哈尔滨工业大学计算机科学与技术学院

实验报告

课程名称： 机器学习
课程类型： 选修
实验题目： 逻辑回归

学号：1170301007
姓名：沈子鸣

一、实验目的

理解逻辑回归模型，掌握逻辑回归模型的参数估计算法。

二、实验要求及实验环境

实验要求：

实现两种损失函数的参数估计（1，无惩罚项；2.加入对参数的惩罚），可以采用梯度下降、共轭梯度或者牛顿法等。

实验环境：

Windows 10 专业教育版；python 3.7.4；jupyter notebook 6.0.1

三、设计思想（本程序中的用到的主要算法及数据结构）

1. 算法原理

   我们分类器做分类问题的实质，就是预测一个已知样本的位置标签，即 $P(Y=1|X=<X_1,\ldots,X_n)$. 按照朴素贝叶斯的方法，可以用贝叶斯概率公式，将其转化为类条件概率（似然）和类概率的乘积。而我们这次实验，则是直接求该概率，推导公式如下：

   假设分类问题是一个0/1二分类问题：

$$P(Y=1|X)=\frac{P(Y=1)P(X|Y=1)}{P(X)}$$

$$=\frac{P(Y=1)P(X|Y=1)}{P(Y=1)P(X|Y=1)+P(Y=0)P(X|Y=0)}$$

$$\text{上下同除}=\frac{1}{1+\frac{P(Y=0)P(X|Y=0)}{P(Y=1)P(X|Y=1)}}$$

$$=\frac{1}{1+exp(ln\frac{P(Y=0)P(X|Y=0)}{P(Y=1)P(X|Y=1)})}$$

$$\text{令}\pi=P(Y=1),\ 1-\pi=P(Y=0)$$

$$=\frac{1}{1+exp(ln(\frac{1-\pi}{\pi})+\sum_i ln\frac{P(X_i|Y=0)}{P(X_i|Y=1)})}$$

$$\text{假设类条件分布服从正态分布且方差不依赖于}k。\ P(x|y_k)=\frac{1}{\sigma_i\sqrt{2\pi}}e^{\frac{-(x-\mu_{ik})^2}{2\sigma_i^2}}$$

$$=\frac{1}{1+exp(ln(\frac{1-\pi}{\pi})+\sum_i ln(P(X_i|Y=0)-P(X_i|Y=1)))}$$

$$=\frac{1}{1+exp(ln(\frac{1-\pi}{\pi})+\sum_i(-ln\sigma_i\sqrt{2\pi}-\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}-(-ln\sigma_i\sqrt{2\pi}-\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2})))}$$

$$=\frac{1}{1+exp(ln(\frac{1-\pi}{\pi})+\sum_i(\frac{(x_i-\mu_{i1})^2}{2\sigma_i^2}-\frac{(x_i-\mu_{i0})^2}{2\sigma_i^2}))}$$

$$=\frac{1}{1+exp(ln(\frac{1-\pi}{\pi})+\sum_i(\frac{\mu_{i0}-\mu_{i1}}{\sigma_i^2}x_i+\frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}))}$$

$$\text{令}w_0=\sum_i\frac{\mu_{i1}^2-\mu_{i0}^2}{2\sigma_i^2}+ln\frac{1-\pi}{\pi},\ w_i=\frac{\mu_{i0}-\mu_{i1}}{\sigma_i^2}$$

$$\text{最后}$$

$$P(Y=1|X)=\frac{1}{1+exp(w_0+\sigma_{i=1}^n w_iX_i)}$$

形如$a=\frac{1}{1+exp(-b)}$的函数被叫做sigmoid函数，它具有将实值映射到0到1之间的某个值的特性，而且sigmoid(0)=0.5，且从x=0向右向左开始，y值分别以很快的速度向1和0逼近。可以看作这个函数将我们的线性模型预测的连续值映射到0到1的概率上，从而得到0/1标签离散值。

Logistics regression的参数，就是w，也就是$w_i, 0 \leq i \leq n$，n为样本维度。要计算损失函数$l(w)$，用极大似然估计（MLE）是难的问题，因为需要计算$P(<X,Y>|W)$，我们可以将其转化为计算极大条件似然估计（MCLE），只需要计算$P(X|Y,W)$。而
$P(Y=0|X,W)=\frac{1}{1+exp(w_0+\sum_i w_i X_i)}$，所以我们有

$$
\begin{aligned}
l(W) &= \sum_l Y^l lnP(Y^l=1|X^l,W) + (1-Y^l) lnP(Y^l=0|X^l,W) \\
&= \sum_l Y^l ln \frac{P(Y^l=1|X,W)}{P(Y^l=0|X^l,W)} + lnP(Y^l=0|X^l,W) \\
&= \sum_l Y^l (w_0 + \sum_i^n w_i X_i^l) - ln(1+exp(w_0+\sum_i^n w_i X_i^l))
\end{aligned}
$$

用梯度下降法求得$W=argmax_w l(w)$，注意要用梯度下降的话，一般要把这里的$l(w)$（MCLE）转化为相反数，$-l(w)$作为损失函数，求其最小值。

$$
\begin{aligned}
\frac{\partial l(w)}{\partial w_i} &= \sum_l X_i^l (Y^l - \frac{1}{1+exp(-(w_0+\sum_i^n w_i X_i^l))}) \\
w_i &\leftarrow w_i + \eta \sum_l X_i^l (Y^l - \frac{1}{1+exp(-(w_0+\sum_i^n w_i X_i^l))})
\end{aligned}
$$

在具体实现中，还要涉及为了控制过拟合问题而加入正则项。考虑到我们的分类问题中，决策面是线性的，那么就存在到底取哪一个线性决策面的问题。实际上肯定是有无数个线性决策面可以有效地区分开我们的两类数据，然而我们总是希望给分类留出“余地”，所以我们选取的决策面，往往是使“最短距离”最大化的那个。这个最短距离，就是两类中距离线性决策面最近的距离之和。记线性决策面方程为

$$w^T x + b = 0$$

设常数c，有

$$(w^T x_i + b) y_i > c$$

其中对于所有的$x_i \in class(1)$，$y_i = 1$，对于所有的$x_i \in class(2)$，$y_i = -1$。

由于$w^T x$是点到线性决策面的法向量的投影，那么记x_i^*和x_j^*分别是类别内距离决策面的最近点，即m为x_i^*与x_j^*到单位长的$\vec{w}$的投影值的差值，则有

$$m = \frac{w^T}{||w||}(x_i^* - x_j^*) = \frac{2c}{||w||}$$

从而我们的优化问题，变成了在约束条件

$$y_i(w^T x_i + b) \geq 1, \forall i$$

下，求得

$$
\begin{aligned}
&max_{w,b} \frac{1}{||w||_2} \\
&i.e. min_w w^T w
\end{aligned}
$$

从而我们加上正则项的梯度下降为

$$w_i \leftarrow w_i - \eta \lambda w_i + \eta \sum_l X_i^l (Y^l - \frac{1}{1+exp(-(w_0+\sum_i^n w_i X_i^l))})$$

2. 算法的实现

首先是生成数据，如果要生成类条件分布满足朴素贝叶斯假设的数据，那么就对每一个类别的每一个维度都用一个独立的高斯分布生成。如果要生成类条件分布不满足朴素贝叶斯假设的数据，那么就对每一个类别的两个维度用一个二维高斯分布生成。需要注意的是，由于高斯分布具有的特性，多维高斯分布不相关可以推出独立性，因此，可以用二维高斯分布生成数据，如果是满足朴素贝叶斯假设的，那么协方差矩阵的非对角线元素均为0，如果是不满足朴素贝叶斯假设的，那么协方差矩阵的非对角线元素不为0（协方差矩阵应该是对称阵）。

计算极大条件似然估计如下

```
'''
    极大条件似然
'''
def likelihood(train_x, train_y, weight):
    total = np.size(train_x, axis=0)
    predict = np.zeros((total, 1))
    for i in range(total):
        predict[i] = np.dot(weight, train_x[i].T)
    t = 0
    for i in range(total):
        t += np.log(1 + np.exp(predict[i]))
    return np.dot(train_y, predict) - t
```

梯度下降算法如下：

```
def descent_gradient(train_x, train_y, epoch, eta, eps, dimension, lam):
    total = np.size(train_x, axis=0)
    weight = np.ones((1, dimension + 1))
    epoch_list = np.zeros(epoch)
    loss_list = np.zeros(epoch)
    for i in range(epoch):
        old_loss = - 1 / total * likelihood(train_x, train_y, weight)
        t = np.zeros((total, 1))
        for j in range(total):
            t[j] = np.dot(weight, train_x[j].T)
        gradient = - 1 / total * np.dot(train_y - sigmoid(t.T), train_x)
        weight = weight - eta * lam * weight - eta * gradient # 梯度下降
        new_loss = - 1 / total * likelihood(train_x, train_y, weight)
        epoch_list[i] = i
        loss_list[i] = new_loss
        if i % 100 == 0:
            print(i, ', loss=', new_loss, ', weight=', weight, 'gradient=', gradient)
        if abs(new_loss - old_loss) < eps:
            epoch_list = epoch_list[:i+1]
            loss_list = loss_list[:i+1]
            break
    return weight, epoch_list, loss_list
```

