个**神经网络**的的计算,都是按照前向或反向传播过程来实现的.

首先,计算出神经网络的输出,紧接着进行一个反向传输的操作,我们利用**后者**这一过程来计算出对应的导数.

举例

计算函数$J(a,b,c) = 3(a + bc)$
计算这一函数实际上有三个过程:

首先,计算bc,并将结果储存在变量u之中,然后计算$v = a + u$,最后输出$J = 3v$

所以,通常来说,反向传输的过程通常是一个链式过程

当我们有一些不同或者特殊的输出变量时,也就是我们想要优化的变量时,在logistic回归中$J(w,b)$是我们想要最小化的成本函数,计算这个值的过程是以上举例类型的正向的过程,而通过这个输出计算导数时,则是一个反向的过程,与上述过程步骤相反

使用计算图求导

过程

在求到的过程中,我们通常会稍微通过改变某一个变量的值,观察输出的变化值的方法来计算导数,在上一章的例子中

$$u = bc$$

$$v = a + u$$

$$J = 3v$$

现在假设

$$a = 5$$

$$b = 3$$

$$c = 2$$

那么

$$u = 6$$

$$v = 11$$

$$J = 33$$

如果我们稍微改变v的值，假设将v从11变为11.001,那么J就会从33变为33.003,v的增量为0.001,J的增量为0.003,所以J对v的导数为3
同理,假设a从5变为5.001,v会变为11.001,J就会变为33.003,同理J对a的导数为3,v对a的导数为1
我们可以看出,J对a的导数可以这样表示

$$\frac{dJ}{dv}.\frac{dv}{da}$$

同理,J对b和c的导数,也可以通过

$$\frac{dJ}{db} = \frac{dJ}{du}.\frac{du}{db}$$

以及

$$\frac{dJ}{dc} = \frac{dJ}{du}.\frac{du}{dc}$$

来表示

当我们在软件里实现这一过程的时候,在**python**中,对于书面上

$$\frac{dOutPutVar}{dVar}$$

的形式,可以使用"$dVar$"来表示最终变量J(也可以是各种中间变量)的导数

logistic回归中的梯度下降法

计算流程

根据公式

$$z = w^T \vec{x}$$

现在假设样本只有两个参数x_1, x_2
为了计算z,将参数带入,则有$z = w_1 x_1 + w_2 x_2 + b$
那么根据下一步公式,可以得到$\hat{y} = a = \sigma(z)$
最后计算得出$L(a, y) = -(y \ln a + (1 - y) \ln (1 - a))\backslash$

在**logistic回归**中，我们需要做的就是变换参数w, b的值,来最小化损失函数$J(w, b)$

现在,我们需要先计算出损失函数的导数,并且很容易求出

$$\frac{dL(a,y)}{da} = -\frac{y}{a} + \frac{1-y}{1-a}$$

并且也可以计算

$$\frac{dL(a,y)}{dz} = \frac{dL(a,y)}{da} \cdot \frac{da}{dz} = (-\frac{y}{a} + \frac{1-y}{1-a}).a(1-a) = a - y$$

最后,我们需要计算关于w, b的导数

$$\frac{dL}{dw_1} = \frac{dL}{dz} \cdot \frac{dz}{dw_1} = x_1(a - y)$$

同理

$$\frac{dL}{dw_2} = x_2(a - y)$$

$$\frac{dL}{db} = a - y$$

因此可以得到迭代步骤

$$w_1^{'} = w_1 - \alpha \frac{dL}{dw_1}$$

$$w_2^{'} = w_2 - \alpha \frac{dL}{dw_2}$$

$$b^{'} = b - \alpha \frac{dL}{db}$$

*这只是针对一个样本的计算，实际上，训练模型应当使用有**m**个训练样本的整个训练集*

针对m个样本的梯度下降

我们仍然假设样本只有两个参数
首先列出损失函数

$$J(w,b) = -\frac{1}{m}\sum_{i=1}^{m} L(a^{(i)}, y)$$

其次可以得到

$$a^{(i)} = \hat{y}^{(i)} = \sigma(z^{(i)}) = \sigma(w^T \vec{x}^{(i)} + b)$$

所以可以得出

$$\frac{\partial J(w,b)}{\partial w_1} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y)}{\partial w_1^{(i)}} = \frac{1}{m}\sum_{i=1}^{m} x_1^{(i)} dz^{(i)}$$

$$\frac{\partial J(w,b)}{\partial w_2} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y)}{\partial w_2^{(i)}} = \frac{1}{m}\sum_{i=1}^{m} x_2^{(i)} dz^{(i)}$$

$$\frac{\partial J(w,b)}{\partial b} = \frac{1}{m}\sum_{i=1}^{m} \frac{\partial L(a^{(i)}, y)}{\partial b^{(i)}} = \frac{1}{m}\sum_{i=1}^{m} dz^{(i)}$$

所以我们现在对参数进行初始化

$$J = 0, dw_1 = 0, dw_2 = 0, db = 0$$

现在,使用**for循环**从1到m遍历训练集,同时计算相应的每个训练样本的导数,分别求和
遍历完成后再将得到的和分别除以m,就计算出了损失函数$J(w,b)$以及对各个参数的偏导数

使用得到的导数对各个参数进行一次迭代,得到新的参数,再使用新的参数重复上述过程,进行若干次直到损失函数收敛

向量化

什么是向量化

以logistic回归中,$z = w^T\vec{x} + b$为例

w, x都是n维列向量,所以在计算中,如果使用了非向量化的方法,以**python**为例,就会有

```
z = 0
for i in range(n)
    z += w[i]*x[i]

z +=b
```

