MS328 Assignment2

周李韬 5180309104O7

2020 年 4 月 4 日

逻辑回归模型与实现

1. 学习一种优化算法，在你熟悉的编程语言中实现逻辑回归的算法；
2. 基于逻辑回归对某个真实数据进行分析。

1 数据处理

我们采用 Breast Cancer Wisconsin (Diagnostic) Data Set 数据集进行逻辑回归分类. 该数据集中采集了 569 位病人的胸部细胞特征信息. 胸部细胞共有十类特征, 如细胞半径, 灰度, 细胞面积等, 均为实数值. 每一类特征具有平均值, 标准差, 极值三个域. 即每位病人有 30 个数值信息. 每位病人被分类为患癌 (M=malignant) 和健康 (B=benign). 数据集中, 共有 357 个健康样本, 212 个患癌样本. 本实验中, 我们取十种特征的平均值作为样本特征进行回归分类.

下面我们读取对应数据.

[1]:
```
import numpy as np
import pandas as pd
import math
data = pd.read_csv('wdbc.data',header=None,index_col=0,usecols=[0,1]+[2+3*i for
↪i in range(10)])
data
```

[1]:

	1	2	5	8	11	14	17	20	23
0									
842302	M	17.99	1001.0	0.30010	0.07871	8.589	0.04904	0.03003	17.33
842517	M	20.57	1326.0	0.08690	0.05667	3.398	0.01308	0.01389	23.41
84300903	M	19.69	1203.0	0.19740	0.05999	4.585	0.04006	0.02250	25.53
84348301	M	11.42	386.1	0.24140	0.09744	3.445	0.07458	0.05963	26.50
84358402	M	20.29	1297.0	0.19800	0.05883	5.438	0.02461	0.01756	16.67

```
…       ..    …      …       …       …     …       …       …     …
926424    M  21.56  1479.0  0.24390  0.05623  7.673  0.02891  0.01114  26.40
926682    M  20.13  1261.0  0.14400  0.05533  5.203  0.02423  0.01898  38.25
926954    M  16.60   858.1  0.09251  0.05648  3.425  0.03731  0.01318  34.12
927241    M  20.60  1265.0  0.35140  0.07016  5.772  0.06158  0.02324  39.42
92751     B   7.76   181.0  0.00000  0.05884  2.548  0.00466  0.02676  30.37

                26      29
0
842302    0.16220  0.2654
842517    0.12380  0.1860
84300903  0.14440  0.2430
84348301  0.20980  0.2575
84358402  0.13740  0.1625
…            …      …
926424    0.14100  0.2216
926682    0.11660  0.1628
926954    0.11390  0.1418
927241    0.16500  0.2650
92751     0.08996  0.0000

[569 rows x 11 columns]
```

我们从数据集中提取自变量与因变量.

[2]:
```python
def read_sig(data):
    x = []
    y = []
    for line in data.values:
        x.append(line[1:])
        if line[0] == 'M':
            y.append([1])
        else:
            y.append([0])
    x = np.array(x)
    y = np.array(y)
    return x,y
```

由于十类特征各不相同, 我们需要对数据做一些处理. 经过尝试发现

$$x = \frac{x - \bar{x}}{x_{max} - x_{min}}$$

最适合逻辑回归模型的处理方式。首先，样本特征与样本均值的差可以保证数据可以均匀分布在 0 点两侧，便于计算。进一步，将差值与特征值域作商可以帮助我们在结论中得到各指标之间的相对关系。

[37]:
```
# def normalize2(x):
#     return ((x - x.min(axis=0))  / (x.max(axis=0) - x.min(axis=0)))
# def normalize3(x):
#     return (x  / (x.max(axis=0) ))
def normalize(x):
    return ((x-x.mean(axis=0)) /(x.max(axis=0)-x.min(axis=0)))
```

首先，我们用 python sklearn 库中的逻辑回归模型对数据进行测试。注意到 sklearn 中的求解算法是”lbfgs” 的近似算法，因此可能会和我们梯度下降、牛顿法的结果略有不同。sklearn 模块的逻辑回归系数结果如下所示。

[38]:
```
x1,y = read_sig(data)
x = normalize(x1)
from sklearn.linear_model import LogisticRegression
log_reg = LogisticRegression()
log_reg.fit(x,y[:,0])
print(log_reg.coef_)
print(log_reg.intercept_)
```

```
[[ 3.84162736  3.21032364  2.83241674 -1.17741758  1.81722502 -0.33158592
   0.10224375  3.14757253  2.2137123   4.90276183]]
[-0.74067841]
```

