**Lab 9 logistic回归应用实践**

|  |  |  |  |
| --- | --- | --- | --- |
| 学号 | 姓名 | 年级 | 专业 |
| 18353070 | 谭嘉伟 | 2018 | 软件工程 |

1. 实验目的：
2. 学习并进一步了解Sigmoid函数等机器学习常用激活函数
3. 学习并实践用Sigmoid函数构建Logistic回归分类器
4. 学习并实践用梯度上升法找寻最佳参数
5. 学习并实践初步的最优化方法，掌握最优化方法的思想与基本理论知识
6. 进一步学习用Python可视化挖掘出的数据特性
7. 进一步熟悉提高用tensorflow实现机器学习算法的能力
8. 学习了解在数据集有部分缺失值时的处理方法
9. 实验环境：

1、硬件环境：PC 1.60GHz四核处理器 8.00G内存

2、软件环境：Windows10 64位操作系统

Python3.5.6 tensorflow1.4.0 Jupyter Notebook

1. 实验步骤：
2. 导入数据集并进行梯度上升

在ipython Notebook的单元格输入以下代码：

dataArr,labe1Mat=loadDataSet()

gradAscent(dataArr,labe1Mat)

运行单元格得到导入的数据集和梯度上升的结果

1. 分析数据，画出决策边界

运行以下代码：

weights=gradAscent(dataArr,labe1Mat)

plotBestFit(weights.getA())

可以画出决策边界

1. 训练算法：随机梯度上升

运行以下代码

dataArr,labe1Mat=loadDataSet()

weights=stocGradAscent0(np.array(dataArr), labe1Mat)

plotBestFit(weights)

可以得到随机梯度上升的最佳决策边界图像

上面的梯度上升算法每次需要遍历整个数据集，数据量大时时间复杂度较高，采用以下优化后的算法

dataArr,labe1Mat=loadDataSet()

weights=stocGradAscent1(np.array(dataArr), labe1Mat)

plotBestFit(weights)

运行可以得到优化后算法画出的决策边界图像

学习轮数更多时(500次迭代)：

weights=stocGradAscent1(np.array(dataArr), labe1Mat,500)

plotBestFit(weights)

1. 疝气病症预测病马的死亡率

准备数据：

训练集存放在命名为 horseColicTrainig.txt的文本文档中，测试

集则存放在命名为 horseColicTest.txt的文本文档中。数据文件的每一行代表一个数据样本，每一列代表一个特征，其中最后一列为类别标签，类别标签的取值只有两种，数据之间用 tab 键或空格分隔开。该数据集中有 30%的缺失值，两文件中缺失值已经用 0代替。

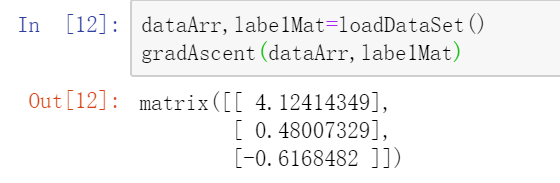
测试算法：利用Logistic回归分类

运行以下代码

multiTest()

1. 实验结果与分析：
2. 导入数据集并进行梯度上升

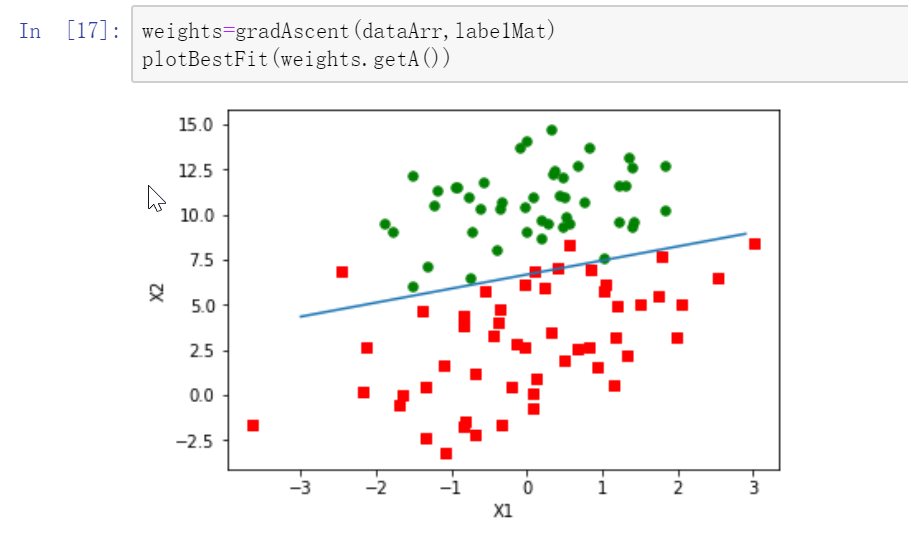
运行结果如下：



loadDataSet()函数读取数据集的txt文件，将数据集样本和标签分别导入到dataArr和labelMat列表中

然后调用gradAscent运用梯度上升法得到回归出的系数向量

1. 分析数据、画出决策边界

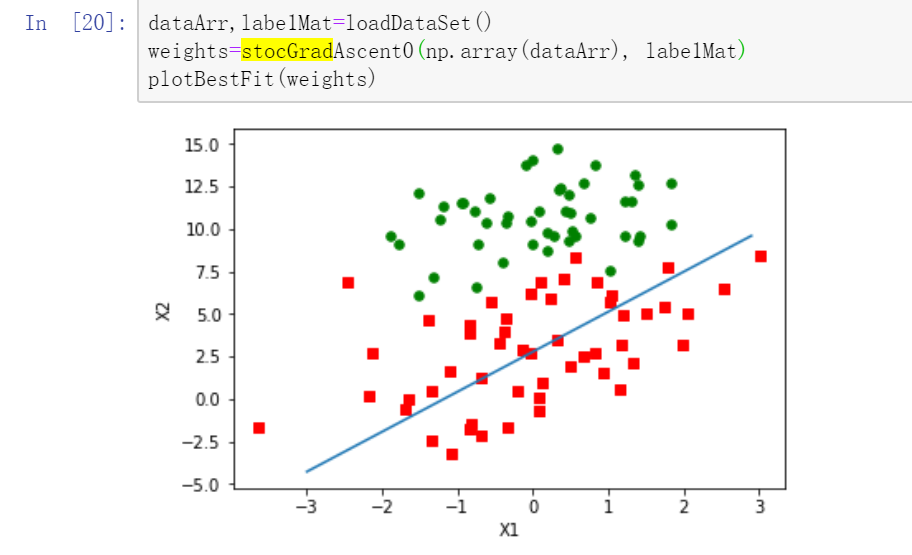


该图画出了数据对应的空间投影到数据前两维度形成的平面上，各样本数据点的分布情况

可以看出回归出的曲线将两类数据点比较准确地分割为了两个部分，分类效果看起来比较理想

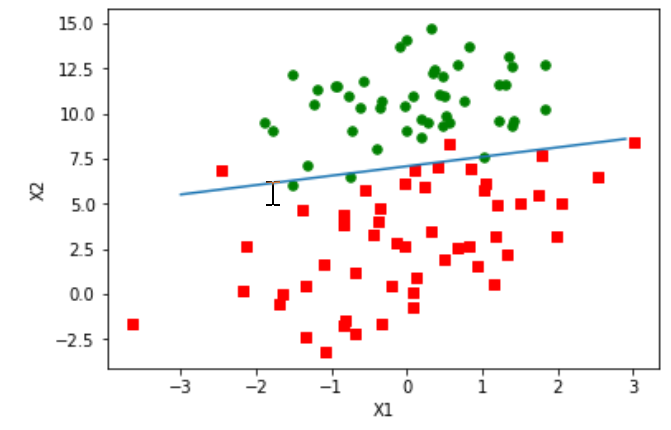
1. 训练算法：随机梯度上升

得到的随机梯度上升法图像如下



由于迭代次数较少，而且步长(alpha)较大，所以分类效果还不够好。肉眼可以看到决策边界没有很好地将两类点分开，该算法还有提升空间。

优化后算法的执行结果：



法

对比分析两个梯度上升函数的代码：

def stocGradAscent0(dataMatrix, classLabels):

