机器学习——Logistic回归

回归：

常见的回归是指我们给定一些数据点，用于一条直线的拟合，在得到这条直线后，当有新的输入时，可以得到它的输出，相较于普通的线性回归（用于连续变量的预测），Logistics主要用于离散变量的分类，它的输出范围是一个离散的集合，表示属于某一类的概率。总的来说，Logistics回归进行分类的主要思想是：根据现有数据对分类边界线建立回归，以此进行分类，本质上是一个基于条件概率的判别模型。

推导：

对于Logistics回归，一个重要的Sigmoid函数需要了解：

$$h_\theta(x) = g(\theta^T x) = \frac{1}{1+e^{-\theta^T x}}$$

对于这个函数的理解：θ是一个参数列向量，这是我们需要求解的拟合参数，x是样本列向量，即我们给定的数据集，函数g（）实现了任意函数到[0,1]的映射，这样无论数据集多大，都可以映射到01区间进行分类，而我们通过这种回归方法得到的输出就是分类为1的概率。

对于这种回归方法，最重要的就是求定这个参数向量θ。

代价函数：

怎么求解这个参数向量呢？我记得参数估计这本书中讲了很多对于参数向量的求解，感兴趣的人可以去看一下，如果我没有记错的话可以有皮尔森、最大似然、贝叶斯风险、最大最小准则等等，我们这里其实就是从代价函数开始考虑的，给定一下函数大家理解一下：

$$Cost(h_\theta(x), y) = h_\theta(x)^y(1-h_\theta(x)^{1-y})$$

对于这个函数我们称之为代价函数，当y=1时，右式子的第二项为1，这时函数表示我们分类得到为1的概率；而当y=0时，右式子的第一项为1，这时表示我们分类得到为0的概率，也就是说我们的输入分类是什么，我们就得到它对应的概率，基于这个基础，综合起来我们就是为了在给定样本情况下，让这个函数越大，我们的拟合越准确，这时我们的问题就是怎么让上面的代价式子取到最大值，如果你熟悉最大似然，那么就容易了，对上式子取对数，然后求让函数最大值的参数变量，经过对数变化之后：

$$J(\theta) = \sum_1^m [y^{(i)} log(h_\theta(x^{(i)})) + (1-y^{(i)}) log(1-h_\theta(x^{(i)}))]$$

梯度上升法：

接下来就是求解使得上式取到最大值的参数变量，我们这里介绍一种算法：梯度上升算法，所谓梯度上升法，就是为了求到某函数的最大值，沿着该函数的梯度方向搜寻，就像爬坡一样，一点一点逼近极值，爬坡的工作数学方式就是：

$$x_{i+1} = x_i + \alpha \frac{\partial f(x_i)}{x_i}$$

其中α为步长，就是我们常说的学习速率，控制更新的幅度。

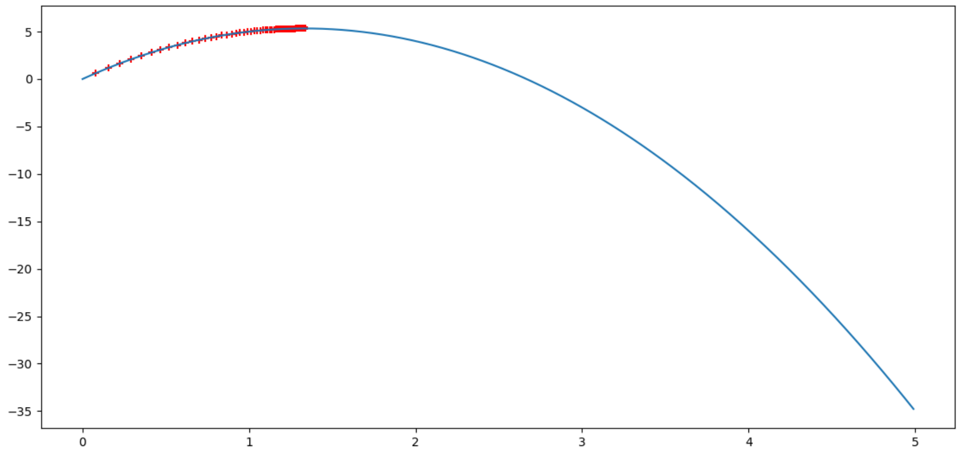
例如对于函数$f(x) = -3x^2 + 8x$，我们可以有以下的梯度上升法求得它的最大值：

```
import numpy as np
```

```
import matplotlib.pyplot as plt
x = np.arange(0,5,0.01)
y = -3*x**2+8*x
plt.figure(1)
plt.plot(x,y)

def Gradient_As():
    #求导函数，根据不同函数可以写出不同的求导函数
    def derivation(x_old):
        return -6*x_old+8
    #设定初始值、步长、精度
    x_old=-2
    x_new=0
    alpha=0.01
    presision=0.00000001
    #爬坡
    while abs(x_new-x_old)>presision:
        x_old=x_new
        x_new=x_old+alpha*derivation(x_old)
        plt.figure(1)
        plt.scatter(x_new,-3*x_new**2+8*x_new,color='r',marker='+')
    plt.show()
    print(x_new)

Gradient_As()
```

