# [Python机器学习笔记：Logistic Regression](https://www.cnblogs.com/wj-1314/p/10181876.html)

### **完整代码及其数据，请移步小编的GitHub**

　　传送门：[请点击我](https://github.com/LeBron-Jian/MachineLearningNote" \t "https://www.cnblogs.com/wj-1314/p/_blank)

　　如果点击有误：https://github.com/LeBron-Jian/MachineLearningNote

## **Logistic回归公式推导和代码实现**

### **1，引言**

　　logistic回归是机器学习中最常用最经典的分类方法之一，有人称之为逻辑回归或者逻辑斯蒂回归。虽然他称为回归模型，但是却处理的是分类问题，这主要是因为它的本质是一个线性模型加上一个映射函数Sigmoid，将线性模型得到的连续结果映射到离散型上。它常用于二分类问题，在多分类问题的推广叫softmax。

　　本文首先阐述Logistic回归的定义，然后介绍一些最优化算法，其中包括基本的梯度上升法和一个改进的随机梯度上升法，这些最优化算法将用于分类器的训练，最好本文将给出一个Logistic回归的实例，预测一匹病马是否能被治愈。

　　在我们的日常生活中遇到过很多最优化问题，比如如何在最短时间内从A点到达B点？如何投入最少工作量却获得最大的效益？如何设计发动机使得油耗最少而功率最大？可见，最优化的作用十分强大，所以此处我们介绍几个最优化算法，并利用它们训练出一个非线性函数用于分类。

　　现在假设有一些数据点，我们用一条直线对这些点进行拟合（该线称为最佳拟合直线），这个拟合过程就称作回归。利用logistic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公式，以此进行分类，这里的“回归”一词源于最佳拟合，表示要找到最佳拟合参数集。训练分类器时的做法就是寻找最佳拟合参数，使用的是最优化算法，下面我们首先介绍一下这个二值型输出分类器的数学原理。

### **2，Logistic回归的一般过程**

* （1）**收集数据**：采用任意方法收集数据
* （2）**准备数据**：由于需要进行距离计算，因此要求数据类型为数值型。另外，结构化数据格式则最佳
* （3）**分析数据**：采用任意方法对数据进行分析
* （4）**训练算法**：大部分时间将用于训练，训练的目的是为了找到最佳的分类回归系数
* （5）**使用算法**：首先，我们需要输入一些数据，并将其转换成对应的结构化数值；接着，基于训练好的回归系数就可以对这些数值进行简单的回归计算，判定他们属于哪个类别；在这之后，我们就可以在输出的类别上做一些其他分析工作。

### **3，Logistic回归的优缺点**

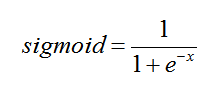
　　优点：计算代码不多，易于理解和实现，计算代价不高，速度快，存储资源低

　　缺点：容易欠拟合，分类精度可能不高

　　适用数据类型：数值型和标称型数据

### **4，基于Logistic回归和Sigmoid函数的分类**

 　　我们想要的函数应该是：**能接受所有的输入，然后预测出类型**。例如，在两个类的情况下，上述函数输出0或1。该函数称为海维赛德阶跃函数（Heaviside step function），或者直接称为单位阶跃函数。然而，海维赛德阶跃函数的问题在于：该函数在跳跃点上从0瞬间跳跃到1，这个瞬间跳跃过程有时很难处理。幸好，另一个函数也有类似的性质（可以输出0或者1），且数学上更易处理，这就是Sigmoid函数。Sigmoid函数具体的计算公式如下：



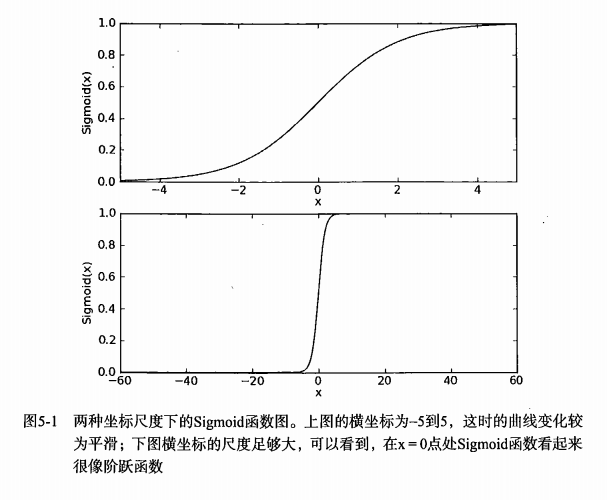
**自变量取值为任意实数，值域[0, 1]**

　　图5-1给出了Sigmoid函数在不同坐标尺度下的两条曲线图。当x为0时，Sigmoid函数值为0.5。随着x的增大，对应的Sigmoid值将逼近于1；而随着x的减少，Sigmoid值将逼近于0.如果横坐标刻度足够大，Sigmoid函数看起来很像一个阶跃函数。

　　解释Sigmoid函数：**将任意的输入映射到了 [0, 1]区间，我们在线性回归中可以得到一个预测值，再将该值映射到 Sigmoid函数中这样就完成了由值到概率的转换，也就是分类任务**。

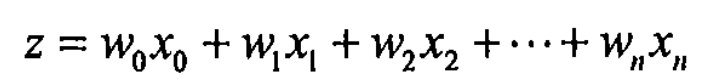
　　因此，为了实现Logistic回归分类器，我们可以在每个特征上都乘以一个回归系数，然后把所有的结果值相加，将这个总和带入Sigmoid函数中，进而得到一个范围在0~1之间的数值。任何大于0.5的数据被分入1类，小于0.5即被归入0类，所以，Logistic回归也可以被看成是一种概率估计。

　　确定了分类器的函数形式之后，现在的问题变成了：**最佳回归系数是多少？如何确定其大小**。

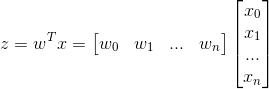


### **5，基于最优化方法的最佳回归系数确定**

　　Sigmoid函数的输入记为z，由下面公式得到：

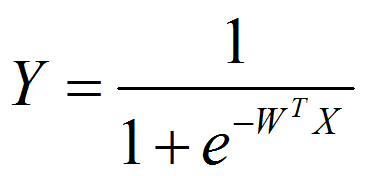


　　如果采用向量的写法，上述公式可以写成  z = wTx  ，它表示将这两个数值向量对应元素相乘，然后全部加起来即得到z值。

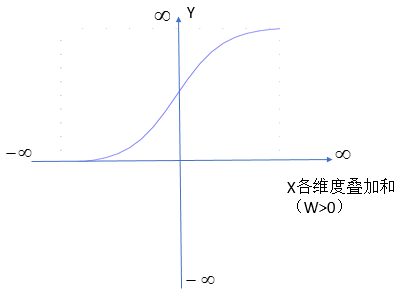


　　其中的向量x是分类器的输入数据，向量w也就是我们要找到的最佳参数（系数），从而使得分类器尽可能的准确，为了寻找该最佳参数，需要用到最优化理论的一些知识。

　　然后再看看我们的Logistic回归模型的公式：



　　这里假设 W>0，Y与X各维度叠加的图形关系，如下图所示（x为了方便取1维）：

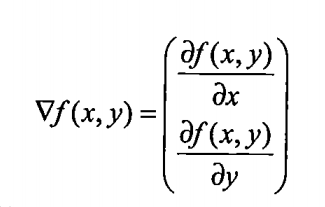


　　下面首先学习梯度上升的最优化方法，我们将学习到如何使用该方法求得数据集的最佳参数，接下来，展示如何绘制梯度上升法产生的决策边界图，该图将梯度上升法的分类效果可视化的呈现出来，最后我们将学习随机梯度上升算法，以及如何对其进行修改以获得很好地结果。

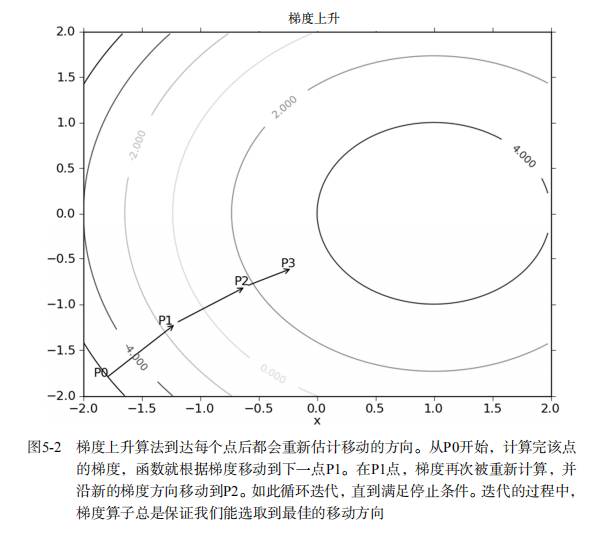
|  |  |
| --- | --- |
| 1  2 | 可能我们最常听到的是梯度下降算法，它与这里的梯度上升算法是一样的，只是公式中的  加法需要变成减法，梯度上升算法用来求函数的最大值，而梯度下降算法是用来求函数的最小值 |

### **6，梯度上升法**

　　梯度上升法基于的思想是：要找到某函数的最大值，最好的方法是沿着该函数的梯度方向探寻，如果梯度记为IMG_262，则函数 f(x,y) 的梯度由下面式子表示：



　　这个梯度意味着要沿着x的方向移动，沿着y方向移动，其中函数f(x,y)必须要在待计算的点上有定义并且可微，一个具体的函数例子见图5-2：



 　　上图中的梯度上升算法沿梯度方向移动了一步，可以看出，梯度算子总是指向函数值增长最快的方向。这里所说的移动方向，而未提到移动量的大小。该量值称为步长，记为IMG_267。用向量来表示的话，梯度算法的迭代公式如下：

IMG_268

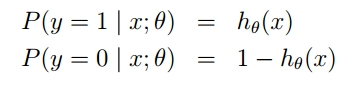
　　该公式将一直被迭代执行，直至达到某个停止条件为止，比如迭代次数达到某个指定值或算法达到某个可以允许的误差范围。

　　基于上面的内容，我们来看一个Logistic回归分类器的应用例子，从图5-3可以看到我们采用的数据集。

### **IMG_269**

### **梯度上升法的公式推导（LR 损失函数）**

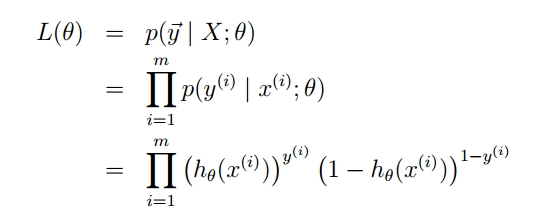
　　在LR中，应用极大似然估计法估计模型参数，由于Sigmoid函数的特性，我们可以做如下的假设：