在做UCI上的数据时，选取了皮肤Skin_NonSkin.txt数据。由于该数据量太大，这里只选取了其中一部分，读取数据时，用numpy切片提取数据信息，用50作为步长，提取部分数据用做实验。还要对样本点进行空间平移，否则在计算MCLE时可能会溢出，因为计算MCLE时，要用参数与样本做矩阵乘法，而且还要作为e的指数计算，可能会溢出。

```python
def skin_gen_data():
    load_data = np.loadtxt('./Skin_NonSkin.txt', dtype=np.int32)
    np.random.shuffle(load_data) # 打乱原数据，以便分成训练集和测试集
    test_data_rate = 0.2 # 测试集比例
    load_data_size = np.size(load_data, axis=0)
    train_data = load_data[:int(test_data_rate * load_data_size), :] # 训练集数据
    test_data = load_data[int(test_data_rate * load_data_size):, :] # 测试集数据
    dim = np.size(load_data, axis=1) - 1 # 训练集样本维度

    step = 50 # 本数据集太大，采用步长为50的方式，选取打乱的数据
    train_x = train_data[:,0:dim]
    train_x = train_x[::step]
    train_x = train_x - 100 # 对样本点进行坐标平移
    train_y = train_data[:,dim:dim + 1] - 1
    train_y = train_y[::step]
    train_size = np.size(train_x, axis=0)
    train_y = train_y.reshape(train_size) # 矩阵转化为行向量

    test_x = test_data[:,0:dim]
    test_x = test_x[::step] - 100  # 对样本点进行坐标平移
    test_y = test_data[:,dim:dim + 1] - 1
    test_y = test_y[::step]
    test_size = np.size(test_x, axis=0)
    test_y = test_y.reshape(test_size) # 矩阵转化为行向量
    return train_x, train_y, test_x, test_y
```

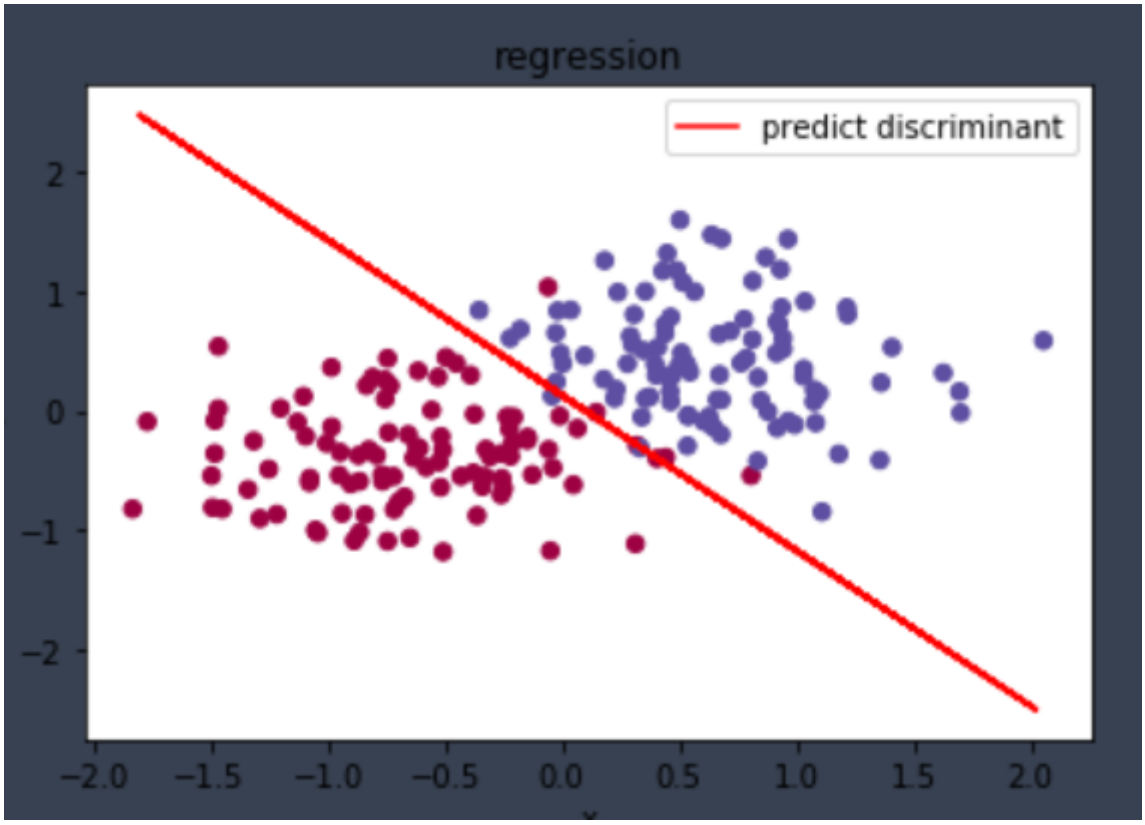
这里还找了UCI上的另一个数据集banknote，这些数据不多，是浮点数形式的，绝对值也不大，正常处理就好。