> 1.333333177763651

可以看到我们得到的梯度上升过程就是逼近极值的过程：



利用这个梯度上升法对上面的对数式子进行求解：

$$\theta_j := \theta_j + \alpha \frac{\partial J(\theta)}{\theta_j}$$

$$J(\theta) = \sum_{1}^{m} [y^{(i)} log(h_\theta(x^{(i)})) + (1 - y^{(i)}) log(1 - h_\theta(x^{(i)}))]$$

$$\frac{\partial}{\theta_j} J(\theta) = \frac{\partial J(\theta)}{\partial g(\theta^T x)} * \frac{\partial g(\theta^T x)}{\partial \theta^T x} * \frac{\partial \theta^T x}{\partial \theta_j}$$

$$\frac{\partial J(\theta)}{\partial g(\theta^T x)} = y * \frac{1}{g(\theta^T x)} + (y-1) * \frac{1}{1-g(\theta^T x)}$$

经过运算后：

$$\theta_j := \theta_j + \alpha \sum_1^m (y^{(i)} - h_\theta(x^{(i)})x_j^{(i)}$$

主要任务推导出来梯度上升的迭代公式，我们就可以直接coding，计算最佳的拟合参数。

实战试验：

这一次的实战对一堆数据进行处理，这些数据都是二维坐标上的一些点，每一个点都有一个标签0或1，我们就是要根据这个数据集进行拟合直线，实现Logistics的回归。

下面时数据集的样子：

> -0.017612 14.053064 0
> -1.395634 4.662541 1
> -0.752157 6.538620 0
> -1.322371 7.152853 0
> 0.423363 11.054677 0
> 0.406704 7.067335 1
> 0.667394 12.741452 0
> -2.460150 6.866805 1
> 0.569411 9.548755 0
> -0.026632 10.427743 0
> 0.850433 6.920334 1
> 1.347183 13.175500 0
> 1.176813 3.167020 1

对于这些数据我们使用线性模型来模拟，$z = w_0x_0 + w_1x_1 + w_2x_2$，且$x_0$时全为1，$x_1$为数据集第一列，$x_2$为数据集第二列，则$z = 0$，我们就是要求得最优化的回归系数。对于梯度上升公式：