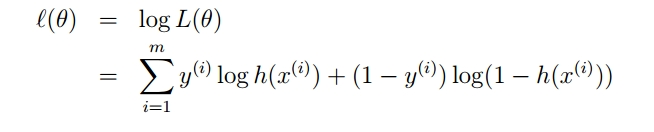
　　上式即为在已知样本X和参数θ的情况下。样本X属性正类（y=1）和负类（y=0）的条件概率，将两个公式合并成一个，如下：

IMG_271

**假定样本与样本之间相互独立，那么整个样本集生成的概率即为所有样本生成概率的乘积**（也就是n个独立样本出现的似然函数如下）：



　　为了简化问题，我们对整个表达式求**对数（即为LR 损失函数）**：



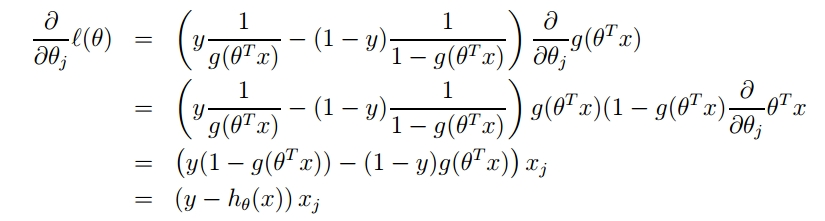
**满足似然函数（θ）的最大的θ值即时我们需要求解的模型。**

　　那么梯度上升法就像爬坡一样，一点一点逼近极值，而上升这个动作用数学公式表达即为：

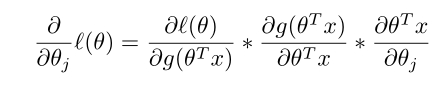
IMG_274

　　其中，α 为步长。

　　回到Logistic回归问题，我们同样对函数求偏导。

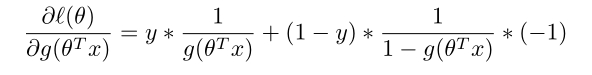


对这个公式进行分解，先看：

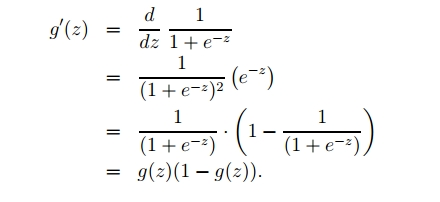


　　我们可以看到，对函数求偏导，分解为三部分，然后我们对这三部分分布求导。

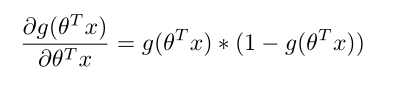
其中：



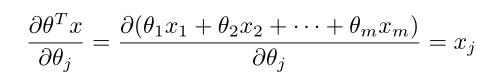
再由：



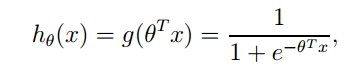
可得：



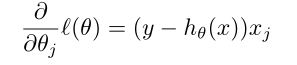
接下来：



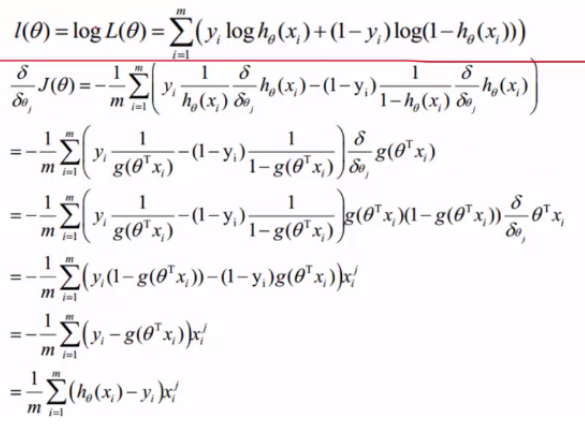
最后：



综合三部分即得到：



　　如果上面链式分解不好理解的话，可以看下面直接求导（结果是一样的）：

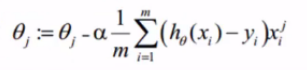


 　　注意上面是将梯度上升求最大值，转换为梯度下降了，本质没变。

因此梯度迭代公式为：

IMG_284

 如果为梯度下降，我们注意符号的变化，如下：



### **7，训练算法：使用梯度上升找到最佳参数**

 　　上图有100个样本点，每个点包含两个数值型特征：X1和X2，在此数据集上，我们将通过使用梯度上升法找到最佳回归系数，也就是拟合出Logistic回归模型的最佳参数。

　　所以我们的目标：**建立分类器，求解出theta参数**

**设定阈值，根据阈值判断结果**

　　梯度上升法的伪代码如下：

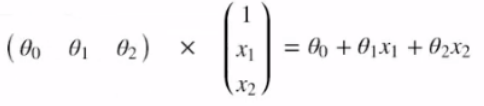
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 每个回归系数初始化为1    重复R次：        计算整个数据集的梯度        使用alpha \* gradient 更新回归系数的向量        返回回归系数 |

　　testSet.txt的文件内容请去我的GitHub下载。地址：https://github.com/LeBron-Jian/MachineLearningNote

　　下面具体实现**梯度上升算法**的代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44 | #\_\*\_coding:utf-8\_\*\_  from numpy import \*    # 读取数据  def loadDataSet(filename):      '''          对于testSet.txt，每行前两个值分别是X1和X2，第三个值数据对应的类别标签          而且为了设置方便，该函数还将X0的值设置为1.0          :return:          '''      dataMat = []      labelMat = []      fr = open(filename)      for line in fr.readlines():          lineArr = line.strip().split()          dataMat.append([1.0, float(lineArr[0]), float(lineArr[1])])          labelMat.append(int(lineArr[2]))      return dataMat,labelMat    def sigmoid(inX):      return 1.0/(1+exp(-inX))    def gradAscent(dataMatIn,classLabels):      '''          :param dataMatIn: 是一个2维Numpy数组，每列分别代表每个不同的特征          每行则代表每个训练样本。          :param classLabels: 是类别标签，是一个1\*100的行向量，为了便于矩阵运算，需要将行向量          转换为列向量，就是矩阵的转置，再将其赋值与labelMat。          :return:          '''      dataMatrix = mat(dataMatIn)      labelMat = mat(classLabels).transpose()      # labelMat = mat(classLabels).T      m,n = shape(dataMatrix)      # alpha是向目标移动的步长      alpha = 0.001      # 迭代次数      maxCycles = 500      weights = ones((n,1))      for k in range(maxCycles):          h = sigmoid(dataMatrix\*weights)          error = (labelMat-h)          weights = weights + alpha\*dataMatrix.transpose()\*error      return weights |

　　注意上面代码（为什么将x0的值设置为1呢，主要是我们要使用矩阵进行运算，可以看下图比较明显）



　　测试结果如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | if \_\_name\_\_  == '\_\_main\_\_':      filename = 'testSet.txt'      dataArr,labelMat = loadDataSet(filename)      weights\_res = gradAscent(dataArr,labelMat)      print(weights\_res)    '''  [[ 4.12414349]   [ 0.48007329]   [-0.6168482 ]]   ''' |

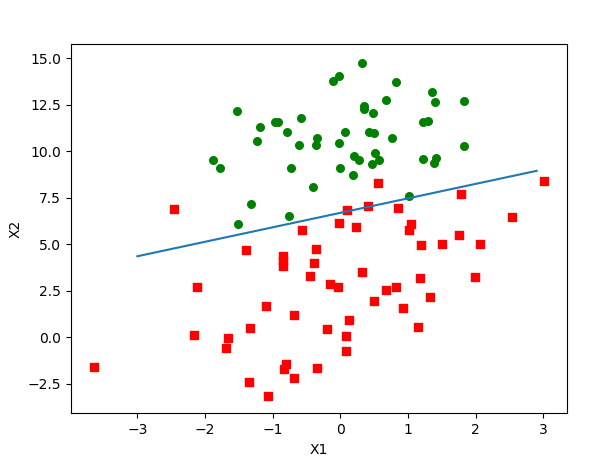
　　上面已经解出了一组回归系数，它确定了不同类别数据之间的分割线，那么怎样画出该分割线，从而使得优化的过程便于理解呢？下面代码来解决这个问题。

　　画出数据集和Logistic回归最佳拟合直线的函数代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | def plotBestFit(wei):      import matplotlib.pyplot as plt      weights = wei.getA()      dataMat,labelMat = loadDataSet(filename)      dataArr = array(dataMat)      n = shape(dataArr)[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=30,c='red',marker='s')      ax.scatter(xcord2,ycord2,s=30,c='green')      x = arange(-3.0,3.0,0.1)      y = (-weights[0]-weights[1] \* x) / weights[2]      ax.plot(x,y)      plt.xlabel('X1')      plt.ylabel('X2')      plt.show() |

　　输出的结果和代码如下图所示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if \_\_name\_\_  == '\_\_main\_\_':      filename = 'testSet.txt'      dataArr,labelMat = loadDataSet(filename)      weights\_res = gradAscent(dataArr,labelMat)      print(weights\_res)      plotBestFit(weights\_res) |



**梯度上升算法在500次迭代后得到的Logistic回归最佳拟合直线**

　　这个分类结果相当不错，从图上看，只错分了四个点。但是，尽管例子简单且数据集很小，这个方法却需要大量的计算（300次乘法），因此下一节将对算法稍作改进，从而使它可以用在真实数据集上。

### **8，训练算法：随机梯度上升**

 　　梯度上升算法在每次更新回归系数时都需遍历整个数据集，该方法在处理100个左右的数据集尚可，但是若有数十亿样本和成千上万的特征，那么该方法的计算复杂度就太高了。一种改进方法是一次仅用一个样本点来更新回归系数，该方法称为随机梯度上升算法。由于可以在新样本到来时对分类器进行增量式更新，因而随机梯度上升算法是一个在线学习算法，与“在线学习”相对应，一次处理所有数据被称作是“批处理”。

　　随机梯度上升算法可以写成如下的伪代码：

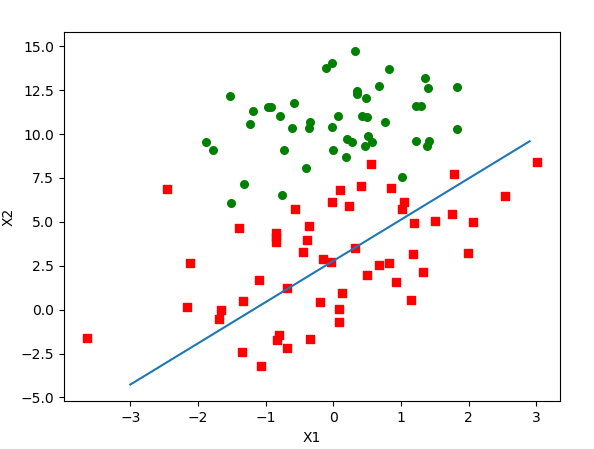
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | 所有回归系数初始化为1    对数据集中每个样本        计算该样本的梯度        使用alpha\*gradient 更新回归系数值    返回回归系数值 |