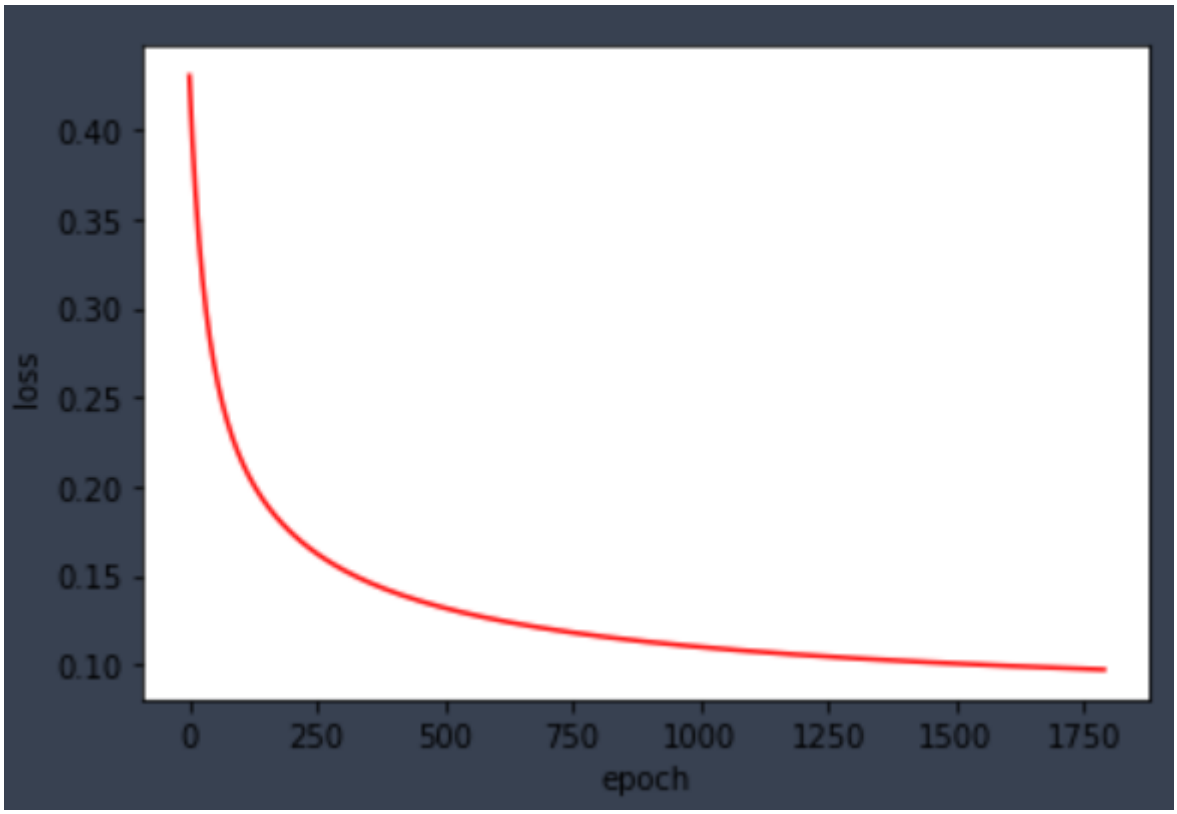
四、实验结果与分析

自己生成数据

类条件概率满足朴素贝叶斯假设，正则项λ=0，size=200

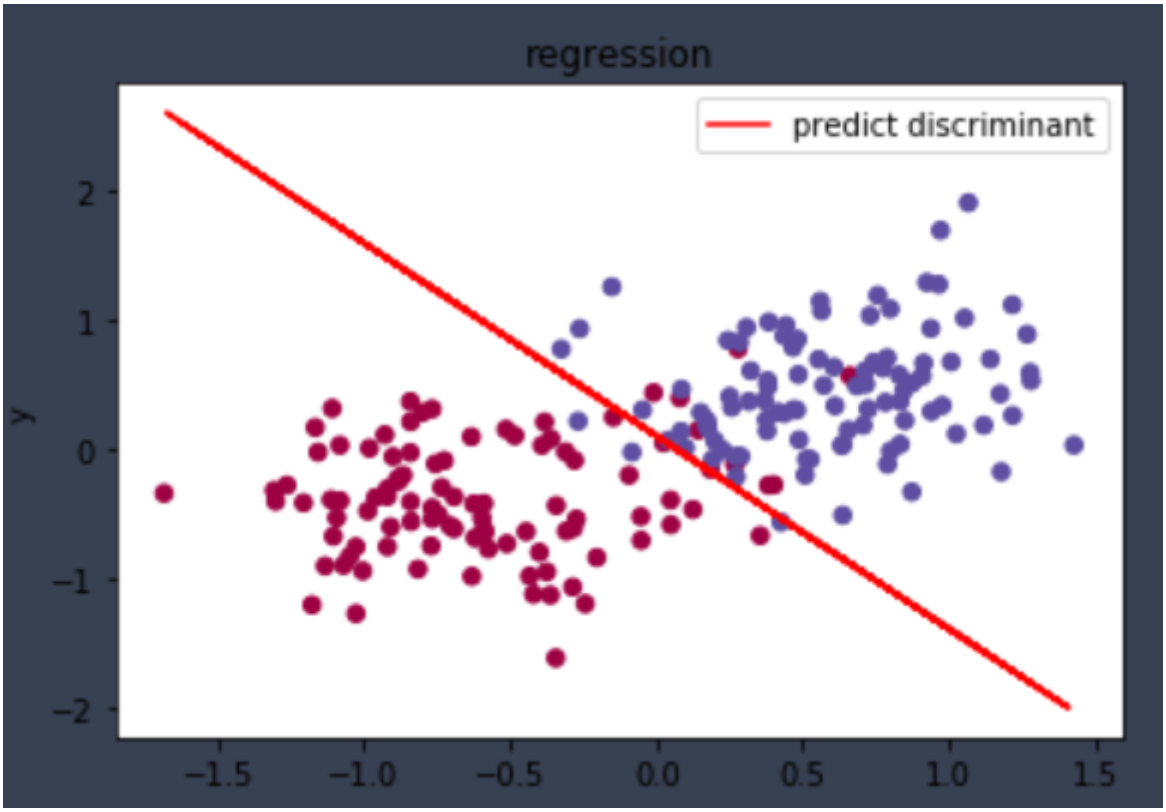
```
1200 , loss= [0.10544331] , weight= [[-0.40642963  4.42804169  3.30557911]] gradient= [[ 0.00191917 -0.01024958 -0.00
92028 ]]
1300 , loss= [0.10363255] , weight= [[-0.42508422  4.52680404  3.39505036]] gradient= [[ 0.00181489 -0.00952784 -0.00
870632]]
1400 , loss= [0.10203516] , weight= [[-0.44275527  4.61885002  3.47983827]] gradient= [[ 0.00172197 -0.00890195 -0.00
826386]]
1500 , loss= [0.10061446] , weight= [[-0.45954596  4.70504396  3.56043501]] gradient= [[ 0.00163843 -0.00835407 -0.00
78663 ]]
1600 , loss= [0.09934191] , weight= [[-0.4755423   4.786094    3.63725218]] gradient= [[ 0.0015628  -0.00787053 -0.00
750653]]
1700 , loss= [0.09819494] , weight= [[-0.49081721  4.86258742  3.71063836]] gradient= [[ 0.00149391 -0.00744064 -0.00
717895]]
The predict discriminant function: y =
-1.306 x + 0.1336
```