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

alpha = 0.01 #梯度上升法迭代式中的 迭代步长

weights = np.ones(n) #将回归系数各分量初始化为1

for i in range(m): #对于所有数据样本

h = sigmoid(sum(dataMatrix[i]\*weights)) #得到sigmoid的输出值

error = classLabels[i] - h #使用向量减法计算误差

weights = weights + alpha \* error \* dataMatrix[i] #梯度上升法迭代式实现

return weights

该函数每次都取单独的一个数据来和权重进行点积来逻辑回归。相当于把每个数据逐一传入含有一个神经元的神经网络。每个样本数据只学习了一次，学习地不够多，因此学习效果还不够好。

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

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

weights = np.ones(n) #将回归系数各分量初始化为1

for j in range(maxIter):

dataIndex = list(range(m))

for i in range(m):

alpha = 4/(1.0+j+i)+0.0001 #【改进1】：alpha在每次迭代都会调整，会一定程度上缓解

#结果的周期性波动。同时，由于常数项的存在，虽然alpha会

#随着迭代次数不断减少，但永远都不会减少到 0。这保证了多

#次迭代后新数据仍然会有影响

randIndex = int(np.random.uniform(0,len(dataIndex)))#【改进2】：随机选取样本更新回归系数

#也可缓解结果的周期性波动

h = sigmoid(sum(dataMatrix[randIndex]\*weights)) #得到sigmoid的输出值

error = classLabels[randIndex] - h #使用向量减法计算误差

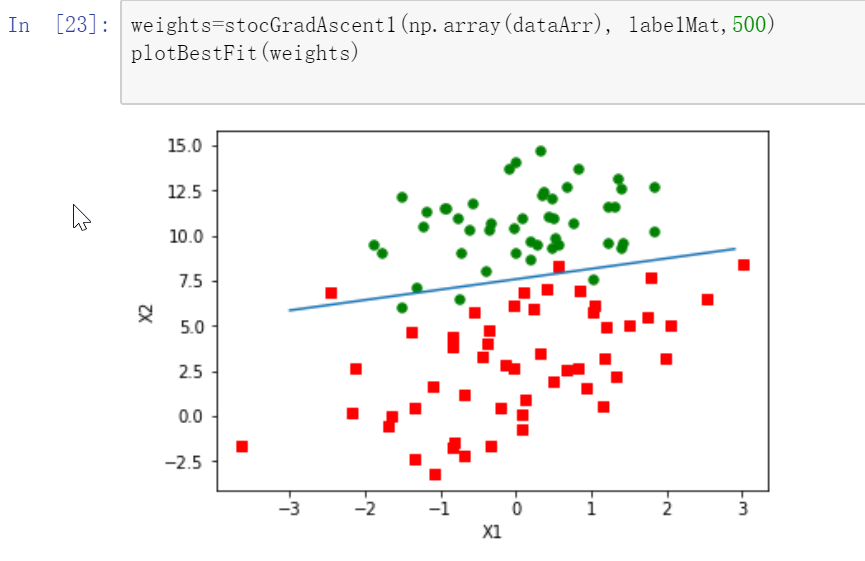
weights = weights + alpha \* error \* dataMatrix[randIndex] #梯度上升法迭代式实现

del(dataIndex[randIndex]) #将随机选择的样本从数据集中删除，避免影响下一次迭代

return weights

优化后的函数，迭代了maxIter次(缺省为150)。每次迭代都以随机顺序依次学习样本集所有数据，学习地更多让分类效果更好。且步长会随机改变，相当于神经网络中的一个dropout，使得逻辑回归效果更好，robust性更强。

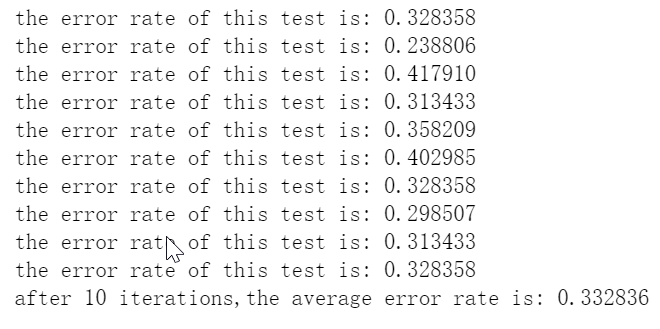
学习轮数更多时的决策边界如下：



分类已经达到了比较理想的状态，已经尽可能将数据点分开。上图显示了数据点投影到前两维自变量形成的平面上不能被简单地用一条直线分开，总会有同类数据点在直线的两侧，所以这已经是比较完美的分类状态了。

4、从疝气病症预测病马的死亡率

运行结果如下：



10次测试的平均错误率为33.2836%，错误率不高，但还有比较大的提升空间。可以之后调整参数尝试获得更好的分类效果

def colicTest():

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

#将训练集数据和对应标签存放到trainingSet和trainingLabels中

trainingSet = []; trainingLabels = []

for line in frTrain.readlines():

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

#训练算法，得到回归系数trainWeights

trainWeights = stocGradAscent1(np.array(trainingSet), trainingLabels, 1000)

#每行读取测试集数据，使用回归系数得到分类结果

errorCount = 0; numTestVec = 0.0

for line in frTest.readlines():

numTestVec += 1.0

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

lineArr =[]

for i in range(21):

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

#若分类结果不一致，则errorCount加一

if int(classifyVector(np.array(lineArr), trainWeights))!= int(currLine[21]):

errorCount += 1

errorRate = (float(errorCount)/numTestVec)

#打印此次测试的错误率

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

return errorRate

每次测试这段代码被执行

每一轮测试训练集和测试集数据先被导入变量，然后用1000轮优化后的梯度下降函数学习训练集，然后得到了训练出的权重向量。之后用逻辑回归分类每个向量，如果概率>0.5，认为类别为1，否则认为类别为0。随机猜的正确率为50%，逻辑回归在这里正确率达到66%，算是比较好的分类效果了。

1. 作业：
2. 将regres.py 中的“from numpy import \* ”用“import numpy as np”代替，修改其中对应的代码，使其能够正常执行

修改后的代码如下

#Regres.py

#-\*-coding:utf-8-\*-

import numpy as np

def loadDataSet():

dataMat = []; labelMat = []

fr = open('D:\\PUB128\\horseColic\\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]))

return dataMat,labelMat

def sigmoid(inX):