这样的计算效率较低，而向量化的计算就非常直接
你可以使用如下命令

```
import numpy as np
z = np.dot(w,x) +b
```

经过检验，可以发现向量化的计算速度快于循环体遍历，且消耗时间是循环体的接近三百分之一

所以得出经验，*只要有其他可能，最好不要使用显式的**for**循环*

向量化的更多例子

一些经验性的法则

- 在编写新的神经网络或者进行回归时，尽量避免for循环

举例

向量$\vec{u} = A\vec{v}$的计算相当于

$$u_i = \sum_{j=1}^{n} A_{ij} v_i$$

如果使用**for循环**,就会产生两个嵌套

```
for i in range(n)
    for j in range(n)
        u[i] += A[i][j]*v[j]
```

显然这样的运行效率比较低下
但如果我们选择向量化计算

```
u = np.dot(A,v)
```

很明显代码更加简洁，运行效率也更加高效

假设现在内存中有一个向量$\vec{v}$,并且对其进行指数运算,就像

$$u_i = e^{v_i}$$

对于非向量化的运算,就有

```
u = np.zeros((n,1))
for i in range(n)
    u[i] = math.exp(v[i])
```

而向量化的实现

```
u = np.exp(v)
```

还有很多例子,并且这些运算都是对于向量$\vec{v}$中的每一个元素进行运算，比如

```
np.log(v)
np.abs(v)
np.maximum(v,0)
```

```
v**2
1/v
```

所以，每当我们想要使用**for循环**时，应当检查能不能使用numpy的内置函数进行计算

针对于logistic回归

首先根据前面的章节推导出的过程,我们可以知道
我们在进行梯度下降时会有两个循环

我们先简化掉**第二个for循环**也就是针对于每个样本中所有参数分别相加的循环

```
dw = np.zeros((n,1))
dw += x[i]*dz[i]
dw /= m
```

这样，我们的代码就从两个for循环简化为了一个for循环

向量化logistic回归

正向传播

首先,我们需要进行logistic回归的正向传播步骤 对第一个样本进行预测,运用公式,计算出该样本的$z,\hat{y}$,以此类推,直到将所有样本计算完成

在这个过程中,我们甚至也可以将**for循环**简化

首先给出训练集矩阵$X_{n\times m}$以及参数矩阵W 构建行向量$\vec{z},\vec{b}$

```
z = np.dot(W,X) + b
```

然后构建向量$\vec{a}$

```
a = 1/(np.exp(-z) + 1)
```

所以在对logistic回归的正向传播中，我们也将计算进行了向量化，可以通过这一步骤，省略**for循环**，同时计算所有数据

向量化logistic回归的梯度输出

导数的向量化计算

与前面所讲的基本类似,我们使用向量来储存导数的值
根据前面部分所讲,我们可以得出导数的计算方法

```
db = np.sum(dz)/m
dw = np.dot(X,dz.T)
```

消去for循环

经过前面的推导，之前繁琐低效循环的代码就可以被简化为如下的代码

```
Z = np.dot(w.T,X) + b
A = 1/(np.exp(-Z))
dz = A - Y
dw = np.dot(X,dz.T)/m
db = np.sum(dz)/m
```

这样的代码简洁高效，并且我们已经消掉了所有的**for循环**

在这之后，我们就可以根据得到的数据进行迭代

$$w^{'} = w - \alpha dw$$

$$b^{'} = b - \alpha db$$

自此,我们完成了一次迭代
但是,如果要进行多次迭代,我们仍然需要使用**for循环**,这个循环大概没有办法去掉

python中的广播

什么是广播

在python中,广播是一种手段,可以让代码段执行的更快

举例

这里有一个矩阵

	Apples	Beef	Eggs	Potatoes
Carb	56.0	0.0	4.4	68.0
Protein	1.2	104.0	52.0	8.0
Fat	1.8	135.0	99.0	0.9

列出了四种不同食物中，每100克食物所含的碳水，蛋白质和脂肪量的卡路里大小
现在我们需要就算四种十五中,卡路里所占的百分比

所以我们要做的就是对矩阵的每一列求和,并且让每个数据除以该列的总和数

那么我们可以**不通过for循环**来得到这一结果吗?
我们可以写出下面的代码

```
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]])
cal = A.sum(axis=0)
percantage = 100*A/cal.reshape(1,4)
```

在这段代码中，我们通过`sum(axis=0)`的函数对A的每一列进行了求和，并将求和结果储存在cal中，再使用`A/cal`,使得A中的每一个元素都被气所在列的和相除,于是得出了结果
这就是广播的作用

其他的一些例子

如果我们将一个四维的列向量与100相加,那么广播就会自动将100扩展为一个四个元素都为100的四维列向量,然后与之前的列向量进行一个加法

如果我们把一个(m,n)的矩阵与一个(1,n)的矩阵相加,那么广播就会把(1,n)的矩阵复制m行,再与前者相加

总结

所以我们对一个矩阵----这里暂且假设为(m,n)的矩阵----用一个(1,n)或者(m,1)的矩阵进行加减乘除中的任何操作,python都会将其自动复制补全,然后对其中的每一个元素进行操作

消除bug的小技巧

我们先创建一个向量

```
a = np.random.randn(5)
a.shape = (5,)
```

如果这样做，实际上我们得到的**并不是一个向量**，或者说，numpy并不认为这是一个向量，而只是一个秩为1的数组。所以我们在创建时，就应当先规定好这是一个怎样的向量或矩阵

```
a = np.random.randn(5,1)# or a = np.random.randn(1,5)
```