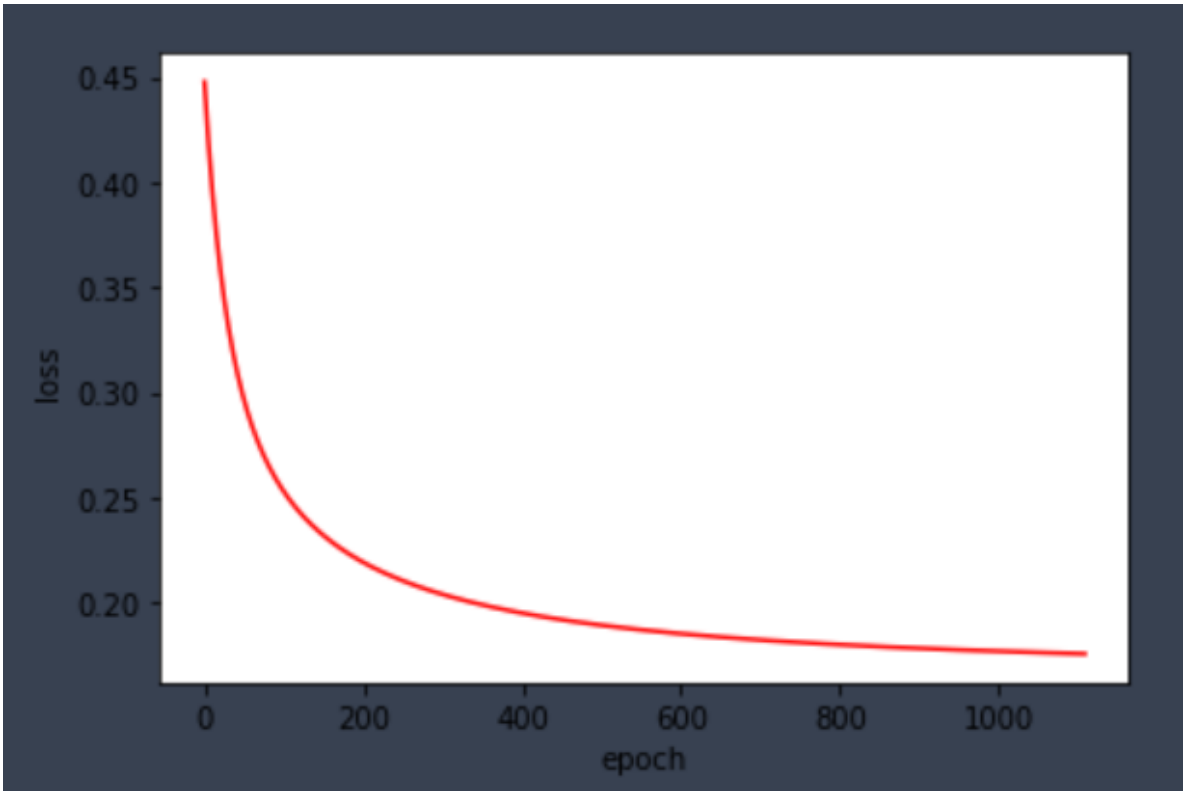




类条件概率不满足朴素贝叶斯假设，正则项λ=0，size=200

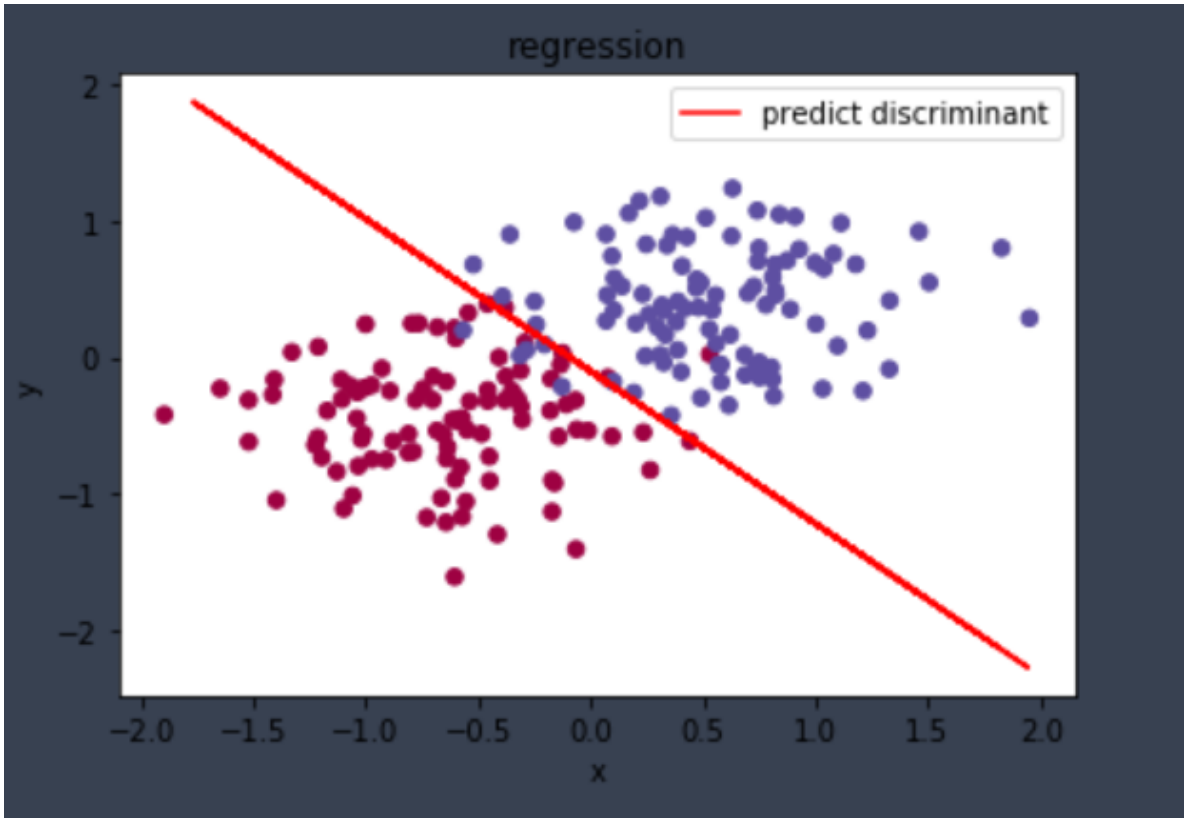
```
700 , loss= [0.18231064] , weight= [[-0.20031526  3.50266831  2.40682382]] gradient= [[ 0.00200353 -0.01389552 -0.007
65763]]
800 , loss= [0.18007005] , weight= [[-0.21927072  3.63264514  2.47848539]] gradient= [[ 0.00179774 -0.01218282 -0.006
72042]]
900 , loss= [0.17832509] , weight= [[-0.23637914  3.74728646  2.54176256]] gradient= [[ 0.00163073 -0.01080659 -0.005
96855]]
1000 , loss= [0.17693717] , weight= [[-0.25195554  3.84946819  2.59823543]] gradient= [[ 0.00148958 -0.0096763  -0.00
535146]]
1100 , loss= [0.17581423] , weight= [[-0.26622086  3.9413244   2.6490713 ]] gradient= [[ 0.00136748 -0.00873132 -0.00
483555]]
The predict discriminant function: y =
-1.488 x + 0.1008
```

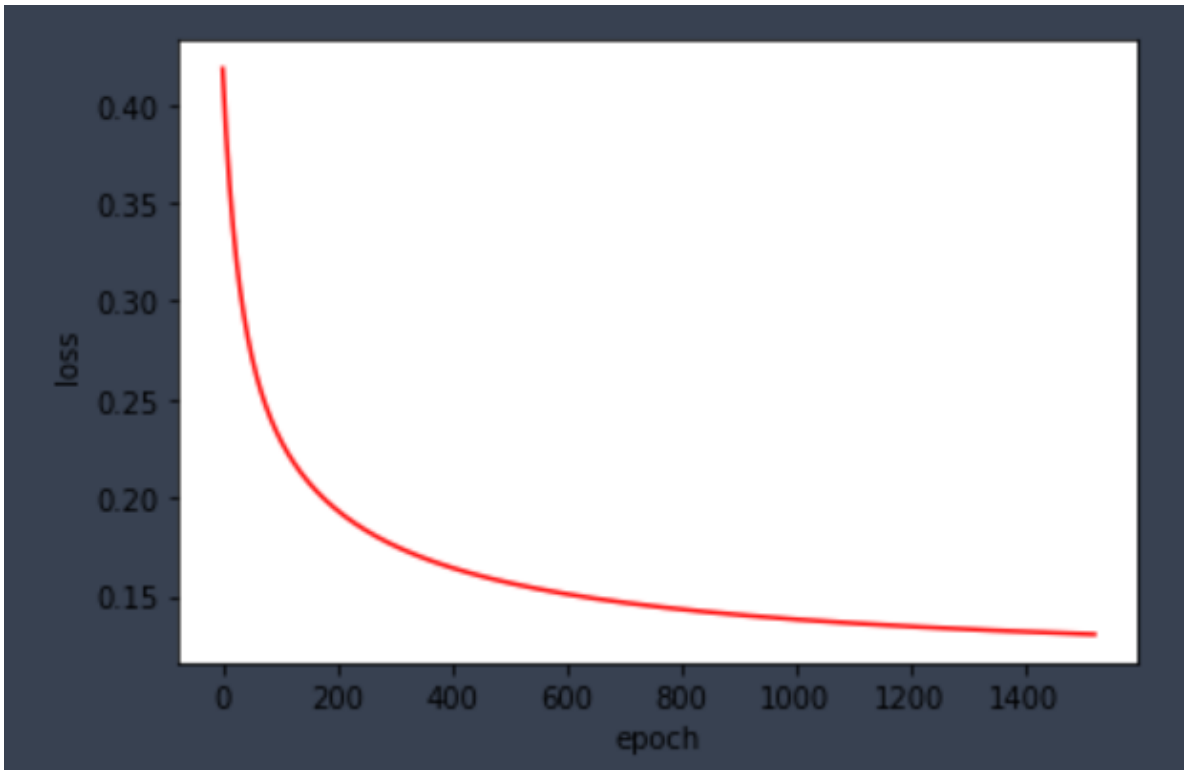




类条件分布满足朴素贝叶斯假设，正则项λ=0.001，size=200

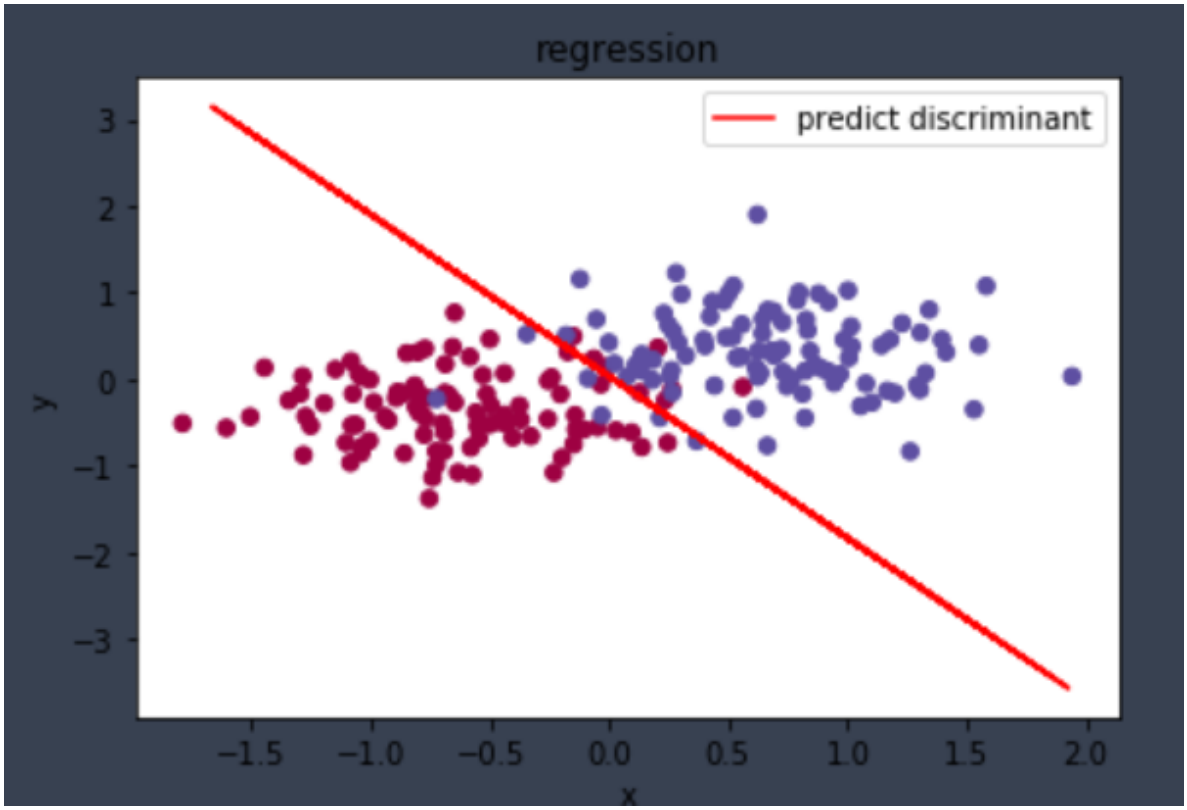
```
1100 , loss= [0.13619749] , weight= [[0.33399084 3.78129915 3.31598165]] gradient= [[-0.00121296 -0.0110548  -0.01170
271]]
1200 , loss= [0.13453544] , weight= [[0.34237578 3.85013267 3.39633794]] gradient= [[-0.00114283 -0.0103704  -0.01110
109]]
1300 , loss= [0.13310255] , weight= [[0.35002715 3.91202056 3.47029935]] gradient= [[-0.00108189 -0.00979113 -0.01057
462]]
1400 , loss= [0.13185564] , weight= [[0.35703556 3.96796588 3.5386054 ]] gradient= [[-0.00102858 -0.00929574 -0.01010
966]]
1500 , loss= [0.13076185] , weight= [[0.36347668 4.01877817 3.60187012]] gradient= [[-0.00098162 -0.00886822 -0.00969
585]]
The predict discriminant function: y =
-1.115 x - 0.1009
```