　　以下是随机梯度上升算法的实现代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10 | # 随机梯度上升算法  def stocGradAscent0(dataMatrix,classLabels):      m,n = shape(dataMatrix)      alpha = 0.01      weights = ones(n)      for i in range(m):          h = sigmoid(sum(dataMatrix[i]\*weights))          error = classLabels[i] - h          weights = weights + alpha\*error\*dataMatrix[i]      return weights |

　　实现的代码如下：

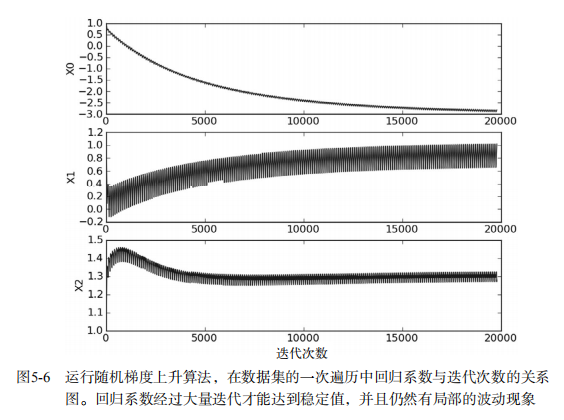
|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if \_\_name\_\_  == '\_\_main\_\_':      filename = 'testSet.txt'      dataArr,labelMat = loadDataSet(filename)      weights\_res = stocGradAscent0(array(dataArr),labelMat)      print(weights\_res)      plotBestFit(weights\_res) |



**图5-5  随机梯度上升算法在上述数据集上的执行结果，最佳拟合直线并非最佳分类线**

　　可以看出，拟合出来的直线效果还不错，但并不像，但是不像上面那个完美，这里的分类器错分了三分之一的样本。

　　直接比较结果两个结果是不公平的，后者的结果是在整个数据集上迭代了500次才得到的。一个判断优化算法优劣的可靠方法是看它是否收敛，也就是说参数是否达到了稳定值，是否还会不断地变化？对此，我们在上面的的随机梯度算法上做了些修改，使其在整个数据集上运行200次，最终绘制的三个回归系数的变化情况如下图所示：



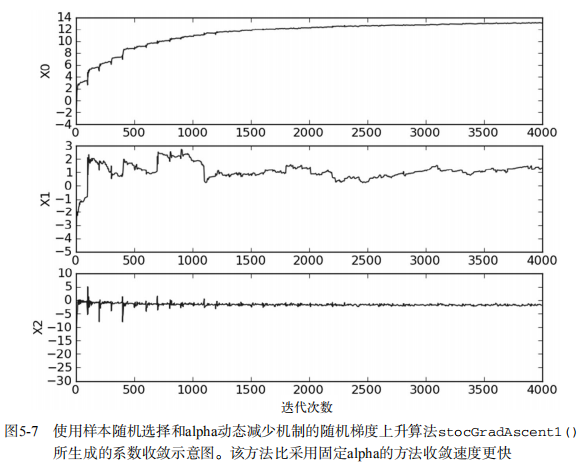
　　上图展示了随机梯度上升算法在200次迭代过程中回归系数的变化情况，其中的系数2，也就是图5-5中的X2只经过了50次迭代就达到了稳定值，但系数1和0则需要更多次的迭代。另外值得注意的是，在大的波动停止后，还有一些小的周期性波动。不难理解，产生这种现象的原因是存在一些不能正确分类的样本点（数据集并非现象可分），在每次迭代时会引发系数的剧烈改变。我们期望算法能避免来回波动，从而收敛到某个值。另外，收敛速度也需要加快。

　　改进的随机梯度上升算法代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | # 改进的随机梯度上升算法  def stocGradAscent1(dataMatrix,classLabels,numIter=150):      m,n = shape(dataMatrix)      weights = ones(n)      for j in range(numIter):          dataIndex = list(range(m))          for i in range(m):              alpha = 4/(1.0+j+i)+0.01              randIndex = int(random.uniform(0,len(dataIndex)))              h = sigmoid(sum(dataMatrix[randIndex]\*weights))              error = classLabels[randIndex] - h              weights = weights + alpha \*error\*dataMatrix[randIndex]              del(dataIndex[randIndex])      return weights |

　　上述代码大体上与之前的随机梯度上升算法一致，修改了两处，一处是alpha在每次迭代的时候都会调整，这会环节之前的数据波动或者高频波动。另外，虽然alpha会随着迭代次数不断减少，但永远不会减少到0,。必须这样做的原因是为了保证在多次迭代之后新数据仍然具有一定的影响。如果要处理的问题是动态变化的，那么可以适当增加常数项，来确保新的值获得更大的回归系数。另外一点值得注意的是，在降低alpha的函数中，alpha每次减少1/(j+1)，其中j是迭代次数，i是样本点的下标，这样当j<max(i)的时候，alpha就不是严格下降的，避免参数的严格下降也常见于模拟退火算法等其他优化算法中。程序的第二个改进地方就是通过随机选取样本来更新回归系数，这种方法将减少周期性的波动，并且改进的算法还增加一个迭代次数作为第三个参数，如果该参数没有给定的话，算法将默认迭代150次，如果给定，那么算法将按照新的参数值进行迭代。

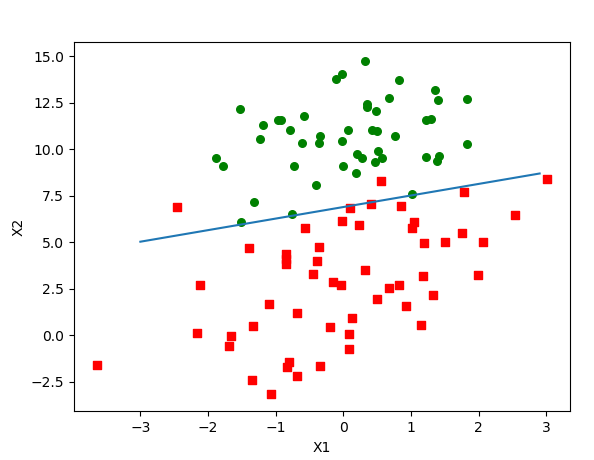
　　与stocGradAscent1()类似，下图显示了每次迭代时各个回归系数的变化情况。



　　比较5-7和5-6可以看到两点不同，第一点是，图5-7中的系数没有像5-6里那样出现周期性的波动，这归功于stocGradAscent1()里的样本随机选择机制，第二点是5-7的水平轴比5-6的短了很多，这是由于stocGradAscent1()可以收敛的更快，这次我们仅仅对数据集做了20次遍历，之前是500次。

　　下面看看在同一个数据集上的分类效果，将程序运行可以看到：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if \_\_name\_\_  == '\_\_main\_\_':      filename = 'testSet.txt'      dataArr,labelMat = loadDataSet(filename)      weights\_res = stocGradAscent1(array(dataArr),labelMat)      print(weights\_res)      plotBestFit(weights\_res) |



　　该分割线达到了与GradientAscent()差不多的效果，但是所使用的计算量更少。

　　默认的迭代次数是150次，但是我们通过stocGradAscent()的第三个参数来对此进行修改，例如：

|  |  |
| --- | --- |
| 1 | weights\_res = stocGradAscent1(array(dataArr),labelMat,500) |

　　迄今为止我们分析了回归系数的变化情况，但是还没有达到目的，，即完成具体的分类任务，下面我们将使用随机梯度上升算法来解决病马的生死预测问题。,

### **9，示例：从疝气病症预测病马的死亡率**

　　本次将**使用logistic回归来预测患有疝气的马的存活问题**，这里的数据包含了368个样本和28个特征。（疝气指的是马胃肠痛的术语，然而这种病不一定源于马 的肠胃问题，其他问题也可以引发疝气）该数据集包含了医院检测马疝气病的一些指标，有的指标比较主观，有的指标难以预测，例如马的疼痛级别

#### **使用Logistic回归估计马疝气病的死亡率的流程**

（1）收集数据：使用给定数据文件

（2）准备数据：用Python解析文本文件并填充缺失值

（3）分析数据：可视化并观察数据

（4）训练算法：使用优化算法，找到最佳的系数

（5）测试算法：为了量化回归的结果，需要观察错误率，根据错误率决定是否回退到训练阶段，通过改变迭代的次数和步长等参数来得到更好的回归系数

（6）使用算法：实现一个简单的命令行程序来收集马的症状并输出预测结果

### **准备数据：处理数据中的缺失值**

　　数据中的缺失值是个非常棘手的问题，有很少文献都致力于解决这个问题。那么，数据缺少究竟带来了什么问题？假设有100个样本和20个特征，这些数据都是机器收集回来的，若机器上的某个传感器损坏导致一个特征无效时该怎么办？此时是否要扔掉整个数据？这种情况下，另外19个特征怎么办？它们是否还可用？答案是肯定的。因为有时候数据相当昂贵，扔掉和重新获取都是不可取的，所以必须采取一些方法来解决这个问题。

　　下面给出了一些可选的做法：

* 使用可用特征的均值来填补缺失值；
* 使用特征值来填充缺失值，如-1
* 忽略有缺失值的样本
* 使用相似样本的均值填补缺失值
* 使用另外的机器学习算法预测缺失值

 　　现在，我们对下一节要用的数据集进行预处理，使其可以顺利地使用分类算法。在预处理阶段需要做两件事：第一，所有的缺失值必须用一个实数值来替换，因为我们使用的Numpy数据类型不允许包含缺失值。这里选择实数0来替换，因为我们使用的Numpy数据类型不允许包括缺失值，**这里选择实数0来替换所有缺失值，恰好能适应于Logistic回归**。这样做的直觉在于，我们需要的是一个在更新时不会影响系数的值，回归系数的更新公式如下：

|  |  |
| --- | --- |
| 1 | weights = weights + alpha \*error\*dataMatrix[randIndex] |

　　如果dataMatirx的某特征对应值为0，那么该特征的系数将不做更新，即：

|  |  |
| --- | --- |
| 1 | weights = weights |

　　另外，由于sigmoid(0) = 0.5 ，即对它结果的预测不具有任何倾向性，因此上述做法也不会对误差项造成任何影响，基于上述原因，将缺失值用0代替既可以保留现有数据，也不需要对优化算法进行修改，此外，数据集中的特征值一般不取0，因此在某种意义上说它也满足“特殊值”这个要求。

　　预处理中做的第二件事，是如果在测试数据集中发现了一条数据的类别标签以及缺失，那么我们的简单做法是将该条数据丢弃。这是因为类别标签与特征不同，很难确定采用某个合适的值来替换，采用Logistic回归进行预处理之后保存两个文件，horseColicTest.txt和horseColicTraining.txt。如果想对于原始数据和预处理之后的数据做个比较。

　　我们有一个“干净”可用的数据集和一个不错的优化算法，下面将这些部分融合在一起训练出一个分类器，然后利用该分类器来预测病马的生死问题。

　　horseColicTest.txt的数据 horseColicTraining.txt的数据  请去我的GitHub下载：https://github.com/LeBron-Jian/MachineLearningNote