'''

输入：sigmoid函数的输入值

输出：sigmoid函数输出结果

描述：sigmoid函数实现

'''

return 1.0/(1.0+np.exp(-inX))

def gradAscent(dataMatIn, classLabels):

'''

输入：数据集，对应数据的类标签

输出：回归系数

描述：批量梯度上升法实现

'''

dataMatrix = np.mat(dataMatIn) #转为NumPy矩阵

labelMat = np.mat(classLabels).transpose() #转为NumPy矩阵

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

alpha = 0.001 #梯度上升法迭代式中的 迭代步长

maxIter = 500 #最大迭代次数

weights = np.ones((n,1)) #将回归系数各分量初始化为1

for k in range(maxIter):

h = sigmoid(dataMatrix\*weights) #得到sigmoid的输出值

error = (labelMat - h) #使用向量减法计算误差

weights = weights + alpha \* dataMatrix.transpose()\* error #梯度上升法迭代式实现

return weights

def stocGradAscent0(dataMatrix, classLabels):

'''

输入：数据集，对应数据的类标签

输出：回归系数

描述：随机梯度上升法实现

'''

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

alpha = 0.01 #梯度上升法迭代式中的 迭代步长

weights = np.ones(n) #将回归系数各分量初始化为1

for i in range(m): #对于所有数据样本

h = sigmoid(sum(dataMatrix[i]\*weights)) #得到sigmoid的输出值

error = classLabels[i] - h #使用向量减法计算误差

weights = weights + alpha \* error \* dataMatrix[i] #梯度上升法迭代式实现

return weights

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

'''

输入：数据集，对应数据的类标签

输出：回归系数

描述：stocGradAscent0函数的改进版本，降低了结果的

周期性波动，提高了结果的收敛速度

注：相对stocGradAscent0改进的部分用

'''

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

weights = np.ones(n) #将回归系数各分量初始化为1

for j in range(maxIter):

dataIndex = list(range(m))

for i in range(m):

alpha = 4/(1.0+j+i)+0.0001 #【改进1】：alpha在每次迭代都会调整，会一定程度上缓解

#结果的周期性波动。同时，由于常数项的存在，虽然alpha会

#随着迭代次数不断减少，但永远都不会减少到 0。这保证了多

#次迭代后新数据仍然会有影响

randIndex = int(np.random.uniform(0,len(dataIndex)))#【改进2】：随机选取样本更新回归系数

#也可缓解结果的周期性波动

h = sigmoid(sum(dataMatrix[randIndex]\*weights)) #得到sigmoid的输出值

error = classLabels[randIndex] - h #使用向量减法计算误差

weights = weights + alpha \* error \* dataMatrix[randIndex] #梯度上升法迭代式实现

del(dataIndex[randIndex]) #将随机选择的样本从数据集中删除，避免影响下一次迭代

return weights

def plotBestFit(weights):

import matplotlib.pyplot as plt

dataMat,labelMat=loadDataSet()

dataArr = np.array(dataMat)

n = np.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 = np.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()

def classifyVector(inX, weights):

'''

输入：样本数据，回归系数

输出：分类结果（0 或 1）

描述：使用优化后的回归系数对数据进行分类

'''

prob = sigmoid(sum(inX\*weights)) #计算sigmoid函数输出结果

if prob > 0.5: return 1.0

else: return 0.0

def colicTest():

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

#将训练集数据和对应标签存放到trainingSet和trainingLabels中

trainingSet = []; trainingLabels = []

for line in frTrain.readlines():

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

#训练算法，得到回归系数trainWeights

trainWeights = stocGradAscent1(np.array(trainingSet), trainingLabels, 1000)

#每行读取测试集数据，使用回归系数得到分类结果

errorCount = 0; numTestVec = 0.0

for line in frTest.readlines():

numTestVec += 1.0

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

lineArr =[]

for i in range(21):

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

#若分类结果不一致，则errorCount加一

if int(classifyVector(np.array(lineArr), trainWeights))!= int(currLine[21]):

errorCount += 1

errorRate = (float(errorCount)/numTestVec)

#打印此次测试的错误率

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

return errorRate

def multiTest(numTests=10):

errorSum=0.0

for k in range(numTests):

errorSum += colicTest()

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

multiTest()

1. 测试 3 种梯度下降函数的时间性能和效果。

梯度下降的三种形式有：随机梯度下降(Stochastic Gradient Descent，SGD),批量梯度下降(Batch Gradient Descent, BGD),小批量梯度下降(Mini-batch Gradient Descent, MBGD)

随机梯度下降SGD每次选择用一个样本来梯度下降，这样要达到和其他几种梯度下降方法相同的学习量，需要比较大的迭代次数。使用这种方法容易受到数据中噪声点影响，每一次不一定顺着往全局最优点的方向下降，准确度比较低，不一定能找到全局最优解。特别是噪声点较多的情况，这样训练得到分类器的性能不够高。

而批量梯度下降BGD每次使用所有数据来梯度下降，这样只需要几轮迭代，运算量就会比较大，使用这种方法时需控制迭代次数较小。而这种方法每次迭代都考虑到了所有数据的影响，可以比较好的找到最优解。

而批量梯度下降MBGD结合了上述两种方法的特点，可以发挥上述两种方法的优点。其迭代次数适中，且每次都能向比较好的方向下降，最后可以找到比较好的最优解。

由于数据集有缺失值，不能简单将缺失值看做0，而必须处理缺失值。

缺失值处理的基本思想有，剔除缺失值的数据/用算法填充缺失值

这里的缺失值处理方法是，对数据的每一维度，用所有非缺失值数据该维度的平均值填充确实数据值，然后进行随机梯度下降，代码如下：

from sklearn.linear\_model import SGDRegressor

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

def SGD(iters=100):

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

trainingSet = []

trainingLabels = []

for line in frTrain.readlines():#将训练集数据存入训练集列表

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

testSet=[]

testLabels=[]

for line in frTest.readlines():#将测试集数据存入测试集列表

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

lineArr=[]

for i in range(21):

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

testSet.append(lineArr)

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

for j in range(21):#对每一维度填充缺失值

sums=0

num=0

for i in range(len(trainingSet)):

if (trainingSet[i][j]!=0):

sums+=trainingSet[i][j]#取得该维度的非缺失值个数和非缺失值之和

num+=1

for i in range(len(trainingSet)):#填充训练集缺失值

if (trainingSet[i][j]==0):

trainingSet[i][j]=sums/num

for i in range(len(testSet)):#填充测试集缺失值

if (testSet[i][j]==0):

testSet[i][j]=sums/num

std = StandardScaler()#对数据进行标准化处理

std.fit(trainingSet)

trainingSetstd = std.transform(trainingSet)

testSetstd = std.transform(testSet)

sgd\_reg = SGDRegressor(max\_iter=iters,loss='squared\_loss')#进行梯度下降

sgd\_reg.fit(trainingSetstd, trainingLabels)

print("score; ",sgd\_reg.score(testSetstd, testLabels))#求出测试得分

weights=sgd\_reg.coef\_#得到权重向量

print("weights: ",weights)

xcord1 = []; ycord1 = []

xcord2 = []; ycord2 = []

for i in range(len(trainingSetstd)):#可视化 将数据投影到第三维度和第四维度构成的平面上

if int(trainingLabels[i])== 1:

xcord1.append(trainingSetstd[i][3]); ycord1.append(trainingSetstd[i][4])

else:

xcord2.append(trainingSetstd[i][3]); ycord2.append(trainingSetstd[i][4])