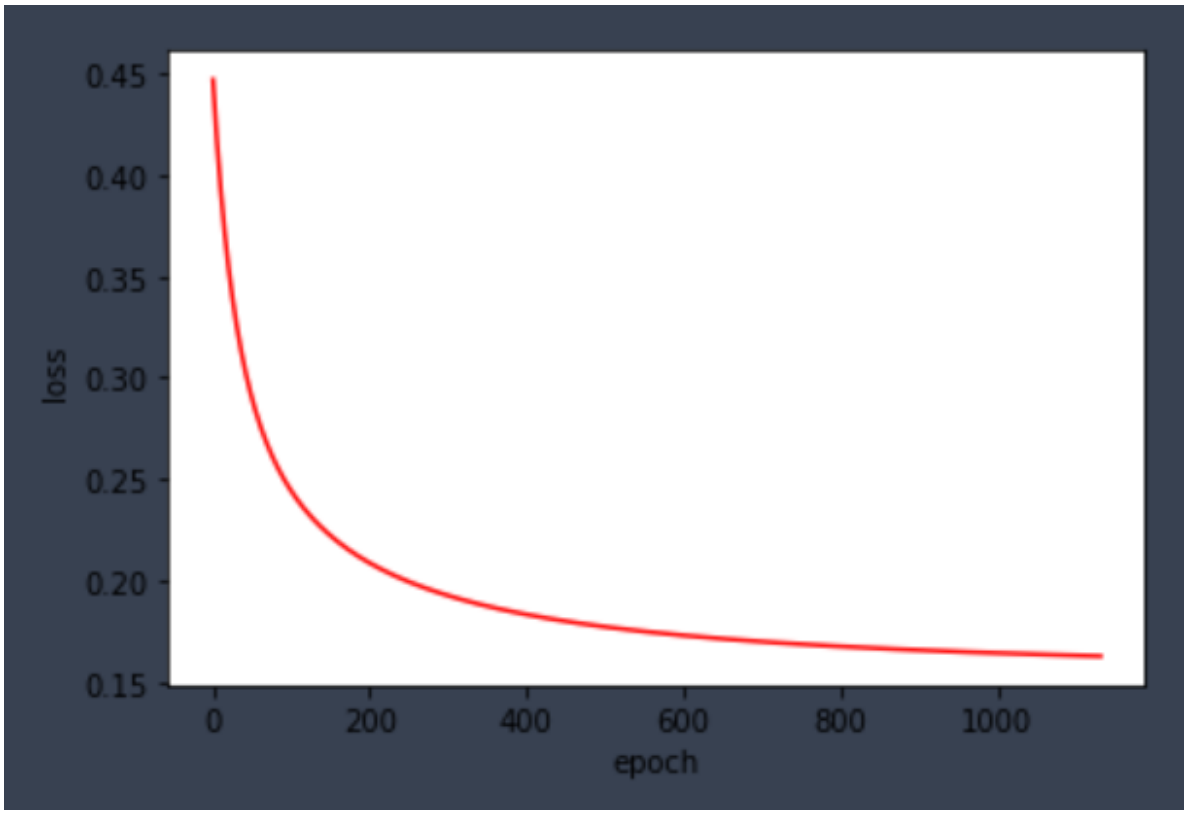




类条件概率不满足朴素贝叶斯假设，正则项λ=0.001，size=200

```
700 , loss= [0.17006162] , weight= [[-0.05947835  3.6859632   2.0390772 ]] gradient= [[ 0.00061048 -0.01712068 -0.007
16576]]
800 , loss= [0.16772179] , weight= [[-0.06447239  3.81024314  2.08665708]] gradient= [[ 0.00051966 -0.0153234  -0.006
50902]]
900 , loss= [0.1659001] , weight= [[-0.06867018  3.91735152  2.12792752]] gradient= [[ 0.00045712 -0.01389329 -0.0059
8394]]
1000 , loss= [0.16445088] , weight= [[-0.07229015  4.01057738  2.16406941]] gradient= [[ 0.00041035 -0.0127309  -0.00
555489]]
1100 , loss= [0.16327752] , weight= [[-0.07546132  4.09236754  2.19596239]] gradient= [[ 0.00037335 -0.01176982 -0.00
519815]]
The predict discriminant function: y =
-1.866 x + 0.03463
```