 完整代码：

#-\*\_coding:utf-8\_\*\_

import math

from numpy import \*

# Sigmoid函数的计算

def sigmoid(inX):

    return 1.0/(1+exp(-inX))

# 改进的随机梯度上升算法

def stocGradAscent1(dataMatrix,classLabels,numIter=150):

    m,n = shape(dataMatrix)

    weights = ones(n)

    for j in range(numIter):

        dataIndex = list(range(m))

        for i in range(m):

            alpha = 4/(1.0+j+i)+0.01

            randIndex = int(random.uniform(0,len(dataIndex)))

            h = sigmoid(sum(dataMatrix[randIndex]\*weights))

            error = classLabels[randIndex] - h

            weights = weights + alpha \*error\*dataMatrix[randIndex]

            del(dataIndex[randIndex])

    return weights

# Logistic回归分类函数

def  classifyVetor(inX,weights):

    prob = sigmoid(sum(inX\*weights))

    if prob > 0.5:

        return 1.0

    else:

        return 0.0

def colicTest(filetrain,filetest):

    frTrain = open(filetrain)

    frTest = open(filetest)

    trainingSet = []

    trainingLabeles = []

    for line in frTrain.readlines():

        currLine = line.strip().split('\t')

        lineArr = []

        for i in range(21):

            lineArr.append(float(currLine[i]))

        trainingSet.append(lineArr)

        trainingLabeles.append(float(currLine[21]))

    trainWeights = stocGradAscent1(array(trainingSet),trainingLabeles,500)

    errorCount = 0

    numTestVec = 0

    for line in frTest.readlines():

        numTestVec += 1.0

        currLine = line.strip().split('\t')

        lineArr =[]

        for i in range(21):

            lineArr.append(float(currLine[i]))

        if int(classifyVetor(array(lineArr),trainWeights)) != int(currLine[21]):

            errorCount += 1

    errorRate = (float(errorCount)/numTestVec)

    print('the error rate of this test is  %f'%errorRate)

    return errorRate

def multTest(filetrain,filetest):

    numTests = 6

    errorSum = 1.0

    for k in range(numTests):

        errorSum += colicTest(filetrain,filetest)

    print('after %d iterations  the average error rate is %f'%(numTests,errorSum/float(numTests)))

if \_\_name\_\_ == '\_\_main\_\_':

    filetrain = 'horseColicTraining.txt'

    filetest = 'horseColicTest.txt'

    multTest(filetrain,filetest)

　　运行结果：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | the error rate of this test is  0.373134  the error rate of this test is  0.358209  the error rate of this test is  0.402985  the error rate of this test is  0.432836  the error rate of this test is  0.462687  the error rate of this test is  0.343284  after 6 iterations  the average error rate is 0.562189 |

　　从结果来看，6次迭代之后的平均错误率为0.56，事实上，这个结果还不错，因为数据有30%的数据缺失，当然如果调整colicTest（）中的迭代次数和stochGradAscent1()中的步长，平均错误率就可以降到20%左右。

### **10，使用梯度下降求解逻辑回归**

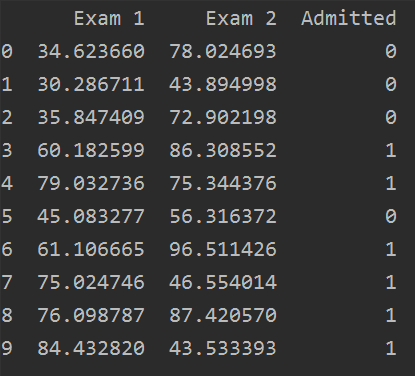
　　这里再补充一个示例，我们将建立一个逻辑回归模型来预测一个学生是否被大学录取。假设你是一个大学习的管理员，你想根据两次考试的结果来决定每个申请人的录取机会。你又以前的申请人的历史数据，你可以用它作为逻辑回归的训练集。对于每一个培训例子，你有两个考试的申请人的分数和录取决定。为了做到这一点，我们将建立一个分类模型，根据考试成绩估计入学概率。

　　数据和代码请参考我的GitHub：https://github.com/LeBron-Jian/MachineLearningNote

　　首先，拿到数据，并查看部分数据：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | import numpy as np  import pandas as pd  import matplotlib.pyplot as plt    path = 'LogiReg\_data.txt'  pdData = pd.read\_csv(path, header=None, names=['Exam 1', 'Exam 2', 'Admitted'])  print(pdData.head(10)) |

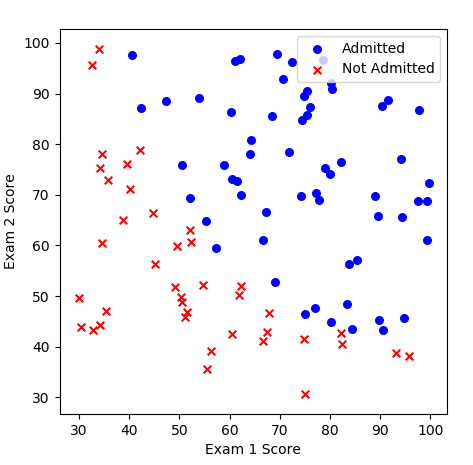
 　　部分数据如下：



 　　拿到数据后，我们将其按照结果分为录取和不被录取的集合，并展示：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8 | positive = pdData[pdData['Admitted'] == 1]  negative = pdData[pdData['Admitted'] == 0]  fig, ax = plt.subplots(figsize=(5, 5))  ax.scatter(positive['Exam 1'], positive['Exam 2'], s=30, c='b', marker='o', label='Admitted')  ax.scatter(negative['Exam 1'], negative['Exam 2'], s=30, c='r', marker='x', label='Not Admitted')  ax.legend()  ax.set\_xlabel('Exam 1 Score')  ax.set\_ylabel('Exam 2 Score') |

 　　图示如下：



 　　下面使用逻辑回归来建立分类器，我们需要求解出三个参数 theta0,  theta1,  theta2。 然后设定阈值，根据阈值判断录取结果。

　　要完成的模块函数：

* Sigmoid：映射到概率的函数
* model：返回预测结果值
* cost：根据参数计算损失
* gradient：计算每个参数的梯度方法
* descent：进行参数更新
* accuracy：计算精度

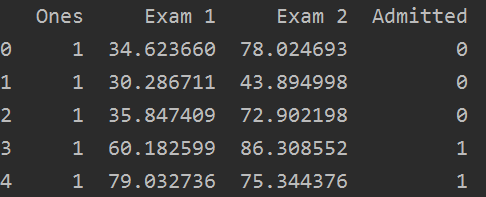
　　首先，完成Sigmoid函数：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def sigmoid(z):      return 1 / (1 + np.exp(-z))    # 如果想查看Sigmoid函数，可以运行下面代码  nums = np.arange(-10, 10, step=1)  fig, ax = plt.subplots(figsize=(12, 4))  ax.plot(nums, sigmoid(nums), 'r') |

 　　然后对数据集进行划分，分为训练数据和标签，其中我们添加了一列，并且初始化了theta值

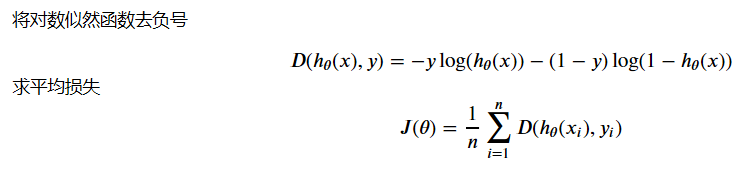
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | # 增加一行1  在第0列  pdData.insert(0, 'Ones', 1)  print(pdData.head())  # 设置X和Y  即训练数据和训练变量  orig\_data = pdData.values  cols = orig\_data.shape[1]  # (100, 4)  X = orig\_data[:, 0:cols-1]   # 等价于 np.matrix(X.values)  Y = orig\_data[:, cols-1:cols]  # 等价于 np.matrox(data.iloc[:, 3:4].value    # 传递numpu矩阵 并初始化参数矩阵 theta  theta = np.zeros([1, 3])  # [[0. 0. 0.]] |

 　　我们可以看一下添加了一列后，总体的数据：



 　　其他的数据均可以自己打印。

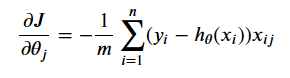
　　下面是损失函数：



 　　代码整理如下：

|  |  |
| --- | --- |
| 1  2  3  4 | def cost(X, y, theta):      left = np.multiply(-y, np.log(model(X, theta)))      right = np.multiply(1-y, np.log(1 - model(X, theta)))      return np.sum(left - right) / (len(X)) |

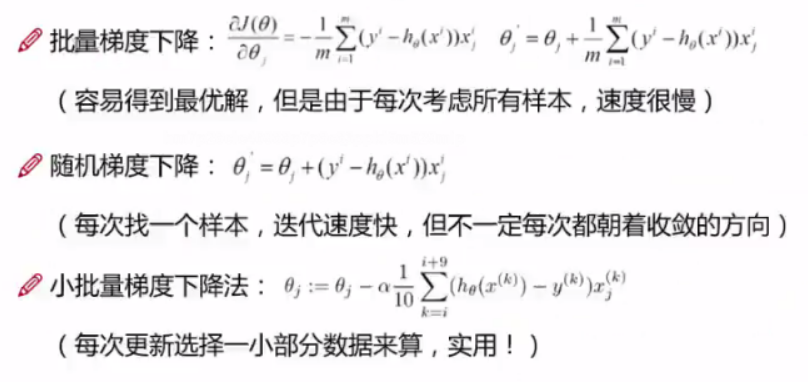
 　　下面是计算梯度，梯度函数如下（计算梯度就是我们对theta求偏导）：



 　　代码可写成下面方式：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7 | def gradient(X, y, theta):      grad = np.zeros(theta.shape)      error = (model(X, theta)-y).ravel()      for j in range(len(theta.ravel())):  # for each parmeter          term = np.multiply(error, X[:, j])          grad[0, j] = np.sum(term) / len(X)      return grad |

 　　下面比较三种不同梯度下降方法，三种梯度下降分别是 批量梯度下降，随机梯度下降，小批量梯度下降。如下图：



 　　我们的方法为1，根据迭代次数停止  2，根据损失函数停止  3，根据梯度下降停止

　　（注意：我们每次进行迭代更新的时候，都会对数据进行洗牌，一把情况数据都会有某种规律）

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50 | # 比较三种不同梯度下降方法  STOP\_ITER = 0  STOP\_COST = 1  STOP\_GRAD = 2    def stopCriterion(type, value, threshold):      # 设定三种不同的停止策略      if type == STOP\_ITER:          return value > threshold      elif type == STOP\_COST:          return abs(value[-1]-value[-2]) < threshold      elif type == STOP\_GRAD:          return np.linalg.norm(value) < threshold    # 洗牌  def shuffleData(data):      np.random.shuffle(data)      cols = data.shape[1]      X = data[:, 0:cols-1]      y = data[:, cols-1:]      return X, y    def descent(data, theta, batchSize, stopType, thresh, alpha):      # 梯度下降求解      init\_time = time.time()      i = 0  # 迭代次数      k = 0  # batch      X, y = shuffleData(data)      grad = np.zeros(theta.shape)  # 计算的梯度      costs = [cost(X, y, theta)]  # 损失值        while True:          grad = gradient(X[k:k+batchSize], y[k:k+batchSize], theta)          k += batchSize  # 取batch数量个数据          if k>=n:              k = 0              X, y = shuffleData(data)  # 重新洗牌          theta = theta = alpha\*grad  # 参数更新          costs.append(cost(X, y, theta))  # 计算新的损失          i += 1            if stopType == STOP\_ITER:              value = i          elif stopType == STOP\_COST:              value = costs          elif stopType == STOP\_GRAD:              value = grad          if stopCriterion(stopType, value, thresh):              break      return theta, i-1, costs, grad, time.time()-init\_time |