2 逻辑回归模型实现

2.1 梯度下降法

逻辑回归损失函数的梯度函数如下所示。

$$\nabla L(\beta) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{e^{X_i^T\beta}}{1+e^{X_i^T\beta}}X_i - Y_iX_i\right) = \frac{1}{m}\sum_{i=1}^{m}\left(\frac{1}{1+e^{-X_i^T\beta}}X_i - Y_iX_i\right)$$

为减小运算量，避免溢出，我们在梯度函数的实现上选择后一种形式。

[39]:
```
# 损失梯度函数
def logit_der1(x,y,b):
    x=np.array(x,dtype=np.float64)
    b=np.array(b,dtype=np.float64)
    dim = x.shape[0]
    sigmoid = 1/(1+np.exp(- np.dot(x,b)))
    return sum([(sigmoid[i] - y[i])*x[i] for i in range(dim)])/dim

# def logit_der1_fail(x,y,b):
#     dim = x.shape[0]
#     return sum([((np.exp(np.vdot(x[i],b))/(np.exp(np.vdot(x[i],b))+1))*x[i] +␣
 ↪y[i]*x[i]) for i in range(dim)])/dim
```

为便于观察，我们希望在迭代过程中记录损失函数值。同样，为简化运算，避免正数指数函数运算时的溢出问题，我们也对损失函数做一定变形，通过第二行的式子实现。

$$
\begin{aligned}
L(\beta) &= \frac{1}{m}\sum_{i=1}^{m}\left(\left(\ln(1+e^{X_i^T\beta}\right) - Y_i X_i^T\beta\right) \\
&= -\frac{1}{m}\sum_{i=1}^{m}\left(Y_i \ln\frac{1}{1+e^{-X_i^T\beta}} + (1-Y_i)\ln\left(1-\frac{1}{1+e^{-X_i^T\beta}}\right)\right)
\end{aligned}
$$

[40]:
```
# 损失函数
def logit_loss(x,y,b):
    x=np.array(x,dtype=np.float64)
    b=np.array(b,dtype=np.float64)
    sigmoid = 1/(1+np.exp(- np.dot(x,b)))
    dim = x.shape[0]
    return np.sum((- y*np.log(sigmoid)- (1-y)*np.log(1-sigmoid)))/dim

# def logit_loss_fail(x,y,b):
#     dim = x.shape[0]
#     return sum([math.log(math.exp(np.vdot(x[i],b))+1) - y[i] * np.
 ↪vdot(x[i],b) for i in range(dim)])/dim
```

我们首先尝试选择固定步长进行迭代运算。

[41]:
```
def logit_gradient_descent(x,y,init,epoch):
    b = np.ones(x.shape[0])
```

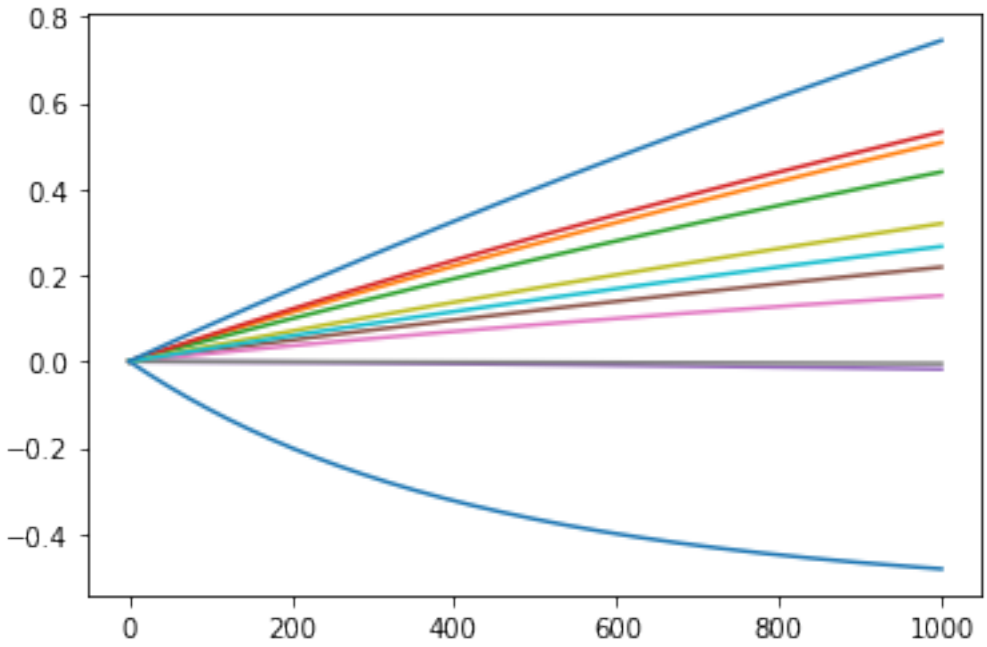
```
    x1 = np.c_[b,x]                 # 为自变量加上系数列
    beta = np.concatenate((init.reshape(1,init.shape[0]),np.zeros((epoch,x1.
↪shape[1])))) # 创建迭代记录数组
    loss = np.zeros(epoch+1)
    loss[0] = logit_loss(x1,y,beta[0])
    for iter in range(epoch):
        beta[iter+1] = beta[iter] - 0.01 * logit_der1(x1,y,beta[iter])
        loss[iter+1] = logit_loss(x1,y,beta[iter+1])
    return beta,loss
```