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 = np.arange(-1,1 ,0.02)

y = (-weights[3]\*x)/weights[4]

ax.plot(x, y)

plt.xlabel('X1'); plt.ylabel('X2');

plt.show()

SGD(100000)

SGD(1000000)

SGD(5000000)

运行结果：

100000次迭代：

score; 0.21056571894720277

weights: [ 1.55571223e-02 -3.38529795e-02 4.13850743e-02 -4.37493586e-02

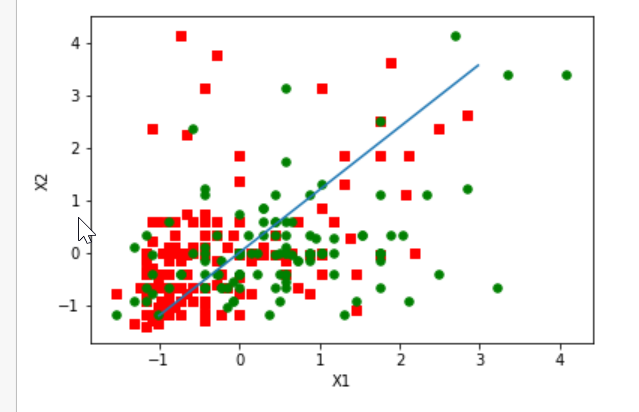
3.65183318e-02 -4.34219153e-02 -4.01600116e-02 -3.99426320e-02

7.96770968e-05 -6.95194578e-02 6.98776721e-03 -5.10447752e-02

3.59275314e-02 -3.06986851e-02 -2.73964278e-02 1.20067158e-02

-3.00440617e-02 -9.24362350e-02 3.21396146e-02 -2.47486906e-02

-1.40648366e-02]



1000000次迭代：

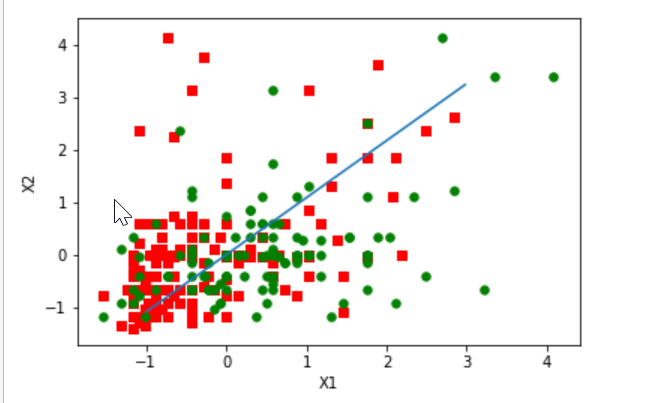
score; 0.2048243052765233

weights: [ 0.01369638 -0.03524564 0.03826624 -0.04021287 0.03698323 -0.04047714

-0.0385357 -0.03933907 0.00032446 -0.0693593 0.00964524 -0.04632561

0.03377799 -0.02943087 -0.03210639 0.01508105 -0.02702727 -0.09294935

0.03437537 -0.02709371 -0.01513602]



5000000次迭代：

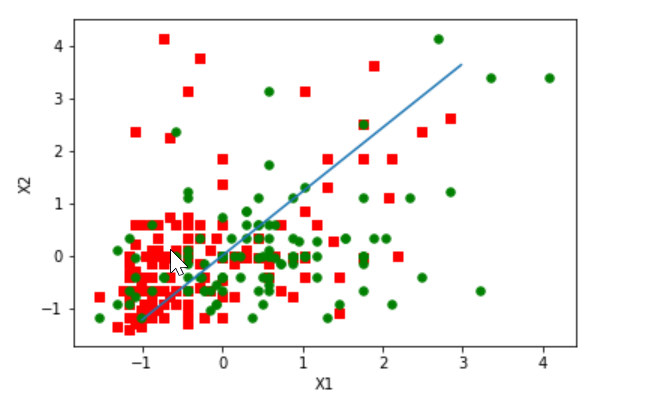
score; 0.2138673457752804

weights: [ 0.01703315 -0.03562803 0.03939966 -0.04089867 0.03352101 -0.0446441

-0.03766271 -0.04053325 0.00220569 -0.07137967 0.00990143 -0.04722968

0.03370403 -0.02856289 -0.03050253 0.01358399 -0.03004718 -0.09072558

0.03006485 -0.02342713 -0.01671548]



可见迭代次数从100000升至5000000时，得分没有明显提升，且得分来回波动。这体现了噪声数据和有缺失值数据对结果的影响，这限制了正确率的进一步提升

BGD的代码如下：

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

def BGD(iters=10):

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

trainingSet = []

trainingLabels = []

for line in frTrain.readlines():#将训练集数据存入训练集列表

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

testSet=[]

testLabels=[]

for line in frTest.readlines():#将测试集数据存入测试集列表

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

lineArr=[]

for i in range(21):

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

testSet.append(lineArr)

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

for j in range(21):#对每一维度填充缺失值

sums=0

num=0

for i in range(len(trainingSet)):

if (trainingSet[i][j]!=0):

sums+=trainingSet[i][j]#取得该维度的非缺失值个数和非缺失值之和

num+=1

for i in range(len(trainingSet)):#填充训练集缺失值

if (trainingSet[i][j]==0):

trainingSet[i][j]=sums/num

for i in range(len(testSet)):#填充测试集缺失值

if (testSet[i][j]==0):

testSet[i][j]=sums/num

std = StandardScaler()#对数据进行标准化处理

std.fit(trainingSet)

trainingSetstd = std.transform(trainingSet)

testSetstd = std.transform(testSet)

weights=stocGradAscent1(np.array(trainingSetstd),trainingLabels,iters)

print("weights: ",weights)

xcord1 = []; ycord1 = []

xcord2 = []; ycord2 = []

for i in range(len(trainingSet)):#可视化 将数据投影到第三维度和第四维度构成的平面上

if int(trainingLabels[i])== 1:

xcord1.append(trainingSetstd[i][3]); ycord1.append(trainingSetstd[i][4])

else:

xcord2.append(trainingSetstd[i][3]); ycord2.append(trainingSetstd[i][4])

error=0

for i in range(len(testSetstd)):

if int(testLabels[i])!=int(classifyVector(np.array(testSetstd[i]),weights)):

error+=1

print("error: ",error/len(testSetstd))

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 = np.arange(-1,3 ,0.02)

y = (weights[3]\*x)/weights[4]

ax.plot(x, y)

plt.xlabel('X1'); plt.ylabel('X2');

plt.show()

BGD(10)

BGD(35)

BGD(100)