 　　下面我们定义一个功能性的函数，此函数的作用就是执行一次梯度下降，并完成更新，选择停止策略。其中主要代码就第一句。。。

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27 | def runExpe(data, theta, batchSize, stopType, thresh, alpha):      theta, iter, costs, grad, dur = descent(data, theta, batchSize, stopType, thresh, alpha)      name = 'Original' if (data[:, 1] > 2).sum() > 1 else 'Scaled'      name += 'data - learning rate: {} -'.format(alpha)      if batchSize == n:          strDescType = 'Gradient'      elif batchSize == 1:          strDescType = 'Stochastic'      else:          strDescType = 'Min-batch({})'.format(batchSize)      name += strDescType + 'descent - Stop: '      if stopType == STOP\_ITER:          strStop = "{} iterations".format(thresh)      elif stopType == STOP\_COST:          strStop = "costs change < {}".format(thresh)      else:          strStop = 'gradient norm < {}'.format(thresh)      name += strStop      print("\*\*\*{}\nTheta: {} - Iter:{}-Last cost: {:03.2f}--Duration: {:03.2f}s".format(          name, theta, iter, costs[-1], dur      ))      fig, ax = plt.subplots(figsize=(12, 4))      ax.plot(np.arange(len(costs)), costs, 'r')      ax.set\_xlabel('Iterations')      ax.set\_ylabel('Cost')      ax.set\_title(name.upper() + '- Error vs. Iteration')      return theta |

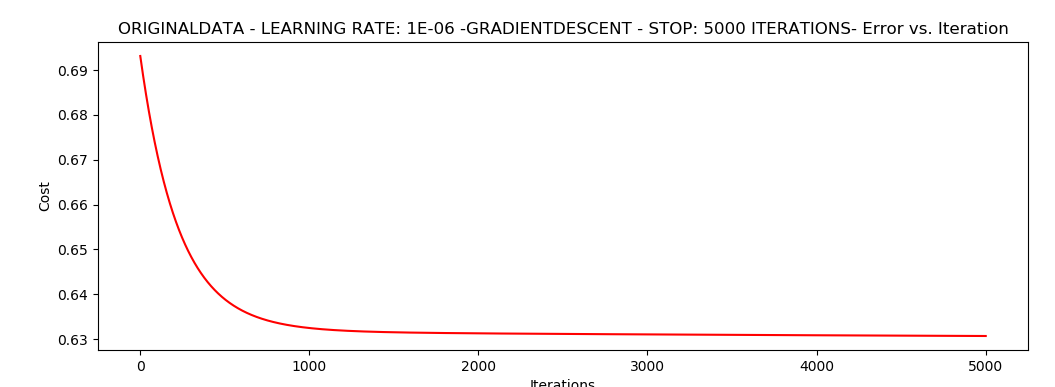
 　　下面按照不同的停止策略画图展示：

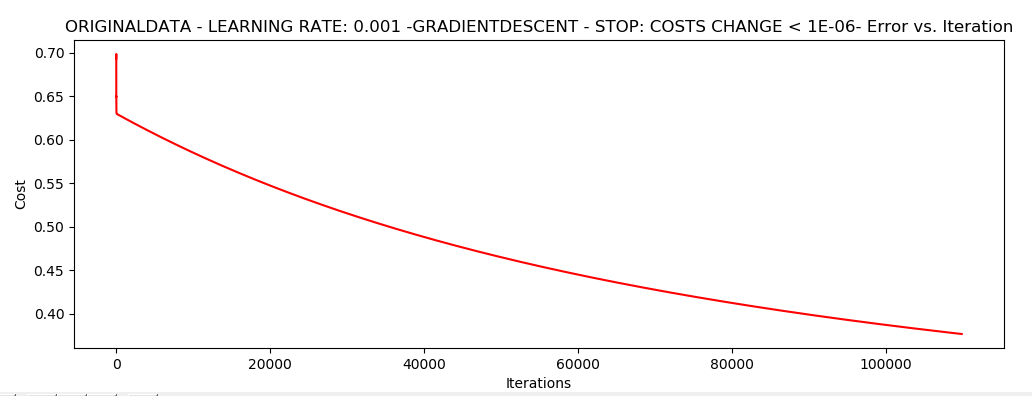
1，设定迭代次数  5000

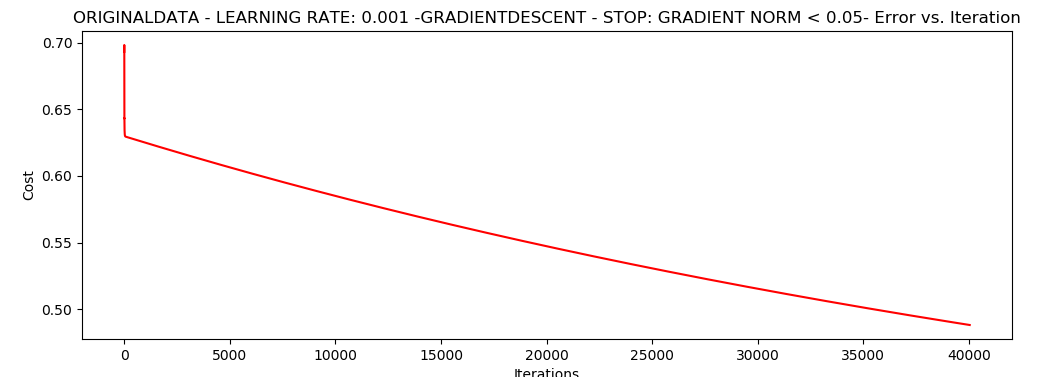
2，设定阈值1e-6，差不多需要 110000次迭代

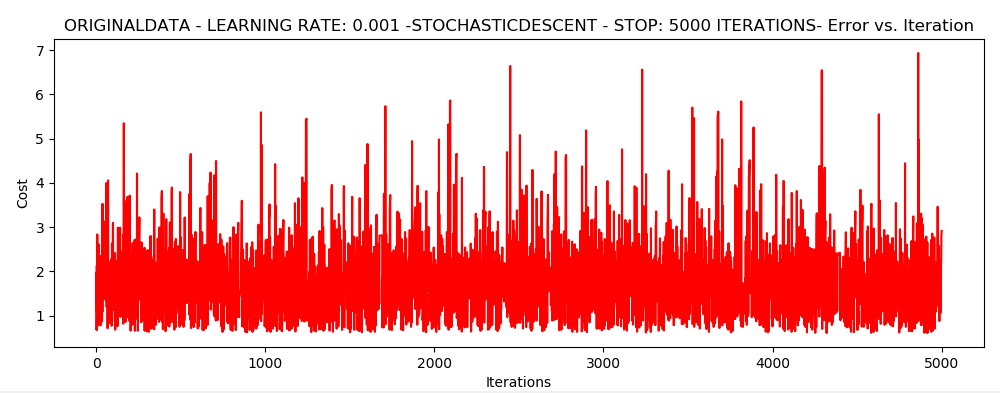
3，设置阈值 0.05，差不多需要 40000次迭代

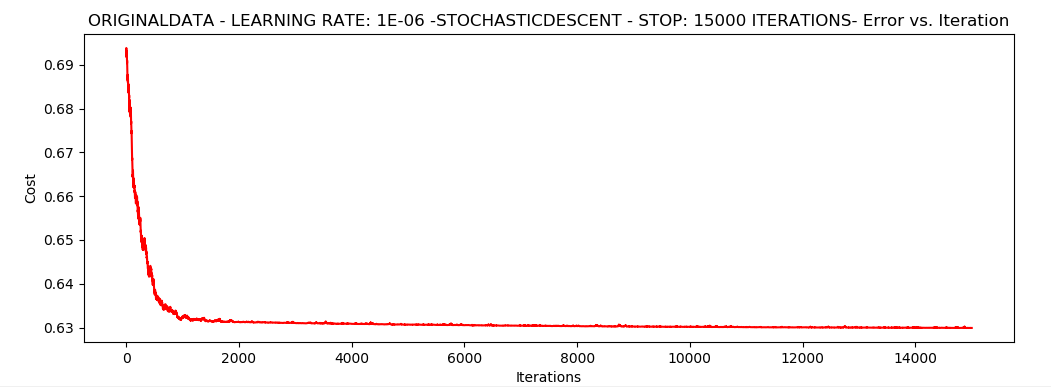
|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14 | if \_\_name\_\_ == '\_\_main\_\_':      # 设定迭代次数   选择的梯度下降方法是基于所有样本的      n = 100      # runExpe(orig\_data, theta, n, STOP\_ITER, thresh=5000, alpha=0.000001)      # 根据损失值停止    设定阈值 1e-6 差不多需要110 000 次迭代      # runExpe(orig\_data, theta, n, STOP\_COST, thresh=0.000001, alpha=0.001)      # 根据梯度变换停止  设定阈值 0.05，差不多需要 40000次迭代      # runExpe(orig\_data, theta, n, STOP\_GRAD, thresh=0.05, alpha=0.001)      # 对比不同的梯度下降方法  stochastic descent  这个不收敛      # runExpe(orig\_data, theta, 1, STOP\_ITER, thresh=5000, alpha=0.001)      # 我们将学习率调小，提高thresh再试试，发现收敛了      # runExpe(orig\_data, theta, 1, STOP\_ITER, thresh=15000, alpha=0.000001)      # Mine-batch descent  这个也不收敛 ，我们将alpha调小 0.000001  发现收敛了      runExpe(orig\_data, theta, 16, STOP\_ITER, thresh=15000, alpha=0.001) |