取步长为 0.01，初始值为 0，迭代 1000 次，我们发现由于变量维数过多，固定步长下，迭代后期步长过大，存在无法收敛的问题。

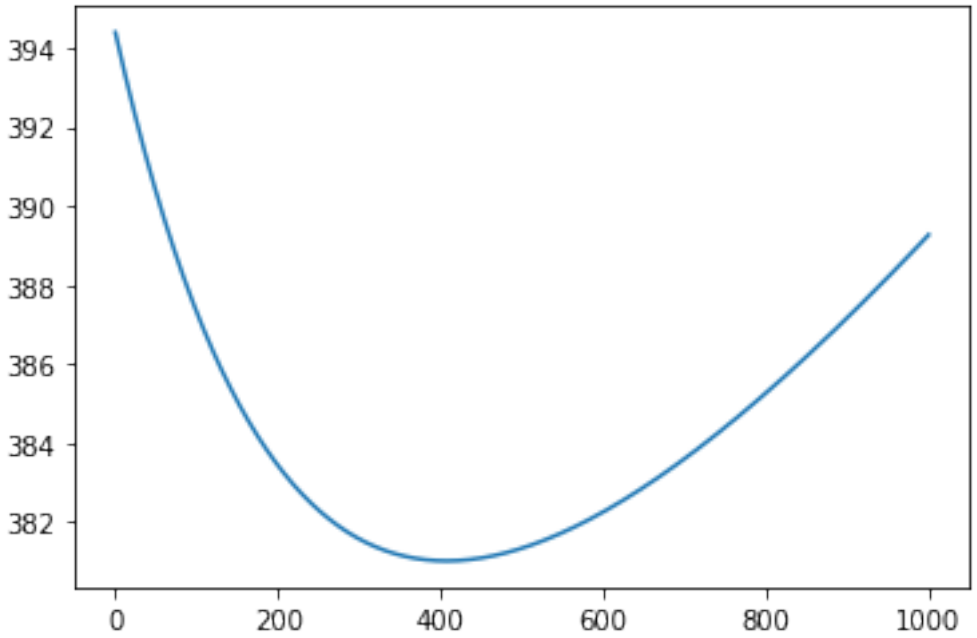
[42]:
```
x1,y = read_sig(data)  # 读取数据
x = normalize(x1)      # 数据归一化
result,loss = logit_gradient_descent(x,y,np.zeros(x.shape[1]+1),1000)
print(result[1000])    # 1000 次迭代后的结果
```

```
[-0.48168394  0.50799971  0.43976882  0.53228799 -0.0179412   0.21863828
  0.15249567 -0.00506235  0.31989157  0.26665304  0.74487394]
```

[43]:
```
import matplotlib.pyplot as plt
x_ray=range(0,1001)
plt.plot(x_ray,result)
plt.show()
# 迭代过程中 beta 值的变化，横轴为迭代次数，纵轴为 beta[i] 的值
```



[44]:
```
plt.plot(x_ray,loss)
plt.show()
# 迭代过程中损失函数值的变化，横轴为迭代次数，纵轴为损失函数值
```



2.2 改进的梯度下降法：Backtracking Line Search

我们试图从改进步长选取的角度解决这一问题。注意到有效步长会随着梯度迭代而减小，因此我们设置一个学习率 lr，在选定梯度下降方向的情况下，如果在该方向上前进原有的步长使损失函数值不减反增，说明我们的步长需要进一步减小，我们规定每次减小的比例为 lr。因此我们在迭代过程中增设一个循环，用于调整步长。