迭代次数10次：

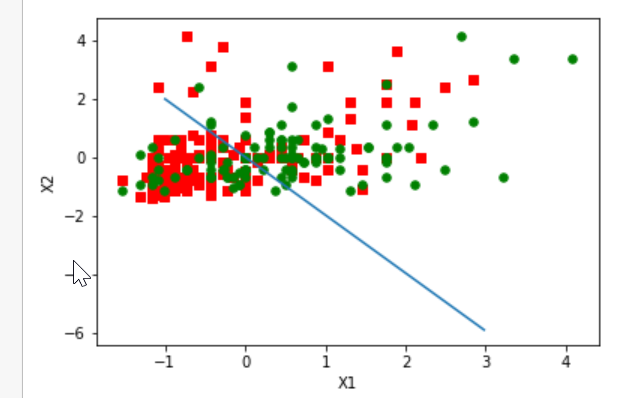
weights: [ 0.15907189 -0.51815374 0.66141366 -1.44359662 0.72952408 -0.82906687

-0.27805585 0.02756433 0.35182085 -0.10829358 0.14923666 -0.98552864

0.43798106 -0.55742546 -0.7415804 0.19389135 -0.55258728 -0.94842063

-0.23078903 -0.17195351 0.10083112]

error: 0.26865671641791045



迭代次数35次

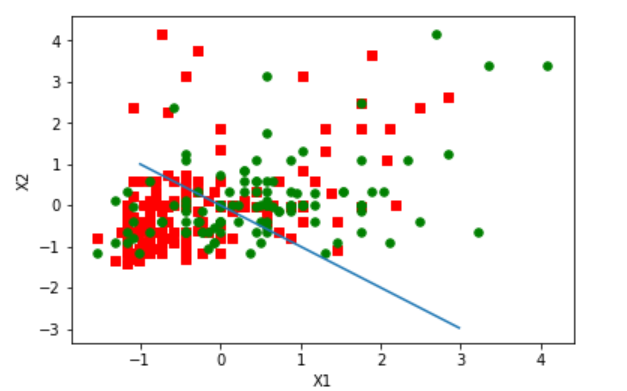
weights: [ 0.32773999 -0.25613676 0.39339218 -0.57175786 0.57245297 -0.09397023

-0.15235978 -0.12182328 -0.03766105 -0.51930989 -0.19518518 -0.39156312

0.21043984 -0.22395143 -0.60458314 -0.0482644 0.12914186 -0.78712149

0.20907746 -0.28338969 -0.11453368]

error: 0.2537313432835821



迭代次数100次

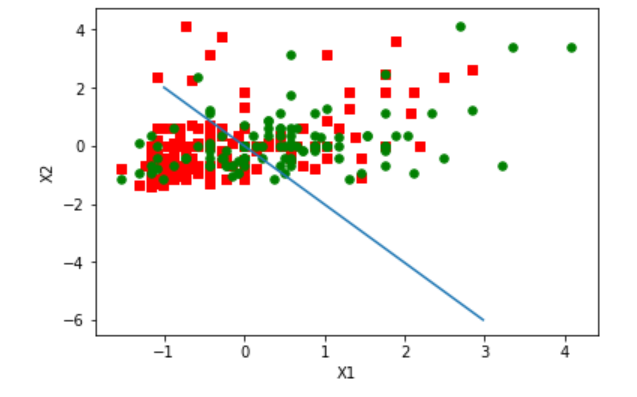
weights: [ 0.13634876 -0.37811806 0.41286054 -0.3561962 0.17710285 -0.24321358

-0.34360073 -0.15363346 -0.05195697 -0.6025301 0.00499647 -0.58224519

0.27977364 -0.09837473 -0.46509521 0.00279993 0.00245144 -0.26468032

-0.07930804 -0.34830785 -0.08096494]

error: 0.22388059701492538



MBGD代码如下:

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

def stocGradAscent2(dataMatrix, classLabels, maxIter=150):

'''

输入：数据集，对应数据的类标签

输出：回归系数

描述：stocGradAscent0函数的改进版本，降低了结果的

周期性波动，提高了结果的收敛速度

注：相对stocGradAscent0改进的部分用

'''

m,n = np.shape(dataMatrix) #得到数据集 行数（样本数）和 列数（特征数)

weights = np.ones(n) #将回归系数各分量初始化为1

for j in range(maxIter):

dataIndex = list(range(m))

for i in range(int(m\*0.1)):

alpha = 4/(1.0+j+i)+0.0001 #【改进1】：alpha在每次迭代都会调整，会一定程度上缓解

#结果的周期性波动。同时，由于常数项的存在，虽然alpha会

#随着迭代次数不断减少，但永远都不会减少到 0。这保证了多

#次迭代后新数据仍然会有影响

randIndex = int(np.random.uniform(0,len(dataIndex)))#【改进2】：随机选取样本更新回归系数

#也可缓解结果的周期性波动

h = sigmoid(sum(dataMatrix[randIndex]\*weights)) #得到sigmoid的输出值

error = classLabels[randIndex] - h #使用向量减法计算误差

weights = weights + alpha \* error \* dataMatrix[randIndex] #梯度上升法迭代式实现

del(dataIndex[randIndex]) #将随机选择的样本从数据集中删除，避免影响下一次迭代

return weights

def MBGD(iters=10):

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

trainingSet = []

trainingLabels = []

for line in frTrain.readlines():#将训练集数据存入训练集列表

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

testSet=[]

testLabels=[]

for line in frTest.readlines():#将测试集数据存入测试集列表

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

lineArr=[]

for i in range(21):

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

testSet.append(lineArr)

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

for j in range(21):#对每一维度填充缺失值

sums=0

num=0

for i in range(len(trainingSet)):

if (trainingSet[i][j]!=0):

sums+=trainingSet[i][j]#取得该维度的非缺失值个数和非缺失值之和

num+=1

for i in range(len(trainingSet)):#填充训练集缺失值

if (trainingSet[i][j]==0):

trainingSet[i][j]=sums/num

for i in range(len(testSet)):#填充测试集缺失值

if (testSet[i][j]==0):

testSet[i][j]=sums/num

std = StandardScaler()#对数据进行标准化处理

std.fit(trainingSet)

trainingSetstd = std.transform(trainingSet)

testSetstd = std.transform(testSet)

weights=stocGradAscent2(np.array(trainingSetstd),trainingLabels,iters)

print("weights: ",weights)

xcord1 = []; ycord1 = []

xcord2 = []; ycord2 = []

for i in range(len(trainingSet)):#可视化 将数据投影到第三维度和第四维度构成的平面上

if int(trainingLabels[i])== 1:

xcord1.append(trainingSetstd[i][3]); ycord1.append(trainingSetstd[i][4])

else:a

xcord2.append(trainingSetstd[i][3]); ycord2.append(trainingSetstd[i][4])

error=0

for i in range(len(testSetstd)):

if int(testLabels[i])!=int(classifyVector(np.array(testSetstd[i]),weights)):

error+=1

print("error: ",error/len(testSetstd))

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 = np.arange(-1,3 ,0.02)

y = (-weights[3]\*x)/weights[4]

ax.plot(x, y)

plt.xlabel('X1'); plt.ylabel('X2');

plt.show()

MBGD(100)

MBGD(350)

MBGD(1000)