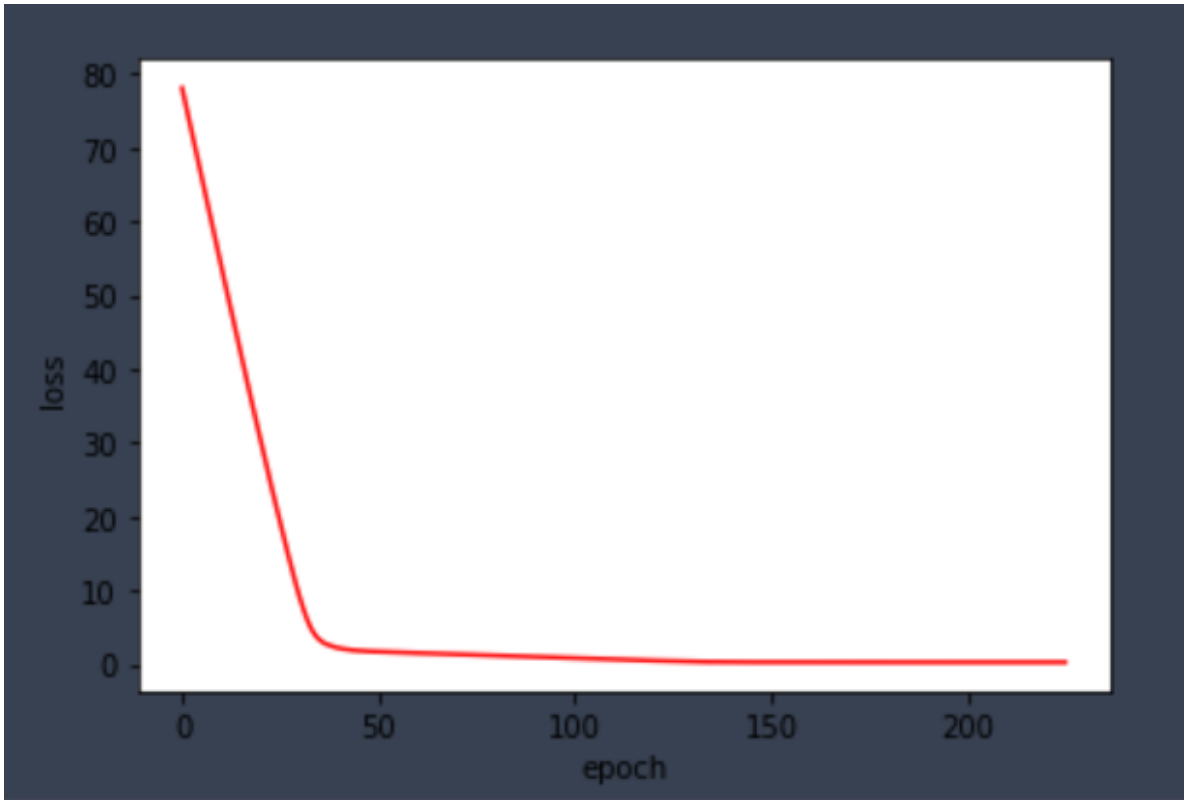


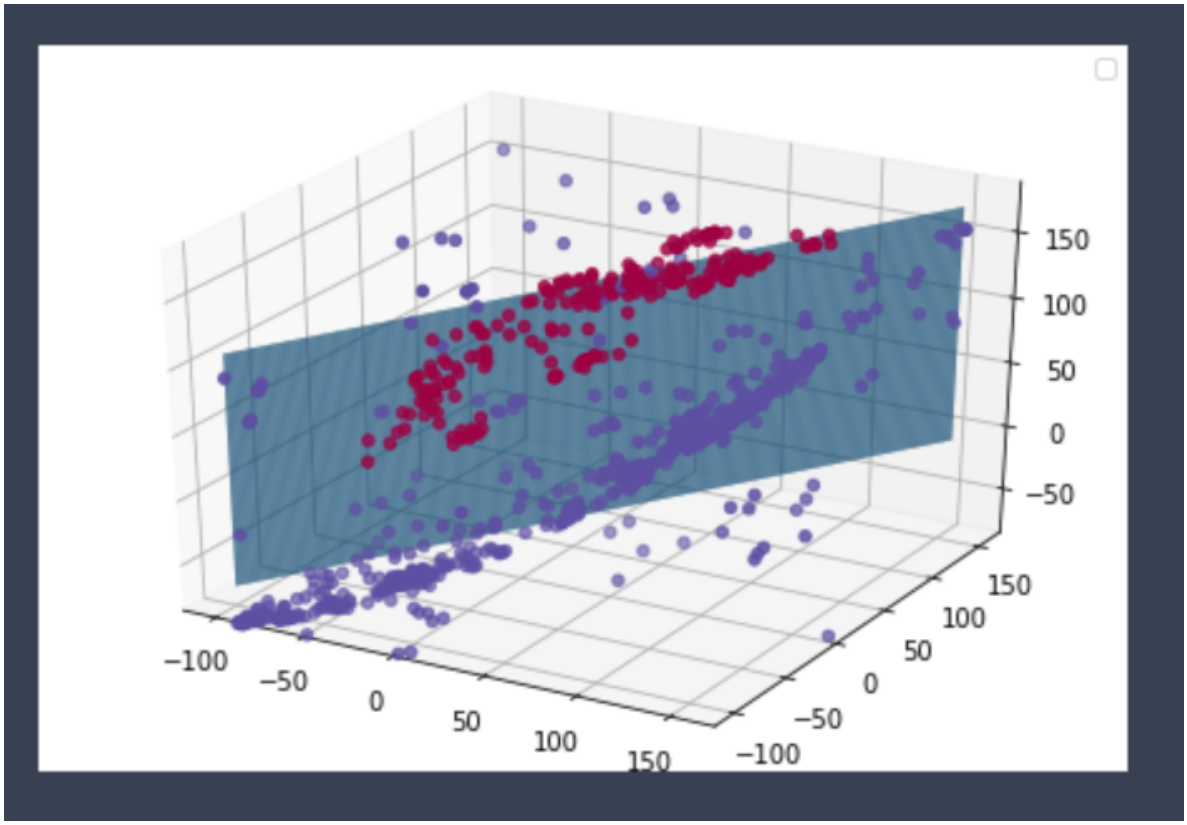


UCI皮肤颜色数据集

正则项λ=0

```
0 , loss= [78.0732635] , weight= [[1.00006045 0.98252067 0.97690828 0.96000533]] gradient= [[-0.06045196 17.47932615
23.09172081 39.99466543]]
100 , loss= [0.80464362] , weight= [[ 1.00863631  0.09626433  0.08062129 -0.16601919]] gradient= [[-0.07696413  2.612
00237  0.66825174 -3.13056851]]
200 , loss= [0.25784807] , weight= [[ 1.01549561  0.00858208  0.0303685  -0.04090911]] gradient= [[-0.06732764 -0.074
39327  0.11880689 -0.04801726]]
```