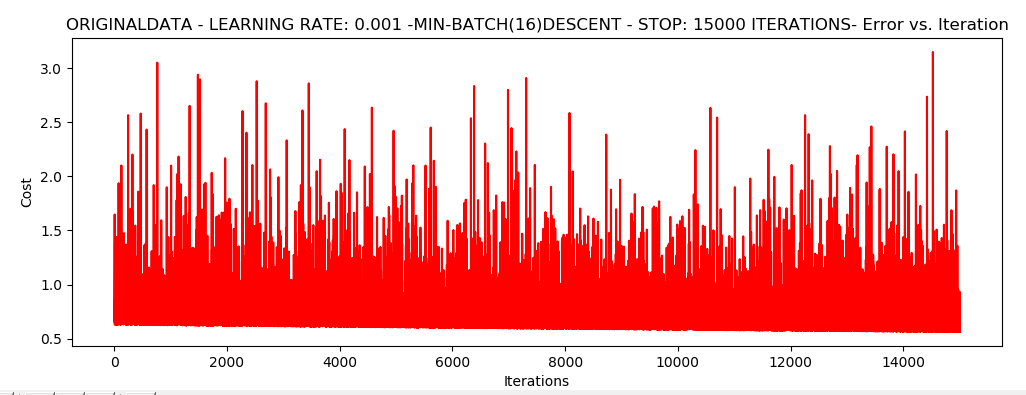








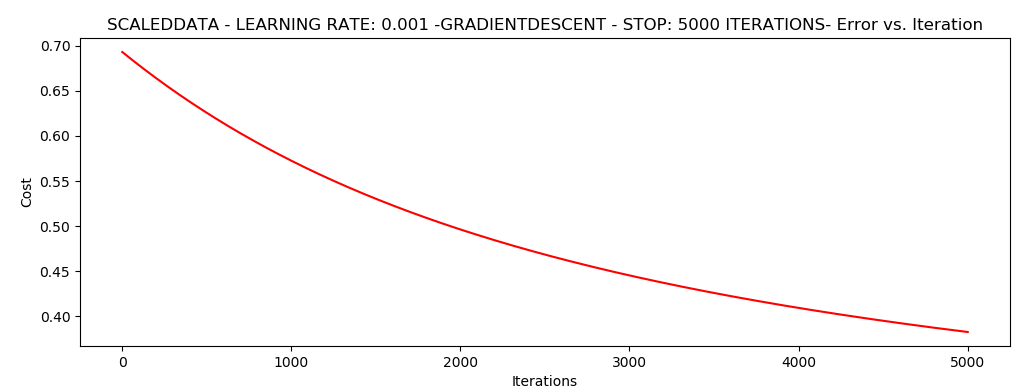


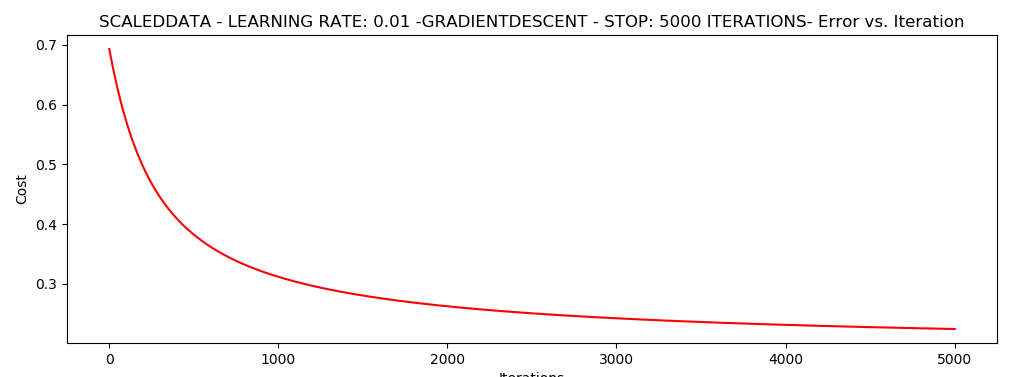


 　　浮动仍然比较大，我们来尝试下对数据进行标准化，将数据按其属下（按列进行）减去其均值，然后除以其方差。最后得到的结果是，对每个属性/每列来说所有数据都聚集在 0 附加，方差值为1。

|  |  |
| --- | --- |
| 1  2  3 | scaled\_data = orig\_data.copy()  scaled\_data[:, 1:3] = pp.scale(orig\_data[:, 1:3])  runExpe(scaled\_data, theta, n, STOP\_ITER, thresh=5000, alpha=0.01) |

 　　下面两幅图，我们分别将学习率设置为 0.001  0.01

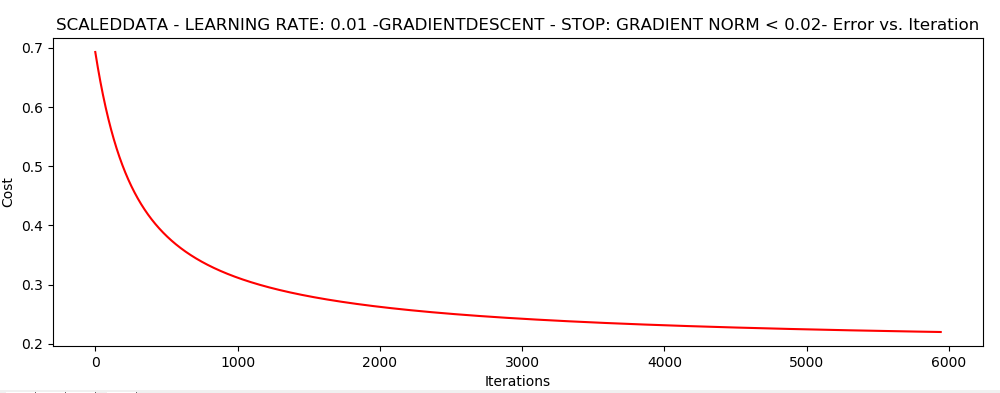




 ·　　我们发现，这里已经降到了0.3以下，所以说做数据预处理非常重要。

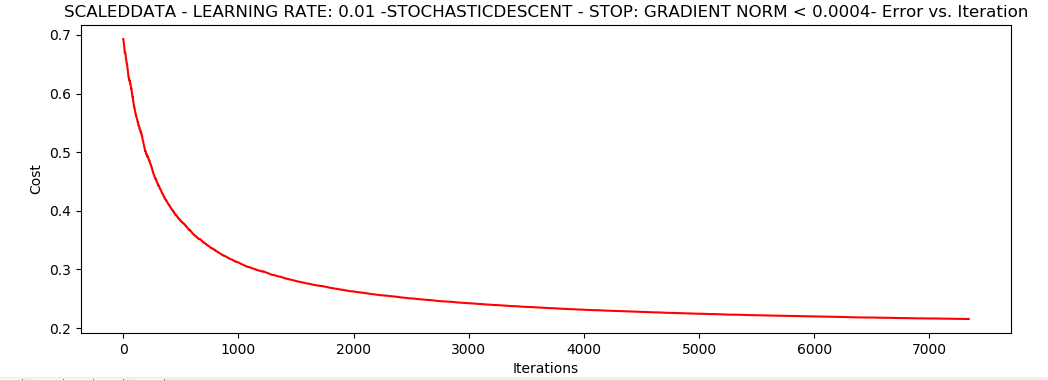
|  |  |
| --- | --- |
| 1  2  3  4 | scaled\_data = orig\_data.copy()  scaled\_data[:, 1:3] = pp.scale(orig\_data[:, 1:3])  # runExpe(scaled\_data, theta, n, STOP\_ITER, thresh=5000, alpha=0.01)  runExpe(scaled\_data, theta, n, STOP\_GRAD, thresh=0.02, alpha=0.01) |

 　　我们修改一下，根据梯度变换停止，发现更多的迭代次数会使得损失下降的更多



|  |  |
| --- | --- |
| 1 | runExpe(scaled\_data, theta, 1, STOP\_GRAD, thresh=0.002/5, alpha=0.01) |

 　　随机梯度下降更快，但是我们需要迭代的次数也需要更多，所以还是用 batch的比较合适。



 　　精度

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | # 设置阈值  def predict(x, theta):      return [1 if x >= 0.5 else 0 for x in model(X, theta)]      if \_\_name\_\_ == '\_\_main\_\_':      # 设定迭代次数   选择的梯度下降方法是基于所有样本的      n = 100      scaled\_data = orig\_data.copy()      scaled\_X = scaled\_data[:, :3]      y = scaled\_data[:, 3]      predictions = predict(scaled\_X, theta)      correct = [1 if ((a == 1 and b == 1) or (a == 0 and b == 0)) else 0 for a, b in zip(predictions, y)]      accuracy = (sum(map(int, correct)) % len(correct))      print('accuracy = {0}%'.format(accuracy))      print(len(correct)) |

 　　不知道为什么，我求出来的精度不高。。。。百分之60。

### **11，总结**

　　Logistic回归的目的是寻找一个非线性函数Sigmoid的最佳拟合参数，求解过程可以由最优化算法来完成。在最优化算法中，最常用的就是梯度上升算法，而梯度上升算法又可以简化为随机梯度上升算法。

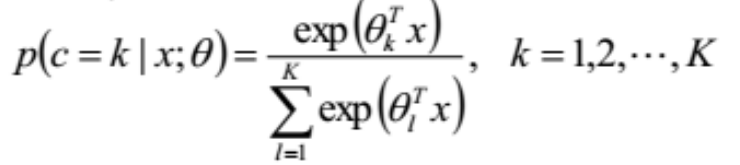
　　随机梯度上升算法与梯度上升算法的效果相当，但是占用更少的计算资源。此外，随机梯度上升是一个在线算法，它可以在新数据到来时就完成参数更新，不需要重新读取整个数据集来进行批处理运算。

　　机器学习的一个重要问题就是如何处理缺失数据，这个问题没有标准答案，取决于实际应用中的需求。

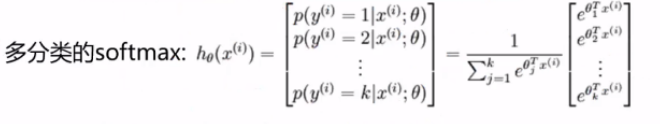
### **12，推广**

　　前面也说了Logistic回归模型主要用于二分类，那么下面说一下多分类问题中的推广——softmax回归。

　　softmax与Logistic回归的主要区别就是，Logistic处理二分类问题，只有一组权重参数θ，而softmax处理多分类问题，如果有k个类别，那么softmax就有k组权值参数。每组权值对应一种分类，通过k组权值求解出样本数据对应每个类别的概率，最后取概率最大的类别作为该数据的分类结果，它的概率函数为：



　　softmax函数经常用于神经网络的最后一层，用于对神经网络已经处理好的特征进行分类。



　　总结来说：逻辑回归真的真的很好用！

## **基于Sklearn构建Logistic回归分类器**

　　下面让我们看一下Sklearn的Logistic回归分类器

　　英文的Sklearn文档地址：[请点击我](http://scikit-learn.org/dev/modules/generated/sklearn.linear_model.LogisticRegression.html" \l "sklearn.linear_model.LogisticRegression" \t "https://www.cnblogs.com/wj-1314/p/_blank)