100次迭代，每次随机取10%数据：

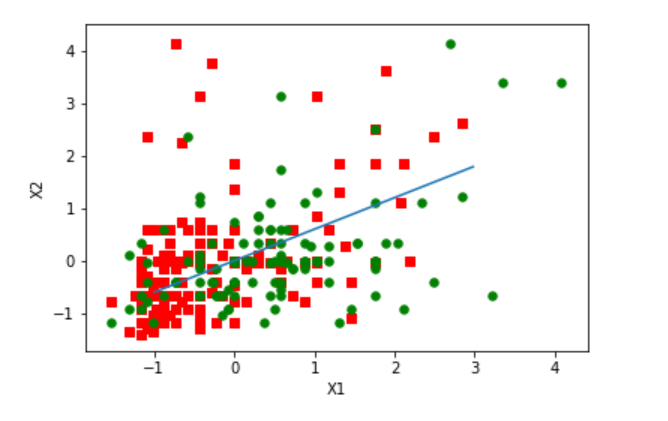
weights: [ 0.35251299 -0.32177716 0.41365969 -0.19778957 0.32917239 -0.21655141

-0.18364463 -0.54842391 0.19434432 -0.46801298 0.26050028 -0.32378385

0.28565917 -0.13709483 -0.49703367 0.24212517 -0.18919045 -0.87896659

0.1732924 -0.25920779 -0.32271488]

error: 0.19402985074626866



350次迭代，每次随机取10%数据：

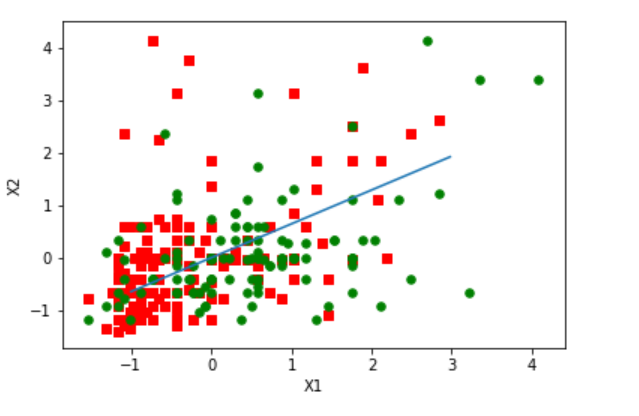
weights: [ 0.12976462 -0.13933287 0.21814097 -0.23122736 0.35843492 -0.21384522

-0.24847034 -0.25191705 0.12699322 -0.52088022 0.14980848 -0.23515414

0.19694288 -0.1361945 -0.04905084 -0.01838807 -0.39228455 -0.56956969

0.20202133 -0.19000215 -0.08873077]

error: 0.22388059701492538



1000次迭代，每次随机取10%数据：

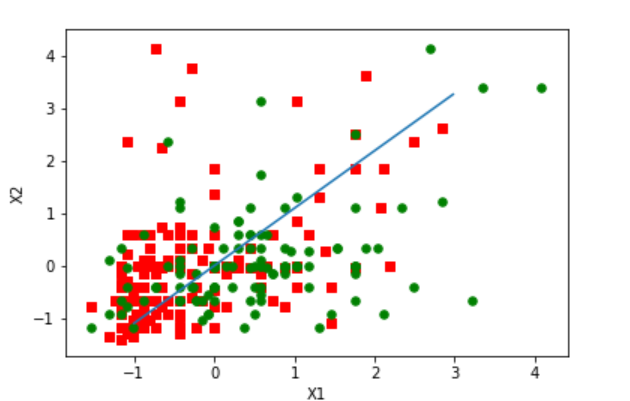
weights: [ 0.07865096 -0.20729786 0.18002586 -0.25178629 0.22977375 -0.17459945

-0.19991754 -0.1406801 0.13290246 -0.46373658 0.07314765 -0.31452461

0.17566624 -0.0966451 -0.15298973 0.04028463 -0.18860458 -0.69042476

0.26452496 -0.25871984 -0.11573154]

error: 0.22388059701492538



该算法也存在随着迭代次数增大，错误率波动的现象，需要进一步优化步长，以及通过参数搜索，来找到最优的参数。

1. 利用 sklearn 中的 LogisticRegression 分类器分类该数据集。

代码如下：

from sklearn.linear\_model import LogisticRegression

from sklearn.preprocessing import StandardScaler

import matplotlib.pyplot as plt

def LOGREG(iters=100):

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

trainingSet = []

trainingLabels = []

for line in frTrain.readlines():#将训练集数据存入训练集列表

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

testSet=[]

testLabels=[]

for line in frTest.readlines():#将测试集数据存入测试集列表

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

lineArr=[]

for i in range(21):

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

testSet.append(lineArr)

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

for j in range(21):#对每一维度填充缺失值

sums=0

num=0

for i in range(len(trainingSet)):

if (trainingSet[i][j]!=0):

sums+=trainingSet[i][j]#取得该维度的非缺失值个数和非缺失值之和

num+=1

for i in range(len(trainingSet)):#填充训练集缺失值

if (trainingSet[i][j]==0):

trainingSet[i][j]=sums/num

for i in range(len(testSet)):#填充测试集缺失值

if (testSet[i][j]==0):

testSet[i][j]=sums/num

std = StandardScaler()#对数据进行标准化处理

std.fit(trainingSet)

trainingSetstd = std.transform(trainingSet)

testSetstd = std.transform(testSet)

log\_reg = LogisticRegression(max\_iter=iters)#进行梯度下降

log\_reg.fit(trainingSetstd, trainingLabels)

print("score; ",log\_reg.score(testSetstd, testLabels))#求出测试得分

weights=log\_reg.coef\_#得到权重向量

print("weights: ",weights)

xcord1 = []; ycord1 = []

xcord2 = []; ycord2 = []

for i in range(len(trainingSet)):#可视化 将数据投影到第三维度和第四维度构成的平面上

if int(trainingLabels[i])== 1:

xcord1.append(trainingSetstd[i][3]); ycord1.append(trainingSetstd[i][4])

else:

xcord2.append(trainingSetstd[i][3]); ycord2.append(trainingSetstd[i][4])

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 = np.arange(-1,3 ,0.02)

y = (-weights[0,3]\*x)/weights[0,4]

ax.plot(x, y)

plt.xlabel('X1'); plt.ylabel('X2');

plt.show()

LOGREG(100)

LOGREG(1000)

LOGREG(5000)

迭代次数100次运行结果：

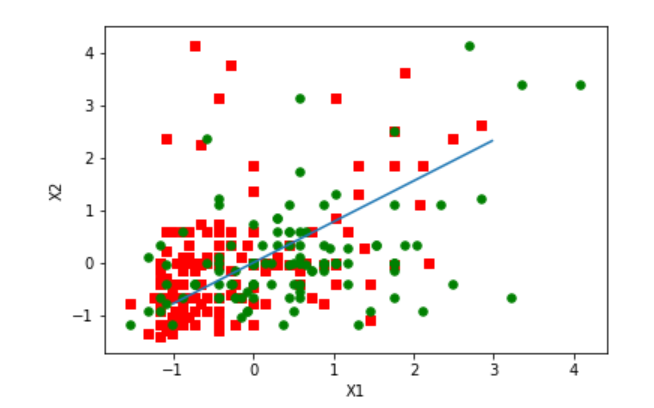
score; 0.7611940298507462

weights: [[ 0.14480527 -0.25858108 0.22157432 -0.13564336 0.17401842 -0.27158229

-0.18926273 -0.23259384 0.04511295 -0.46558005 0.0426536 -0.32683358

0.24701751 -0.16936233 -0.1558688 0.07653782 -0.22861975 -0.60942261

0.22848405 -0.15526269 -0.04915067]]