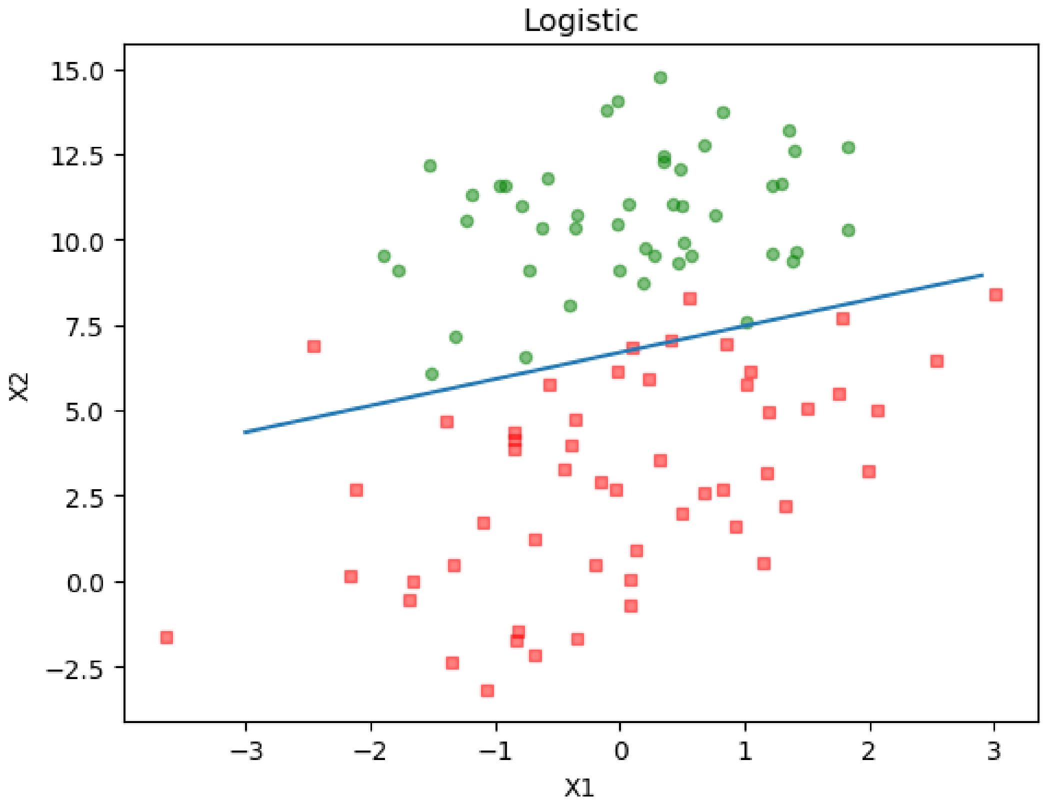
$$\theta_j := \theta_j + \alpha \sum_1^m (y^{(i)} - h_\theta(x^{(i)})x_j^{(i)}$$

可以得到代码：

```
import numpy as np
import matplotlib.pyplot as plt
"""
加载数据
把数据按一行一行都进去
然后返回数据矩阵以及相应的标签矩阵
"""
def loadDataSet():
    dataMat=[]
    labelMat=[]
    fr=open('testSet.txt')
    for line in fr.readlines() :
        lineArr = line.strip().split()
        dataMat.append([1.0,float(lineArr[0]),float(lineArr[1])])
        labelMat.append(int(lineArr[2]))
    fr.close()
```

```
    return dataMat,labelMat

"""
sigmoid 函数:
返回sigmoid函数的值
"""
def sigmoid(inx):
    return 1.0/(1+np.exp(-inx))


def gradAscent(dataMatIn,classLabels):
    dataMatrix = np.mat(dataMatIn)
    labelMat = np.mat(classLabels).transpose()
    m,n =np.shape(dataMatrix)
    alpha =0.001
    maxCycles = 500
    weights=np.ones((n,1))
    for k in range(maxCycles):
        h=sigmoid(dataMatrix*weights)
        error=labelMat-h
        weights=weights+alpha*dataMatrix.transpose()*error
    return weights.getA()

dataMat,labelMat=loadDataSet()
print(gradAscent(dataMat,labelMat))

def plotFit(weights):
    dataMat,labelMat = loadDataSet()
    dataArr=np.array(dataMat)
    n=np.shape(dataMat)[0]
    xcord1=[];ycord1=[];
    xcord2=[];ycord2=[];
    for i in range(n):
        if int(labelMat[i])==1:
            xcord1.append(dataArr[i,1])
            ycord1.append(dataArr[i,2])
        else:
            xcord2.append(dataArr[i,1])
            ycord2.append(dataArr[i,2])
    fig = plt.figure()
    ax=fig.add_subplot(111)
    ax.scatter(xcord1, ycord1, s = 20, c = 'red', marker = 's',alpha=.5)
    ax.scatter(xcord2,ycord2,s=20,c='green',alpha=.5)
    x=np.arange(-3.0,3.0,0.1)
    y=(-weights[0]-weights[1]*x)/weights[2]
    ax.plot(x,y)
    plt.title('Logistic')
    plt.xlabel('X1'); plt.ylabel('X2')
    plt.show()

weights=gradAscent(dataMat,labelMat)
plotFit(weights)
```

得到的分类:



这个结果已经相当不错了，但是还有很多可以改进的，我会跟进方案。

结论：

总的来说，Logistics的回归步骤就是先进行数据的收集，然后处理，这一步需要我们根据数据集的格式进行灵活的处理，最后的输出结果一般是带标签的矩阵，然后我们对参数向量乘以样本列向量作为sigmoid函数的输入，得到输出后根据梯度上升得到参数向量，就可以得到最终的模型并画出决策边界。