**sklearn.linear\_model模块提供了很多模型供我们使用，比如Logistic回归、Lasso回归、贝叶斯脊回归等，可见需要学习的东西还有很多很多。本次，我们使用LogisticRegressioin。**

### ****1，LogisticRegression****

　　让我们先看一下LogisticRegression这个函数，一共有14个参数

参数说明如下：

**penalty**：惩罚项，str类型，可选参数为l1和l2，默认为l2。用于指定惩罚项中使用的规范。newton-cg、sag和lbfgs求解算法只支持L2规范。L1G规范假设的是模型的参数满足拉普拉斯分布，L2假设的模型参数满足高斯分布，所谓的范式就是加上对参数的约束，使得模型更不会过拟合(overfit)，但是如果要说是不是加了约束就会好，这个没有人能回答，只能说，加约束的情况下，理论上应该可以获得泛化能力更强的结果。  
　　**dual**：对偶或原始方法，bool类型，默认为False。对偶方法只用在求解线性多核(liblinear)的L2惩罚项上。当样本数量>样本特征的时候，dual通常设置为False。  
　　**tol**：停止求解的标准，float类型，默认为1e-4。就是求解到多少的时候，停止，认为已经求出最优解。  
　　**c**：正则化系数λ的倒数，float类型，默认为1.0。必须是正浮点型数。像SVM一样，越小的数值表示越强的正则化。  
　　**fit\_intercept**：是否存在截距或偏差，bool类型，默认为True。如果使用中心化的数据，可以考虑设置为False,不考虑截距。注意这里是考虑，一般还是要考虑截距  
　　**intercept\_scaling**：仅在正则化项为”liblinear”，且fit\_intercept设置为True时有用。float类型，默认为1。  
　　**class\_weight**：用于标示分类模型中各种类型的权重，可以是一个字典或者balanced字符串，默认为不输入，也就是不考虑权重，即为None。如果选择输入的话，可以选择balanced让类库自己计算类型权重，或者自己输入各个类型的权重。举个例子，比如对于0,1的二元模型，我们可以定义class\_weight={0:0.9,1:0.1}，这样类型0的权重为90%，而类型1的权重为10%。如果class\_weight选择balanced，那么类库会根据训练样本量来计算权重。某种类型样本量越多，则权重越低，样本量越少，则权重越高。当class\_weight为balanced时，类权重计算方法如下：n\_samples / (n\_classes \* np.bincount(y))。n\_samples为样本数，n\_classes为类别数量，np.bincount(y)会输出每个类的样本数，例如y=[1,0,0,1,1],则np.bincount(y)=[2,3]。  
　　　　那么class\_weight有什么作用呢？ 在分类模型中，我们经常会遇到两类问题：

* 1，第一种是误分类的代价很高。比如对合法用户和非法用户进行分类，将非法用户分类为合法用户的代价很高，我们宁愿将合法用户分类为非法用户，这时可以人工再甄别，但是却不愿将非法用户分类为合法用户。这时，我们可以适当提高非法用户的权重。
* 2，第二种是样本是高度失衡的，比如我们有合法用户和非法用户的二元样本数据10000条，里面合法用户有9995条，非法用户只有5条，如果我们不考虑权重，则我们可以将所有的测试集都预测为合法用户，这样预测准确率理论上有99.95%，但是却没有任何意义。这时，我们可以选择balanced，让类库自动提高非法用户样本的权重。提高了某种分类的权重，相比不考虑权重，会有更多的样本分类划分到高权重的类别，从而可以解决上面两类问题。

**random\_state**：随机数种子，int类型，可选参数，默认为无，仅在正则化优化算法为sag,liblinear时有用。  
　　**solver**：优化算法选择参数，只有五个可选参数，即newton-cg,lbfgs,liblinear,sag,saga。默认为liblinear。solver参数决定了我们对逻辑回归损失函数的优化方法，有四种算法可以选择，分别是：

* **liblinear**：使用了开源的liblinear库实现，内部使用了坐标轴下降法来迭代优化损失函数。
* **lbfgs**：拟牛顿法的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
* **newton-cg**：也是牛顿法家族的一种，利用损失函数二阶导数矩阵即海森矩阵来迭代优化损失函数。
* **sag**：即随机平均梯度下降，是梯度下降法的变种，和普通梯度下降法的区别是每次迭代仅仅用一部分的样本来计算梯度，适合于样本数据多的时候。
* **saga**：线性收敛的随机优化算法的的变重。

总结：

* liblinear适用于小数据集，而sag和saga适用于大数据集因为速度更快。
* 对于多分类问题，只有newton-cg,sag,saga和lbfgs能够处理多项损失，而liblinear受限于一对剩余(OvR)。啥意思，就是用liblinear的时候，如果是多分类问题，得先把一种类别作为一个类别，剩余的所有类别作为另外一个类别。一次类推，遍历所有类别，进行分类。
* newton-cg,sag和lbfgs这三种优化算法时都需要损失函数的一阶或者二阶连续导数，因此不能用于没有连续导数的L1正则化，只能用于L2正则化。而liblinear和saga通吃L1正则化和L2正则化。
* 同时，sag每次仅仅使用了部分样本进行梯度迭代，所以当样本量少的时候不要选择它，而如果样本量非常大，比如大于10万，sag是第一选择。但是sag不能用于L1正则化，所以当你有大量的样本，又需要L1正则化的话就要自己做取舍了。要么通过对样本采样来降低样本量，要么回到L2正则化。
* 从上面的描述，大家可能觉得，既然newton-cg, lbfgs和sag这么多限制，如果不是大样本，我们选择liblinear不就行了嘛！错，因为liblinear也有自己的弱点！我们知道，逻辑回归有二元逻辑回归和多元逻辑回归。对于多元逻辑回归常见的有one-vs-rest(OvR)和many-vs-many(MvM)两种。而MvM一般比OvR分类相对准确一些。郁闷的是liblinear只支持OvR，不支持MvM，这样如果我们需要相对精确的多元逻辑回归时，就不能选择liblinear了。也意味着如果我们需要相对精确的多元逻辑回归不能使用L1正则化了。

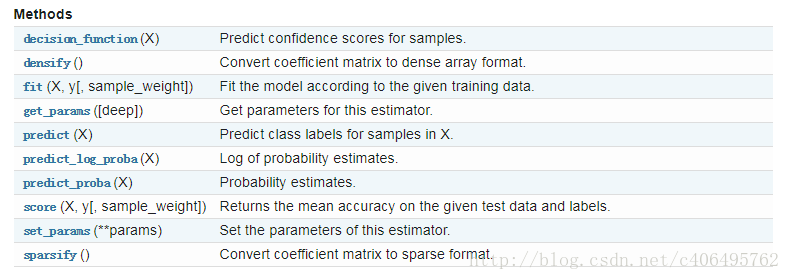
**max\_iter**：算法收敛最大迭代次数，int类型，默认为10。仅在正则化优化算法为newton-cg, sag和lbfgs才有用，算法收敛的最大迭代次数。  
**multi\_class**：分类方式选择参数，str类型，可选参数为ovr和multinomial，默认为ovr。ovr即前面提到的one-vs-rest(OvR)，而multinomial即前面提到的many-vs-many(MvM)。如果是二元逻辑回归，ovr和multinomial并没有任何区别，区别主要在多元逻辑回归上。

##### **OvR和MvM有什么不同？**

1. OvR的思想很简单，无论你是多少元逻辑回归，我们都可以看做二元逻辑回归。具体做法是，对于第K类的分类决策，我们把所有第K类的样本作为正例，除了第K类样本以外的所有样本都作为负例，然后在上面做二元逻辑回归，得到第K类的分类模型。其他类的分类模型获得以此类推。
2. 而MvM则相对复杂，这里举MvM的特例one-vs-one(OvO)作讲解。如果模型有T类，我们每次在所有的T类样本里面选择两类样本出来，不妨记为T1类和T2类，把所有的输出为T1和T2的样本放在一起，把T1作为正例，T2作为负例，进行二元逻辑回归，得到模型参数。我们一共需要T(T-1)/2次分类。
3. 可以看出OvR相对简单，但分类效果相对略差（这里指大多数样本分布情况，某些样本分布下OvR可能更好）。而MvM分类相对精确，但是分类速度没有OvR快。如果选择了ovr，则4种损失函数的优化方法liblinear，newton-cg,lbfgs和sag都可以选择。但是如果选择了multinomial,则只能选择newton-cg, lbfgs和sag了。

**verbose**：日志冗长度，int类型。默认为0。就是不输出训练过程，1的时候偶尔输出结果，大于1，对于每个子模型都输出。  
　　warm\_start：热启动参数，bool类型。默认为False。如果为True，则下一次训练是以追加树的形式进行（重新使用上一次的调用作为初始化）。  
　　**n\_jobs**：并行数。int类型，默认为1。1的时候，用CPU的一个内核运行程序，2的时候，用CPU的2个内核运行程序。为-1的时候，用所有CPU的内核运行程序。

除此之外，LogisticRegression也有一些方法供我们使用：



有一些方法和MultinomialNB的方法都是类似的，因此不再累述。

### **2，编写代码**

了解了基本知识点，我们就可以编写Sklearn分类器的代码了，代码如下：

#\_\*\_ codingLutf-8\_\*\_

from sklearn.linear\_model import LogisticRegression

def colicSklearn(filetrain,filetest):

    frTrain = open(filetrain)

    frTest = open(filetest)

    trainingSet = []

    trainingLabels = []

    testSet = []

    testLabels = []

    for line in frTrain.readlines():

        currLine = line.strip().split('\t')

        lineArr = []

        for i in range(len(currLine)-1):

            lineArr.append(float(currLine[i]))

        trainingSet.append(lineArr)

        trainingLabels.append(float(currLine[-1]))

    for line in frTest.readlines():

        currLine = line.strip().split('\t')

        lineArr = []

        for i in range(len(currLine)-1):

            lineArr.append(float(currLine[i]))

        testSet.append(lineArr)

        testLabels.append(float(currLine[-1]))

    classifier = LogisticRegression(solver='liblinear',max\_iter=20).fit(trainingSet,trainingLabels)

    test\_accurcy = classifier.score(testSet,testLabels)\*100

    print("正确率为%s%%"%test\_accurcy)

if \_\_name\_\_ == '\_\_main\_\_':

    filetrain = 'horseColicTraining.txt'

    filetest = 'horseColicTest.txt'

    colicSklearn(filetrain,filetest)

　　执行结果如下：

|  |  |
| --- | --- |
| 1 | 正确率为73.13432835820896% |

　　可以看到，正确率又搞了，更改solver参数，比如设置为sag，使用随机平均梯度下降算法，看一看效果，你会发现：正确率高了，但是发出了警告，发出警告是因为算法还没有收敛，更改迭代次数即可。

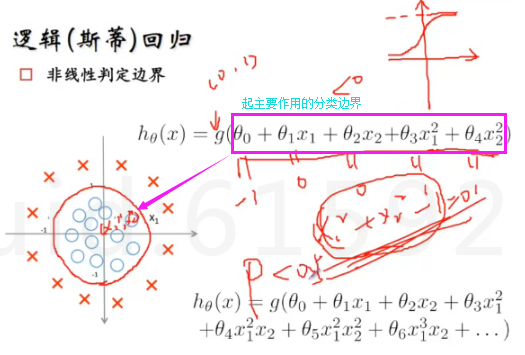
 当我们迭代到5000的时候，就不会报错了，所以说，对于小数据集，sag算法需要迭代上千次才收敛，而liblinear只需要不到10次。