迭代次数1000次运行结果：

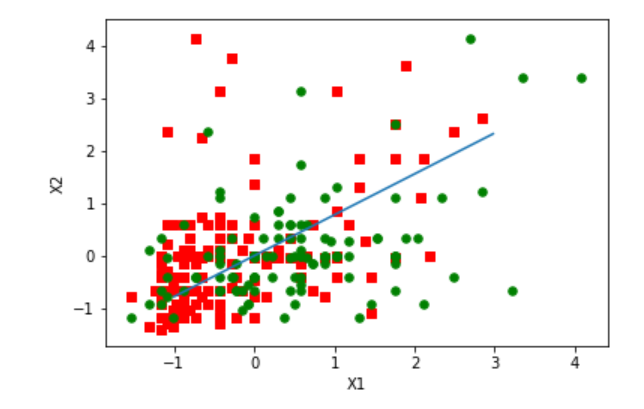
score; 0.7611940298507462

weights: [[ 0.14480527 -0.25858108 0.22157432 -0.13564336 0.17401842 -0.27158229

-0.18926273 -0.23259384 0.04511295 -0.46558005 0.0426536 -0.32683358

0.24701751 -0.16936233 -0.1558688 0.07653782 -0.22861975 -0.60942261

0.22848405 -0.15526269 -0.04915067]]



迭代次数5000次运行结果：

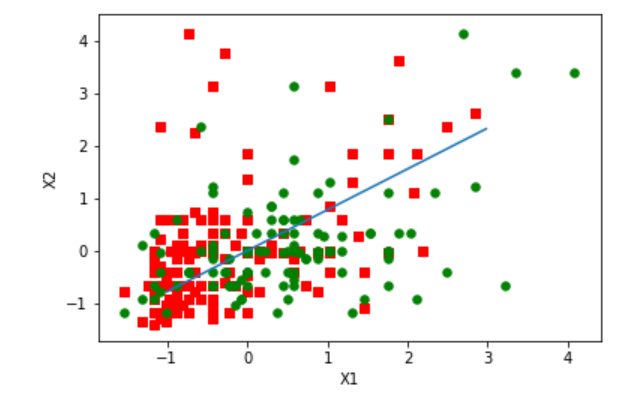
score; 0.7611940298507462

weights: [[ 0.14480527 -0.25858108 0.22157432 -0.13564336 0.17401842 -0.27158229

-0.18926273 -0.23259384 0.04511295 -0.46558005 0.0426536 -0.32683358

0.24701751 -0.16936233 -0.1558688 0.07653782 -0.22861975 -0.60942261

0.22848405 -0.15526269 -0.04915067]]



可见逻辑回归的分数达到0.76，有比较理想的分类效果，算法运行速度也较快。

1. 利用 tensorflow 实现 Logistic Regression 分类器分类该数据集。（扩展）

代码如下:

import tensorflow as tf

def tfLogReg(max\_iter):

frTrain = open('D:\\PUB128\\horseColic\\horseColicTraining.txt')#打开训练集文件

frTest = open('D:\\PUB128\\horseColic\\horseColicTest.txt') #打开测试集文件

trainingSet = []

trainingLabels = []

for line in frTrain.readlines():#将训练集数据存入训练集列表

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

lineArr =[]

for i in range(21):

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

trainingSet.append(lineArr)

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

testSet=[]

testLabels=[]

for line in frTest.readlines():#将测试集数据存入测试集列表

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

lineArr=[]

for i in range(21):

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

testSet.append(lineArr)

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

for j in range(21):#对每一维度填充缺失值

sums=0

num=0

for i in range(len(trainingSet)):

if (trainingSet[i][j]!=0):

sums+=trainingSet[i][j]#取得该维度的非缺失值个数和非缺失值之和

num+=1

for i in range(len(trainingSet)):#填充训练集缺失值

if (trainingSet[i][j]==0):

trainingSet[i][j]=sums/num

for i in range(len(testSet)):#填充测试集缺失值

if (testSet[i][j]==0):

testSet[i][j]=sums/num

std=StandardScaler()#对数据进行标准化处理

std.fit(trainingSet)

trainingSetstd=std.transform(trainingSet)

testSetstd=std.transform(testSet)

dataMat=np.mat(trainingSetstd).astype(np.float32)#shape=[n,21]

labelMat=np.mat(trainingLabels).transpose().astype(np.float32)#shape=[1,n]

n=dataMat.shape[0]

weight=tf.Variable(tf.zeros([21,1]))

bias=tf.Variable(tf.zeros([1,1]))

x\_=tf.placeholder(tf.float32,[None,21])

y\_=tf.placeholder(tf.float32,[None,1])

g=tf.matmul(x\_,weight)+bias#内积加偏置

hyp=tf.sigmoid(g) #传入sigmoid函数

cost=(y\_\*tf.log(hyp)+(1-y\_)\*tf.log(1 - hyp))/-n # logistic回归的对数损失函数

optimizer = tf.train.GradientDescentOptimizer(0.01)

train = optimizer.minimize(cost)# 最小化logistic回归的对数损失函数

init=tf.initialize\_all\_variables()

with tf.Session() as sess:

sess.run(init)

for niter in range(max\_iter):#迭代轮数，训练多少轮

for data,label in zip(dataMat,labelMat):#

sess.run(train, feed\_dict={x\_: data, y\_: label})

print(sess.run(weight).flatten(),sess.run(bias).flatten())

#cost\_val=sess.run(cost, {x\_: data, y\_: label})

#print('cost = ',cost\_val)

#print('weight = ',weight.eval(sess).flatten())

w=weight.eval(sess).flatten()

b=sess.run(bias).flatten()

error=0

for i in range(len(testSetstd)):

if int(testLabels[i])!=int(classifyVector(np.array(testSetstd[i]),w)+b):

error+=1

print("error: ",error/len(testSetstd))

xcord1 = []; ycord1 = []

xcord2 = []; ycord2 = []

for i in range(len(trainingSet)):#可视化 将数据正规化后投影到第三维度和第四维度构成的平面上

if int(trainingLabels[i])== 1:

xcord1.append(trainingSetstd[i][3]); ycord1.append(trainingSetstd[i][4])

else:

xcord2.append(trainingSetstd[i][3]); ycord2.append(trainingSetstd[i][4])

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 = np.arange(-1,1 ,0.02)

y = (b-w[3]\*x)/w[4]

ax.plot(x, y)

plt.xlabel('X1'); plt.ylabel('X2');

plt.show()

tfLogReg(10)

tfLogReg(20)

tfLogReg(40)