[12]:
```
def logit_line_search(x,y,init,epoch,lr):
    b = np.ones(x.shape[0])
    x1 = np.c_[b,x]                # 为自变量加上系数列
    beta = np.concatenate((init.reshape(1,init.shape[0]),np.zeros((epoch,x1.
↪shape[1])))) # 准备用于记录迭代结果的数组
    loss = np.zeros(epoch)
    step = 0.01                    # 起始步长
    for iter in range(epoch):
        dx = logit_der1(x1,y,beta[iter])
        loss[iter] = logit_loss(x1,y,beta[iter])
        while (logit_loss(x1,y,beta[iter] - step*dx) > loss[iter] - 0.25*step␣
↪*np.vdot(dx,dx)):
            step = lr * step           # 在一定条件下调整步长
        beta[iter+1] = beta[iter] - step * logit_der1(x1,y,beta[iter])
    return beta,loss
```

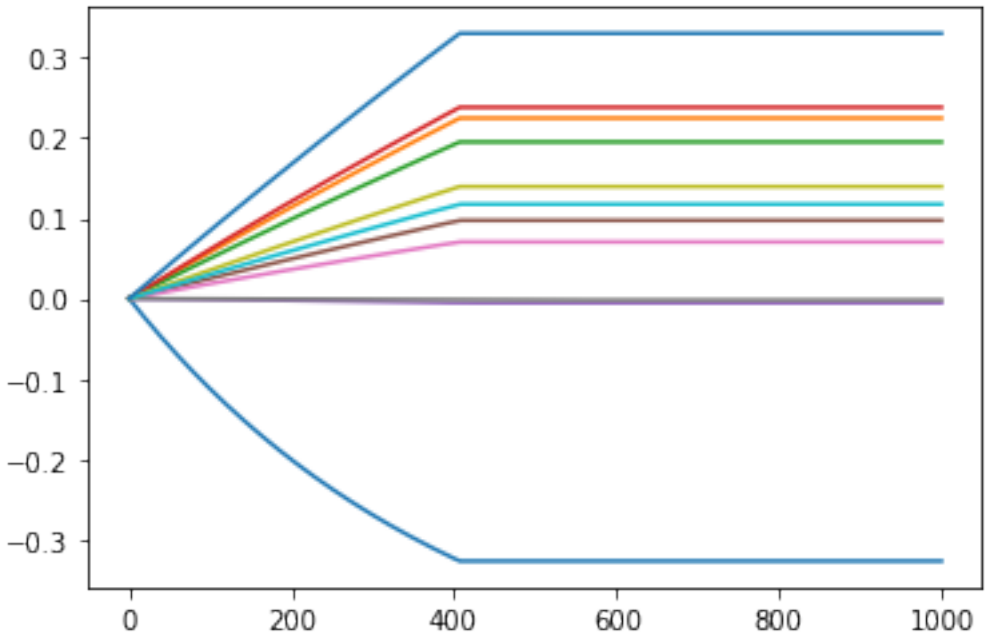
我们仍设置初始步长为 0.01，迭代过程中步长以 0.75 的比例减小。同样迭代 1000 次。

[13]:
```
x1,y = read_sig(data)
x = normalize(x1)
ls_result,ls_loss = logit_line_search(x,y,np.zeros(x.shape[1]+1),1000,0.75)
print (ls_result[1000])
```