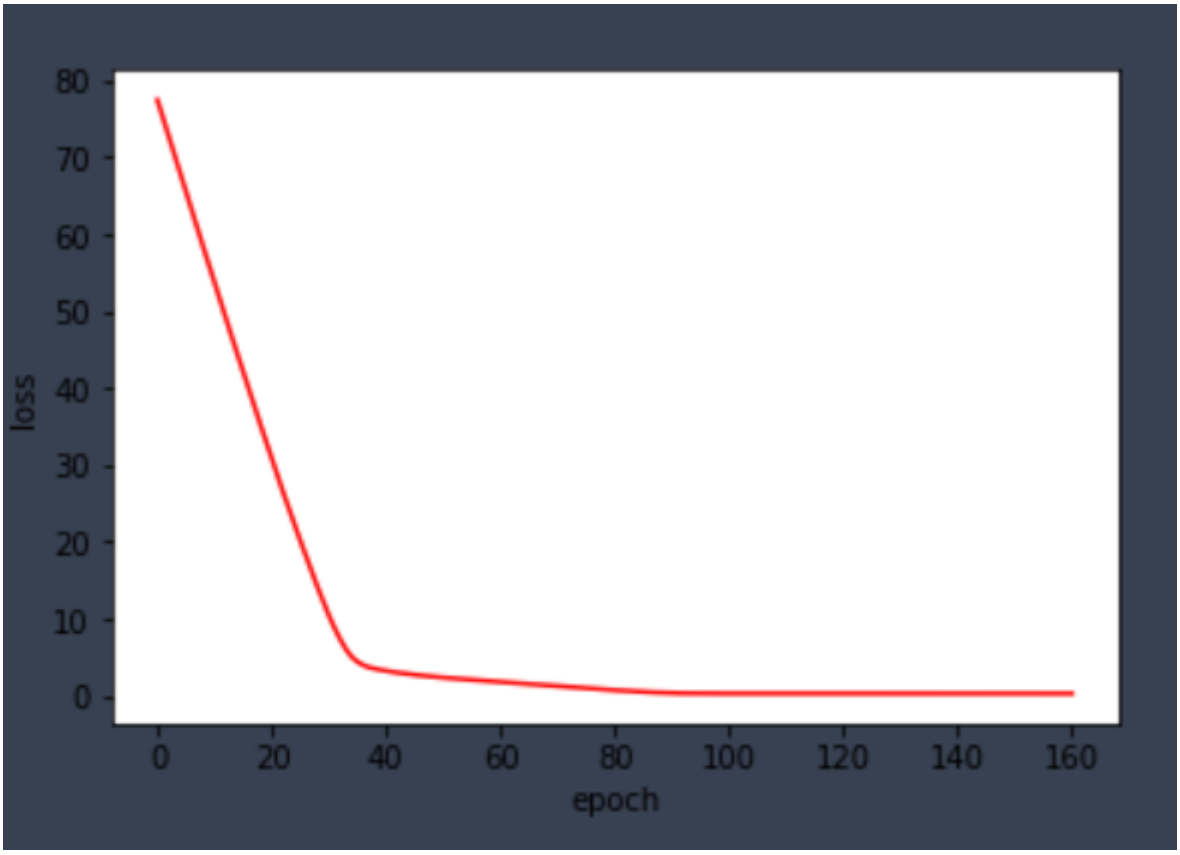


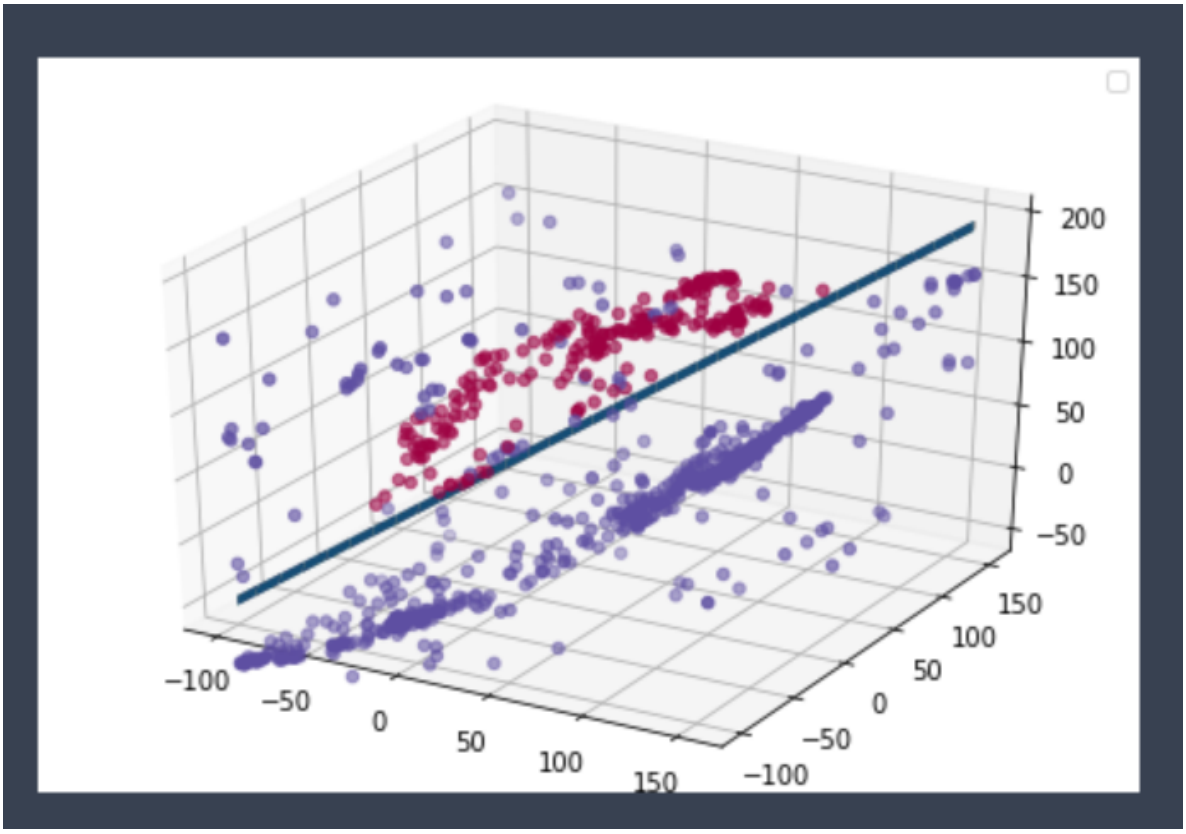
20%测试集准确率

```
0.9466972711043101
```

正则项λ=0.01

```
0 , loss= [77.38886887] , weight= [[1.00004208 0.98305931 0.97690587 0.96016225]] gradient= [[-0.05208352 16.93068734
23.08412807 39.82774965]]
100 , loss= [0.34776756] , weight= [[ 1.00989463  0.03055627 -0.01119228 -0.02029575]] gradient= [[-0.11392781  0.815
55458 -0.46662685 -0.24005138]]
```





20%测试集准确率

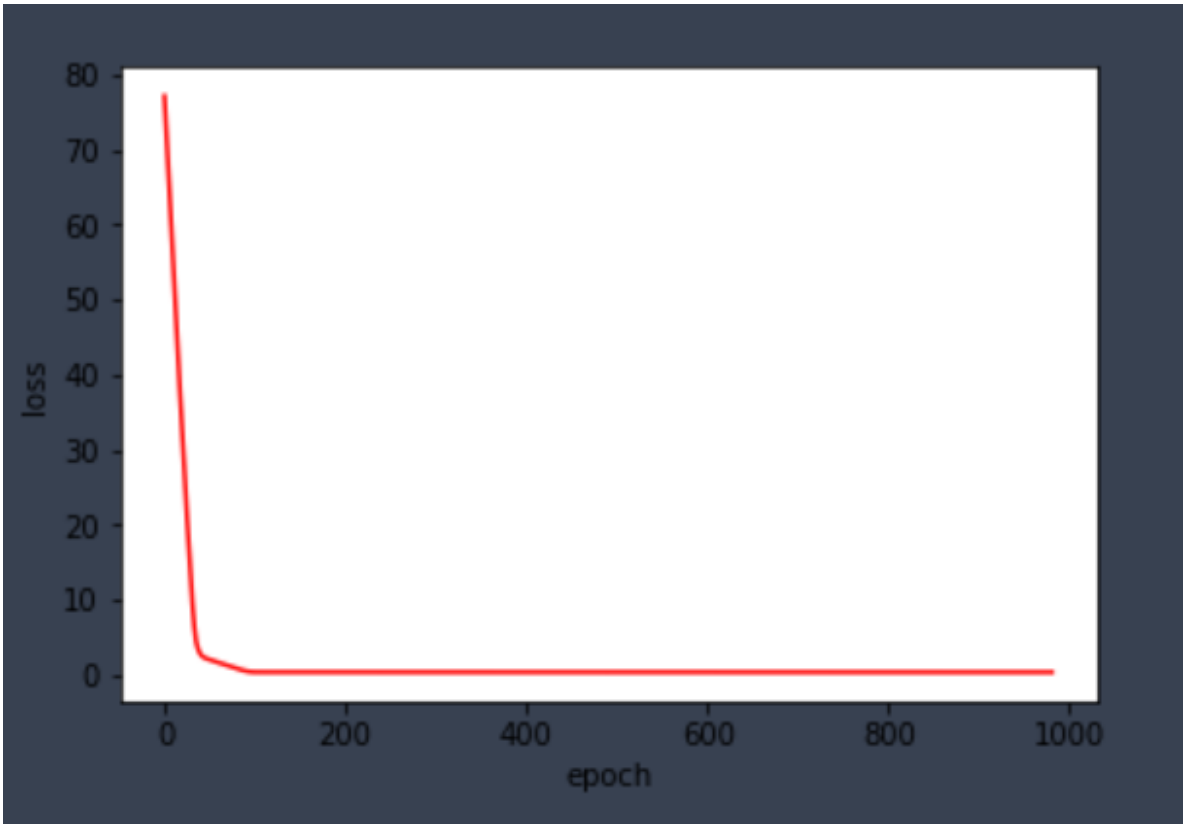
```
0.9380260137719969
```

UCI banknote数据集

这个数据集的样本维数是4维，所以就没办法简单地画图了。

正则项λ=0

```
600 , loss= [0.31363632] , weight= [[ 1.06536109  0.01766324  0.00327044 -0.02381963]] gradient= [[-1.03762813e-01 -
8.45175745e-04  1.48886882e-03 -8.78871294e-05]]
700 , loss= [0.31257003] , weight= [[ 1.07568603  0.01774741  0.00312246 -0.02381126]] gradient= [[-1.02748658e-01 -
8.38305308e-04  1.47103372e-03 -7.97475791e-05]]
800 , loss= [0.31152443] , weight= [[ 1.08591033  0.01783089  0.00297624 -0.02380368]] gradient= [[-1.01749918e-01 -
8.31444891e-04  1.45347565e-03 -7.18610933e-05]]
900 , loss= [0.310499] , weight= [[ 1.09603552  0.01791369  0.00283177 -0.02379689]] gradient= [[-1.00766311e-01 -8.2
4598199e-04  1.43619043e-03 -6.42196595e-05]]
```

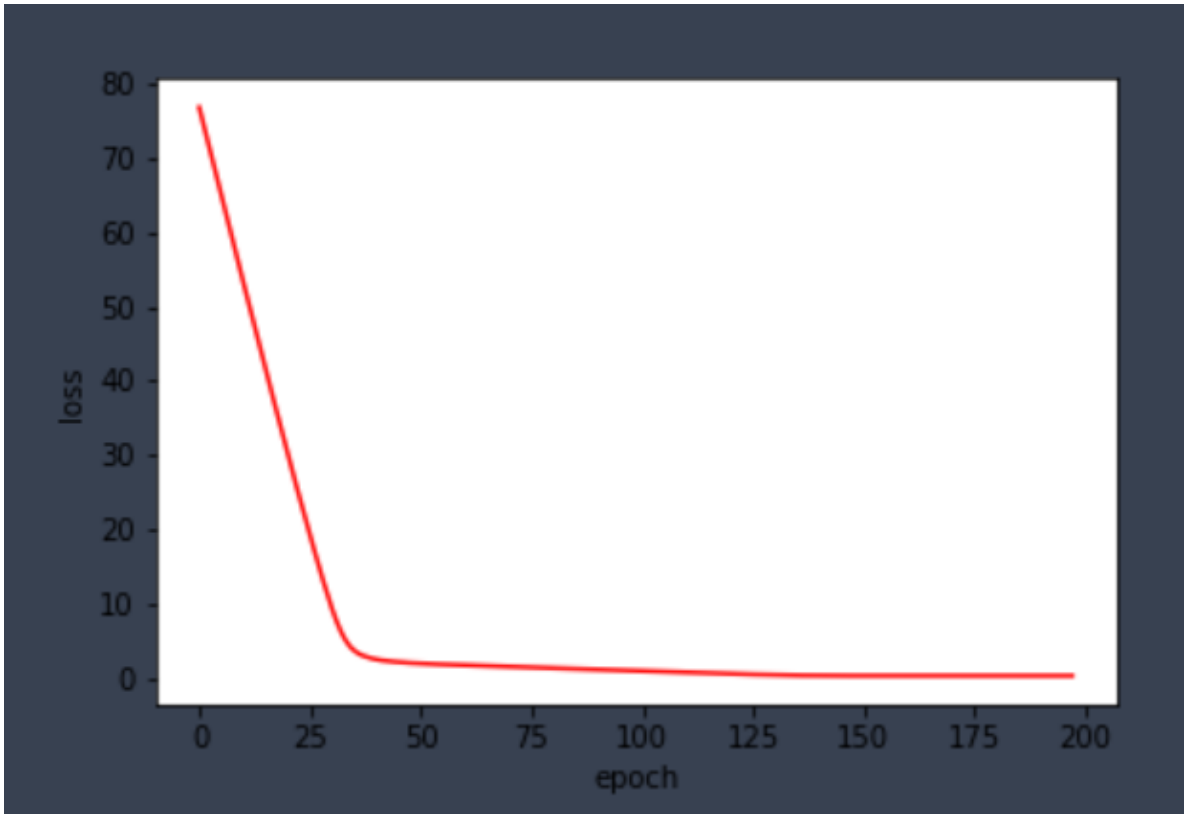


20%测试集准确率

```
0.9382810507523591
```

正则项λ=0.01

```
0 , loss= [76.83953028] , weight= [[1.00004489 0.9832704  0.97733132 0.9600917 ]] gradient= [[-0.05488676 16.71959601
22.65868104 39.89830429]]
100 , loss= [0.89789853] , weight= [[ 1.00753468  0.13236574  0.04148546 -0.15919396]] gradient= [[-0.07937962  2.440
36901  1.22793437 -3.45650807]]
```