10次迭代：

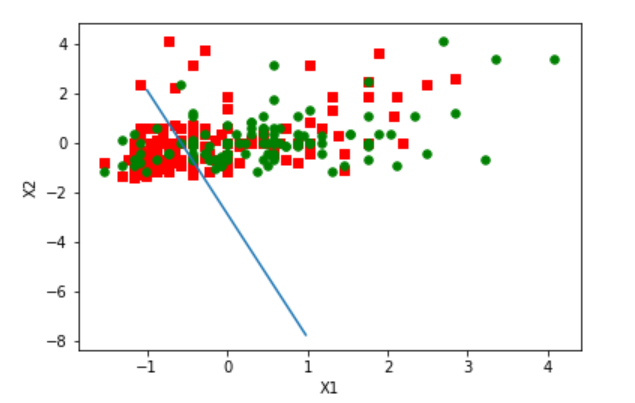
[ 0.0077714 -0.0026997 0.00182931 -0.01639554 -0.00328529 -0.0152642

-0.01796598 -0.01892537 -0.01449901 -0.01651487 -0.01389687 -0.01612194

-0.00010844 -0.00930663 -0.00168012 -0.00890162 -0.01138659 -0.01898703

0.00619274 -0.01160097 -0.00284627] [0.0094093]

error: 0.29850746268656714



20次迭代：

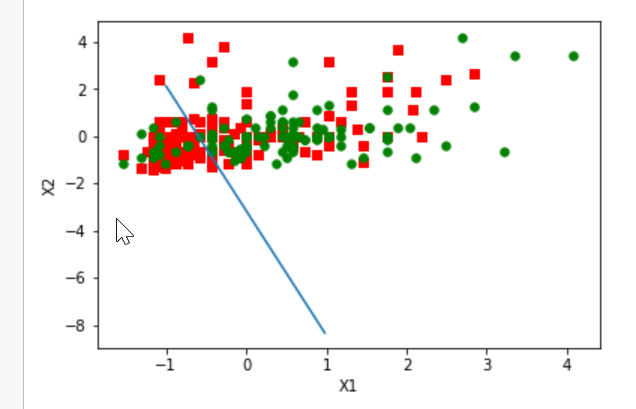
[ 0.01457255 -0.00518714 0.00384806 -0.03100945 -0.00587454 -0.02889598

-0.03391837 -0.03586965 -0.027243 -0.03142923 -0.02599684 -0.0304652

0.00010965 -0.01764863 -0.00341726 -0.01645188 -0.02144502 -0.0361907

0.01199576 -0.02197551 -0.00554199] [0.01858804]

error: 0.29850746268656714



40次迭代：

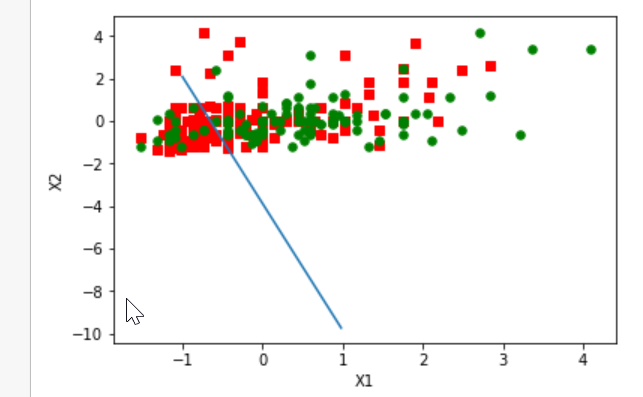
[ 0.02577025 -0.00963975 0.00832977 -0.05576714 -0.00933733 -0.05207057

-0.06076656 -0.06477869 -0.04833103 -0.05724841 -0.04572544 -0.05470734

0.00134625 -0.03191243 -0.0070059 -0.02820918 -0.03825954 -0.06610342

0.02259385 -0.03963985 -0.01052867] [0.0362883]

error: 0.29850746268656714



1. 实验总结：
2. 可用Python指令方便地进行读取文件数据集的处理

fr = open(str)可以读取str路径的文件内容到fr中

for line in fr.readlines()可以逐行取得文件内容，line表示文件中的一行，通过for循环迭代可以取得每一行内容

line.strip()可以将一行中开头和结尾的多余字符移除，line.strip().split()可以将一行中开头和结尾的多余字符移除后根据空白符分割字符串

这样就可以很方便地将数据集数据给分开成一个个整数导入变量

1. numpy.mat()可以将python中的嵌套列表转换为matrix类型。matrix类型可以直接用\*实现两个矩阵相乘的运算。numpy.shape()可以求出matrix的形状，得到行数和列数。matrix.transport()可以求出矩阵的转置。
2. fig = plt.figure()可以通过fig操控图片对象，ax = fig.add\_subplot(111)表示将画布分成1行1列，ax在这1行一列的第1块。类似的，ax = fig.add\_subplot(234)/ax = fig.add\_subplot(2,3,4)表示将画布分成2行3列，ax在这6块小画布中的从左到右从上到下第4块。如果参数有超过9的，则需要使用上面的第二种参数调用方式。
3. ax.scatter()可以用来在子图上画出散点图，可以指出每种点的颜色，这个方法很适合两维自变量和一维因变量(标签)数据的可视化。
4. plt.xlabel('X1')可以用来给图片的坐标轴添加标签，说明这个坐标轴对应的是哪一维数据
5. np.arange()可以返回一个ndarray，存放一个固定起点和终点的固定步长的序列。range()不支持步长为小数，返回类型是range object。而np.arange()支持返回小数
6. ax.plot(x,y)可以画出曲线图，可以用于画出权重向量，即逻辑回归的决策边界
7. Logistic回归相当于一个神经元的神经网络。Logistic可以处理二分类，根据回归返回的概率大小，判断数据属于哪一个类别
8. 数据集中噪声点和缺失值的处理都需要一定技巧。随机梯度下降可能会因为噪声点过多而性能无法达到比较高，不能找到全局最优解，这时需要每次选取一批数据进行梯度下降。而对于缺失值，可以考虑忽略还有缺失项的数据，或者用算法填充缺失项。中位数和平均值是常用的两种填充指标。
9. SGD为随机梯度下降，每次随机选取单个数据进行梯度下降。BGD为批量梯度下降，每次选取所有数据进行梯度下降。MBGD为小批量梯度下降，每次随机选取一定量的数据进行梯度下降，是SGD和BGD的折中方案，结合了SGD和BGD的优缺点。
10. 逻辑回归与神经网络对有噪声值、缺失值的数据集有比较好的训练效果，其会自动调整出比较合适的权重向量
11. tf.matmul可以实现tensorflow框架下的矩阵乘法/向量内积，而tf.transpose可以实现tensorflow框架下的矩阵转置。
12. zip(a,b)可以将a和b打包成元组列表。for data,label in zip(dataMat,labelMat)即可实现对dataMat,labelMat中第i项元素的逐一遍历
13. Flatten()可以将numpy对象转换为一个一维的列表
14. 在使用tensorflow框架时，运行会话前需要用init=tf.initialize\_all\_variables()进行初始化
15. optimizer = tf.train.GradientDescentOptimizer(0.01)可以实现tensorflow框架下的梯度下降，参数可以设定步长
16. train = optimizer.minimize(cost)可以实现tensorflow框架下对损失函数cost的优化，cost可以是自定义的函数形式
17. tensorflow适合于搭建神经网络，而逻辑回归相当于单个神经元的神经网络，因此搭建逻辑回归器也比较方便