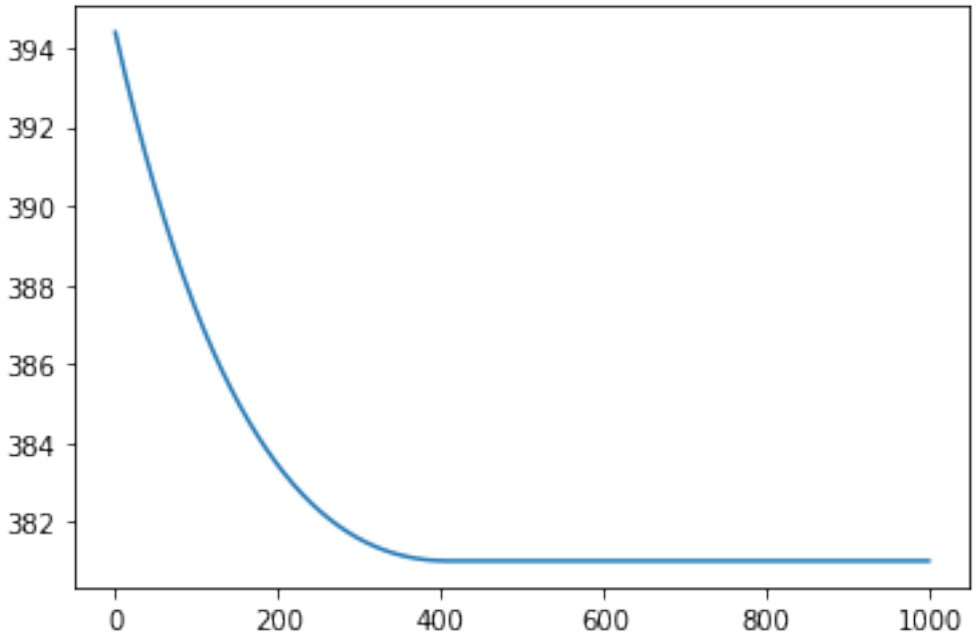
```
[-0.32692597  0.22482821  0.1951127   0.23828183 -0.00534315  0.09740933
  0.0705666  -0.00171805  0.13957216  0.11758014  0.3303888 ]
```

[14]:
```
x_ray=range(0,1001)
plt.plot(x_ray,ls_result)
plt.show()
```

```
# 迭代过程中 beta 值的变化，横轴为迭代次数，纵轴为 beta[i] 的值
```



[15]:
```
x_ray=range(0,1000)
plt.plot(x_ray,ls_loss)
plt.show()
# 迭代过程中损失函数值的变化，横轴为迭代次数，纵轴为损失函数值
```



2.3 牛顿法

梯度下降法存在迭代次数过多, 运算时间过长的问题。由于优化目标是一个 convex problem。我们可以使用牛顿法求解。首先我们计算损失函数的二阶导数。

$$\begin{aligned}\nabla^2 L(\beta) &= \sum_{i=1}^{m} \frac{e^{X_i^T\beta}}{\left(1+e^{X_i^T\beta}\right)^2} X_i^T X_i \\ &= \sum_{i=1}^{m} \frac{1}{2+e^{X_i^T\beta}+e^{-X_i^T\beta}} X_i^T X_i\end{aligned}$$

[16]:
```
# 损失函数二阶导数
def logit_der2(x,y,b):
    dim = x.shape[0]
    return (sum([  np.matmul(x[i].reshape(x[i].shape[0],1),x[i].reshape(1,x[i].
 ↪shape[0]))
                  /(np.exp(- np.vdot(x[i],b)) + np.exp(np.vdot(x[i],b)) + 2)␣
 ↪for i in range(dim)])/dim)
```

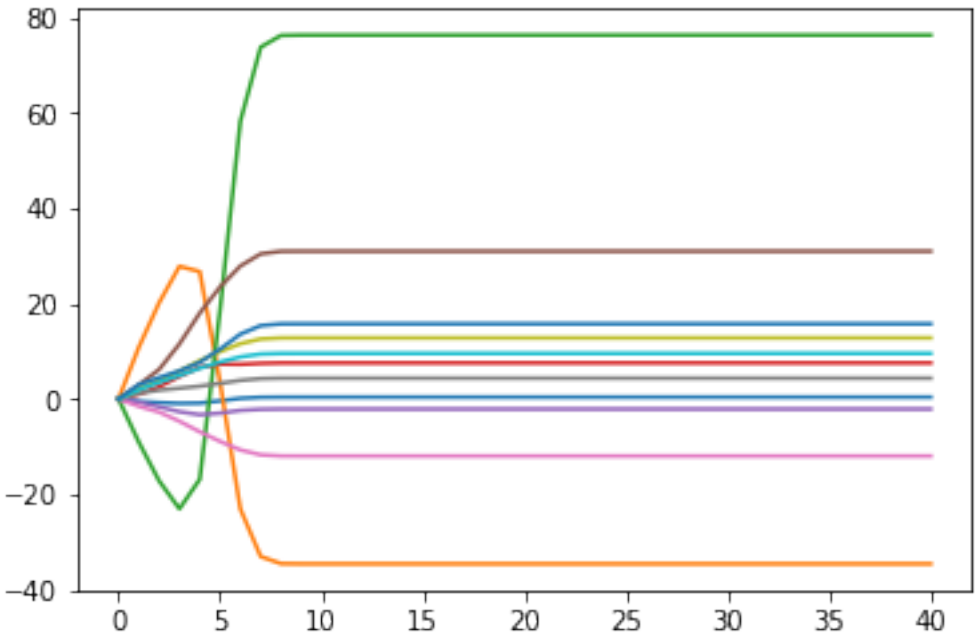
```
def logit_newton(x,y,init,epoch):
    b = np.ones(x.shape[0])
    x1 = np.c_[b,x]                # 为自变量加上系数列
    beta = np.concatenate((init.reshape(1,init.shape[0]),np.zeros((epoch,x1.
 ↪shape[1]))))
    loss = np.zeros(epoch+1)
    loss[0] = logit_loss(x1,y,init)
    for iter in range(epoch):
        second_der =logit_der2(x1,y,beta[iter])
        der_inv = np.linalg.inv(np.array(second_der,dtype='float'))
        beta[iter+1] = beta[iter] - np.dot(der_inv,logit_der1(x1,y,beta[iter]))
    return beta,loss
```