　　所以，我们需要根据数据集情况，选择最优化算法。

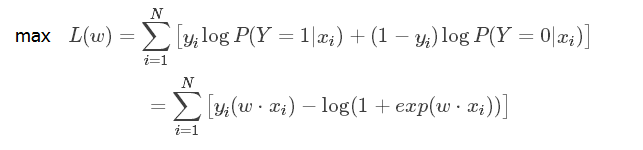
### **3，sklearn利用LR模型进行三分类的原理及其代码**

　　首先，LR将线性模型利用Sigmoid函数进一步做了非线性映射。将分类超平面两侧的正负样本点通过压缩函数转化成了以 0.5 为分类的两类：类别0 和类别1。

　　在上面我们讲述了线性边界与LR分布函数（即Sigmoid函数）的映射对应关系；同样，对于非线性判定边界Sigmoid函数也使用，如下：



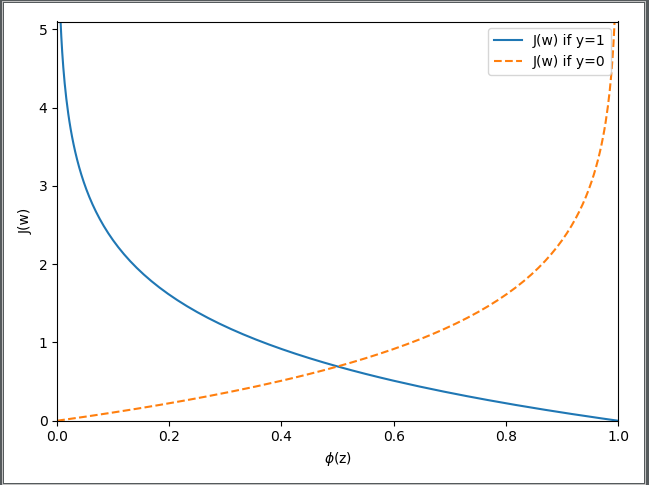
　　然后，再去考虑如何去获得模型参数，就是判定边界的参数怎么获得，这里是利用MLE进行求解的，具体求解过程可以百度。



　　代码如下：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35 | #\_\*\_coding:utf-8\_\*\_  import numpy as np  import matplotlib.pyplot as plt    def sigmoid(z):      return 1.0/(1.0 + np.exp(-z))    z = np.arange(-7, 7, 0.1)  print(z)  phi\_z = sigmoid(z)    def cost\_1(z):      return -np.log(sigmoid(z))    def cost\_0(z):      return -np.log(1-sigmoid(z))      z = np.arange(-10, 10, 0.1)  phi\_z = sigmoid(z)    c1 = [cost\_1(x) for x in z]  plt.plot(phi\_z, c1, label='J(w) if y=1')    c0 = [cost\_0(x) for x in z]  plt.plot(phi\_z, c0, linestyle='--', label='J(w) if y=0')    plt.ylim(0.0, 5.1)  plt.xlim([0, 1])  plt.xlabel('$\phi$(z)')  plt.ylabel('J(w)')  plt.legend(loc='best')  plt.tight\_layout()  # plt.savefig('./figures/log\_cost.png', dpi=300)  plt.show() |

　　结果如下：



　　最后，考虑LR进行三分类（多分类）时，是特征的线性组合和Sigmoid函数符合的函数进行概率计算和分类的。

from IPython.display import Image

# Added version check for recent scikit-learn 0.18 checks

from distutils.version import LooseVersion as Version

from sklearn import \_\_version\_\_ as sklearn\_version

from sklearn import datasets

import numpy as np

iris = datasets.load\_iris()

# http://scikit-learn.org/stable/auto\_examples/datasets/plot\_iris\_dataset.html

X = iris.data[:, [2, 3]]

print(X.shape)

y = iris.target  # 取species列，类别

if Version(sklearn\_version) < '0.18':

    from sklearn.cross\_validation import train\_test\_split

else:

    from sklearn.model\_selection import train\_test\_split

# train\_test\_split方法分割数据集

X\_train, X\_test, y\_train, y\_test = train\_test\_split(

    X, y, test\_size=0.3, random\_state=0)

from sklearn.preprocessing import StandardScaler

sc = StandardScaler()  # 初始化一个对象sc去对数据集作变换

sc.fit(X\_train)  # 用对象去拟合数据集X\_train，并且存下来拟合参数

X\_train\_std = sc.transform(X\_train)

X\_test\_std = sc.transform(X\_test)

from sklearn.linear\_model import LogisticRegression

def sigmoid(z):

    return 1.0 / (1.0 + np.exp(-z))

lr = LogisticRegression(C=1000.0, random\_state=0)

# http://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.LogisticRegression.html#sklearn.linear\_model.LogisticRegression

lr.fit(X\_train\_std, y\_train)

# 计算该预测实例点属于各类的概率

lr.predict\_proba(X\_test\_std[0, :].reshape(1, -1))

# Output:array([[  2.05743774e-11,   6.31620264e-02,   9.36837974e-01]])

# 验证predict\_proba的作用

c = lr.predict\_proba(X\_test\_std[0, :].reshape(1, -1))

# c[0, 0] + c[0, 1] + c[0, 2]

# Output:0.99999999999999989

# 查看lr模型的特征系数

lr = LogisticRegression(C=1000.0, random\_state=0)

# http://scikit-learn.org/stable/modules/generated/sklearn.linear\_model.LogisticRegression.html#sklearn.linear\_model.LogisticRegression

lr.fit(X\_train\_std, y\_train)

print(lr.coef\_)

# Output:[[-7.34015187 -6.64685581]

#        [ 2.54373335 -2.3421979 ]

#        [ 9.46617627  6.44380858]]

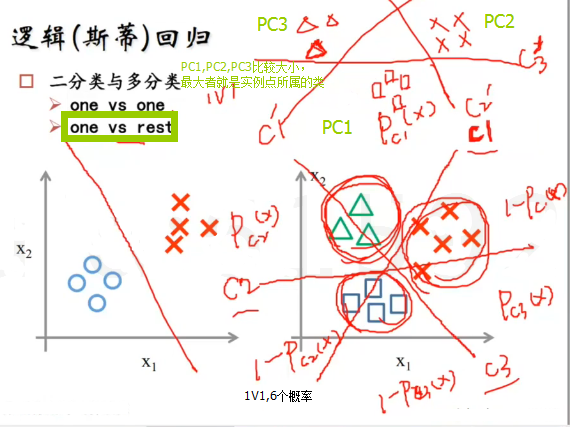
# 验证predict\_proba工作原理

Zz = np.dot(lr.coef\_, X\_test\_std[0, :].T) + lr.intercept\_

np.array(sigmoid(Zz)) / sum(np.array(sigmoid(Zz)))

# Output:array([  2.05743774e-11,   6.31620264e-02,   9.36837974e-01])

# 此结果就是预测实例点各类的概率



### **4，使用Sklearn的Logistic回归算法计算鸢尾花**

　　对鸢尾花进行分类代码：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17 | from sklearn import datasets  from numpy import \*  from sklearn.linear\_model import LogisticRegression  from sklearn.model\_selection import train\_test\_split      def colicSklearn():      iris = datasets.load\_iris()      X = iris.data      Y = iris.target      trainingSet,testSet,trainingLabels,testLabels = train\_test\_split(X,Y,test\_size=0.25,random\_state=40)      classifier = LogisticRegression(solver='sag', max\_iter=5000).fit(trainingSet, trainingLabels)      test\_accurcy = classifier.score(testSet, testLabels) \* 100      print("正确率为%s%%" % test\_accurcy)    if \_\_name\_\_  == '\_\_main\_\_':      colicSklearn() |

　　结果：

|  |  |
| --- | --- |
| 1 | 正确率为100.0% |

### **5，使用Sklearn的Logistic回归算法预测线性回归函数**

 代码：

|  |  |  |
| --- | --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59  60  61  62  63  64  65  66  67  68  69  70 | #\_\*\_coding:utf-8\_\*\_  '''  下面这个例子，从数据产生，到数据提取，数据标准化  模型训练和评估来说明各个API的调用过程  '''  print(\_\_doc\_\_)  from sklearn.preprocessing import StandardScaler  from sklearn.linear\_model import LinearRegression  from sklearn.model\_selection import train\_test\_split  import matplotlib.pyplot as plt  import numpy as np  import pandas as pd  # import matplotlib as mpl  import pylab as mpl    # 设置字符集，防止中文乱码  mpl.rcParams['font.sans-serif'] = [u'simHei']  mpl.rcParams['axes.unicode\_minus'] = False    # 定义目标函数通过改函数产生对应的y  # y = 1 \*x[0] + 2 \*x[1] + ... (n+1) \*x[n]  def l\_model(x):      params = np.arange(1, x.shape[-1] +1)      y = np.sum(params \*x) + np.random.randn(1) \* 0.1      return y    # 定义数据集  x = pd.DataFrame(np.random.rand(500,6))  # print(x)  y = x.apply(lambda x\_rows:pd.Series(l\_model(x\_rows)),axis=1)  # print(y)    # 划分数据集  x\_train, x\_test, y\_train, y\_test = train\_test\_split(x,y,test\_size=0.3,random\_state=2)    # 数据标准化  ss = StandardScaler()  x\_train\_s = ss.fit\_transform(x\_train)  x\_test\_s = ss.fit\_transform(x\_test)    # 输出下元数据的标准差和平均数  print(ss.scale\_)  print(ss.mean\_)    # 训练模型  lr = LinearRegression()  lr.fit(x\_train\_s , y\_train)  # 训练后的输入端模型系数，如果label有两个，即y值有两列，那么是一个2D的array  print(lr.coef\_)  # 截距  print(lr.intercept\_)    # 用模型预测,并计算得分  y\_predict = lr.predict(x\_test\_s)  test\_accuracy = lr.score(x\_test\_s, y\_test)    print("正确率为%s%%" % test\_accuracy)    # 预测值和实际值画图比较  t = np.arange(len(x\_test\_s))  # 建一个画布，facecolor是背景色  plt.figure(facecolor='W')  plt.plot(t, y\_test, 'r-', linewidth= 2, label = '真实值')  plt.plot(t, y\_predict, 'b-', linewidth= 1, label = '预测值')  # 显示图例，设置图例的位置  plt.legend(loc= 'upper left')  plt.title("线性回归预测真实值之间的关系", fontsize = 20)  # 加网格  plt.grid(b = True)  plt.show() | |
| 1  2  3  4  5  6  7  8  9  10 | 下面这个例子，从数据产生，到数据提取，数据标准化  模型训练和评估来说明各个API的调用过程      [0.28299035 0.3138177  0.29522932 0.30125841 0.28402977 0.28989605]  [0.53282546 0.51045318 0.48925819 0.5202345  0.4989613  0.54335117]  [[0.29202621 0.5787407  0.87280326 1.12786483 1.50043119 1.73271735]]  [10.38441274]    正确率为0.9706613427158242% |