[ ]:
```
对全部数据进行牛顿法逻辑回归计算，得到结果如下，发现收敛速度很快。
```

[17]:
```
x1,y = read_sig(data)
x = normalize(x1)
newton_result, newton_loss = logit_newton(x,y,np.
 ↪array([0,0,0,0,0,0,0,0,0,0,0]),40)
print (newton_result[10])
```

```
[  0.38755791 -34.6008646   76.36750292   7.56697541  -2.10969107
  31.01540005 -11.97725587   4.36794273  12.81851241   9.56045522
  15.77466697]
```

[18]:
```
x_ray=range(0,41)
plt.plot(x_ray,newton_result)
plt.show()
```



3 模型测试

我们使用留出法对模型进行简单测试，我们调用 sklearn 的方法将数据集随机分为 75% 的训练集和 25% 的测试集，调用牛顿法逻辑回归模型，训练结果如下所示。

[48]:
```
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(x, y, test_size=0.25,␣
 ↪random_state=42)
train_result,train_loss = logit_newton(X_train,y_train,np.
 ↪array([0,0,0,0,0,0,0,0,0,0,0]),40)
print(train_result[-1:])
```

```
[[  0.28598197 -31.55924357  70.88713246   8.54281063  -2.25819591
   29.13817011 -11.75582428   3.75706483  13.57355246   8.8459486
   14.78123584]]
```

[63]:
```
def test_model(x1,b):
    one = np.ones(x1.shape[0])
    x = np.c_[one,x1]
```

```
    x=np.array(x,dtype=np.float64)
    b=np.array(b,dtype=np.float64)
    sigmoid = 1/(1+np.exp(- np.dot(x,b.T)))
    return (np.sign(sigmoid-0.5)+1)/2

np.set_printoptions(threshold=1)
test_results = np.c_[test_model(X_test,train_result[-1:]),y_test]
print (test_results)# 测试集结果与真实值比较
```

```
[[0. 0.]
 [1. 1.]
 [1. 1.]
 …
 [0. 0.]
 [1. 1.]
 [0. 0.]]
```

我们对测试集和真实结果进行比较, 发现逻辑回归对该数据集预测效果较好。

[70]:
```
def test_analysis(result):
    TP, FP, TN, FN = 0, 0, 0, 0
    for i in range(result.shape[0]):
        if (result[i] == [0,0]).all():
            TN += 1
        if (result[i] == [0,1]).all():
            FN += 1
        if (result[i] == [1,1]).all():
            TP += 1
        if (result[i] == [1,0]).all():
            FP += 1
    print (TN,FN,TP,FP)
    print ("查准率 P: ", TP/(TP+FP))
    print ("查全率 R: ", TP/(TP+FN))
    print ("F1: ", 2*TP/(2*TP+FN+FP))
    return

test_analysis(test_results)
```

```
88 2 52 1
查准率 P:  0.9811320754716981
查全率 R:  0.9629629629629629
F1:  0.9719626168224299
```

4 问题和讨论

由于时间精力和计算资源有限，本次实验中遇到了以下问题有待解决。

1. 如何看待 sklearn 标准库中的逻辑回归模型、梯度下降模型、牛顿法模型得到的参数值有较大出入？

2. 实验中遇到了大量浮点数运算溢出的问题，经过对原数据不同程度 normalize 的尝试才得以避免，这样的操作合理吗？

3. 在梯度下降的模型中，如何选择合适的步长？经过尝试，发现使用固定步长存在不收敛的可能性，而使用 backtracking line search 等方法虽然理论上有可以收敛，但可能导致步长锐减太快达到浮点数运算的极限，是否存在一些调整参数、选择方法上的原则？