20%测试集准确率

```
0.9410864575363428
```

五、结论

1. 反复实验，UCI的数据的20%测试集的准确率基本稳定在93%-94%。
2. 正则项在数据量较大时，对结果的影响不大，可能是因为此时过拟合现象较小。在数据量较小时，应该可以有效解决过拟合问题。
3. 从图中可以看出，类条件分布在满足朴素贝叶斯假设时的分类表现，要比不满足假设时略好。
4. logistics 回归可以很好地解决简单的线性分类问题，而且收敛速度较快。

六、参考文献

七、附录：源代码（带注释）

```
import numpy as np
import math
import matplotlib.pyplot as plt
import random
from sklearn.datasets import make_blobs
```

In [321]:

```
'''
    根据train_x和train_y画出二维散点图，并画出判别函数predict_discriminant
'''
def plt_show(train_x, train_y, predict_discriminant):
    plot = plt.scatter(train_x[:,0], train_x[:,1], c=train_y, s=30, marker='o',
cmap=plt.cm.Spectral)
    if predict_discriminant:
        real_x = min(train_x[:,0]) + (max(train_x[:,0]) - min(train_x[:,0])) *
np.random.random(50)
```

```
        real_y = predict_discriminant(real_x)
        plt.plot(real_x, real_y, 'r', label='predict discriminant')
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title('regression')
    plt.show()
```

In [322]:

```
'''
    根据epoch_list和loss_list画出损失函数图像
'''
def plt_show_loss(epoch_list, loss_list):
    plt.plot(epoch_list, loss_list, 'r')
    plt.xlabel('epoch')
    plt.ylabel('loss')
    plt.show()
```

In [323]:

```
'''
    定义目标判别函数
'''
def aim_discriminant(x):
    return 3 * x + 2
```

In [324]:

```
'''
    在[0,1]上根据目标判别函数生成sample_N个二维数据点集，并添加一个N(0,1)的高斯噪声。
'''
def genData(sample_N, naive):
    n = np.ceil(sample_N / 2).astype(np.int32)
    lam = 0.2 # 随机变量方差
    cov_xy = 0.01 # 两个维度的协方差
    x_mean1 = [-0.6, -0.4] # 类别1的均值
    x_mean2 = [0.6, 0.4] # 类别2的均值
    train_x = np.zeros((sample_N, 2))
    train_y = np.zeros(sample_N)
    if naive: # 满足朴素贝叶斯假设
        train_x[:n,:] = np.random.multivariate_normal(x_mean1, [[lam, 0], [0,
lam]], size=n)
        train_x[n:,:] = np.random.multivariate_normal(x_mean2, [[lam, 0], [0,
lam]], size=sample_N-n)
        train_y[:n] = 0
        train_y[n:] = 1
    else: # 不满足朴素贝叶斯假设
        train_x[:n,:] = np.random.multivariate_normal(
            x_mean1, [[lam, cov_xy], [cov_xy, lam]], size=n)
        train_x[n:,:] = np.random.multivariate_normal(
            x_mean2, [[lam, cov_xy], [cov_xy, lam]], size=sample_N-n)
        train_y[:n] = 0
        train_y[n:] = 1
    return train_x, train_y
```

In [325]:

```
'''
    sigmoid函数
'''
def sigmoid(x):
    return 1 / (1 + np.exp(-x))
```

In [326]:

```
'''
    极大条件似然
'''
def likelihood(train_x, train_y, weight):
    total = np.size(train_x, axis=0)
    predict = np.zeros((total, 1))
    for i in range(total):
        predict[i] = np.dot(weight, train_x[i].T)
    t = 0
    for i in range(total):
        t += np.log(1 + np.exp(predict[i]))
    return np.dot(train_y, predict) - t
```

In [327]:

```
'''
    根据train_x和train_y对参数w做梯度下降，并返回w
    训练最大轮数为epoch
    学习率eta
    迭代误差eps
    数据集样本维度dimension
    正则项超参数lam
'''
def descent_gradient(train_x, train_y, epoch, eta, eps, dimension, lam):
    total = np.size(train_x, axis=0)
    weight = np.ones((1, dimension + 1))
    epoch_list = np.zeros(epoch)
    loss_list = np.zeros(epoch)
    for i in range(epoch):
        old_loss = - 1 / total * likelihood(train_x, train_y, weight)
        t = np.zeros((total, 1))
        for j in range(total):
            t[j] = np.dot(weight, train_x[j].T)
        gradient = - 1 / total * np.dot(train_y - sigmoid(t.T), train_x)
        weight = weight - eta * lam * weight - eta * gradient # 梯度下降
        new_loss = - 1 / total * likelihood(train_x, train_y, weight)
        epoch_list[i] = i
        loss_list[i] = new_loss
        if i % 100 == 0:
            print(i, ', loss=', new_loss, ', weight=', weight, 'gradient=',
gradient)
        if abs(new_loss - old_loss) < eps:
            epoch_list = epoch_list[:i+1]
            loss_list = loss_list[:i+1]
            break
    return weight, epoch_list, loss_list
```

In [328]:

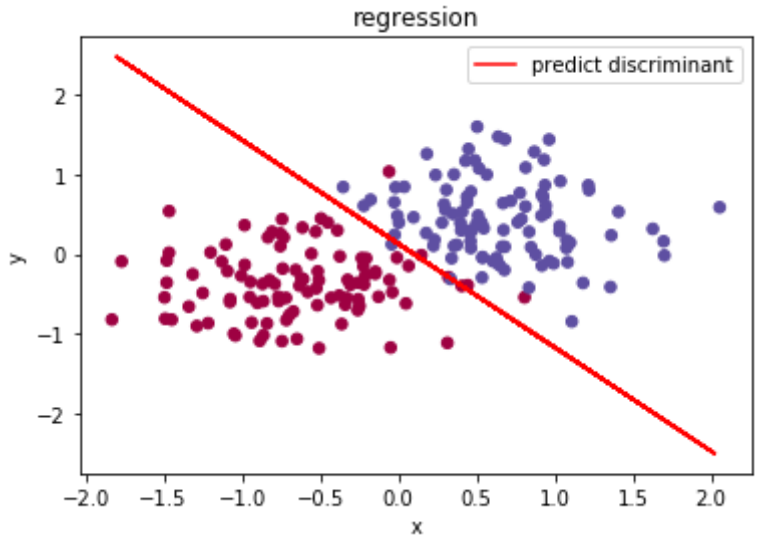
```
def custom_exp(sample_N, lam, naive):
    train_x, train_y = genData(sample_N, naive) # 生成数据
    # 构造X矩阵
    train_X = np.ones((sample_N, 3))
    train_X[:,1] = train_x[:,0]
    train_X[:,2] = train_x[:,1]
    # 规定梯度下降参数
    epoch = 1000000
    eta = 0.1
    eps = 1e-5
    # 执行训练
    weight, epoch_list, loss_list = descent_gradient(
        train_X, train_y, epoch, eta, eps, np.size(train_x, axis=1), lam)
    # 训练得到的weight是一个shape=(1, dimension=3)的矩阵
    # 需要先将weight改成行向量，对weight[dimension-1=2]做归一化，移项后得到判别函数解析式
predict_discriminant
    weight = weight.reshape(3)
    coefficient = -(weight / weight[2])[0:2]
    predict_discriminant = np.poly1d(coefficient[::-1])
    print('The predict discriminant function: y = ', predict_discriminant)
    plt_show(train_x, train_y, predict_discriminant)
    plt_show_loss(epoch_list, loss_list)
```

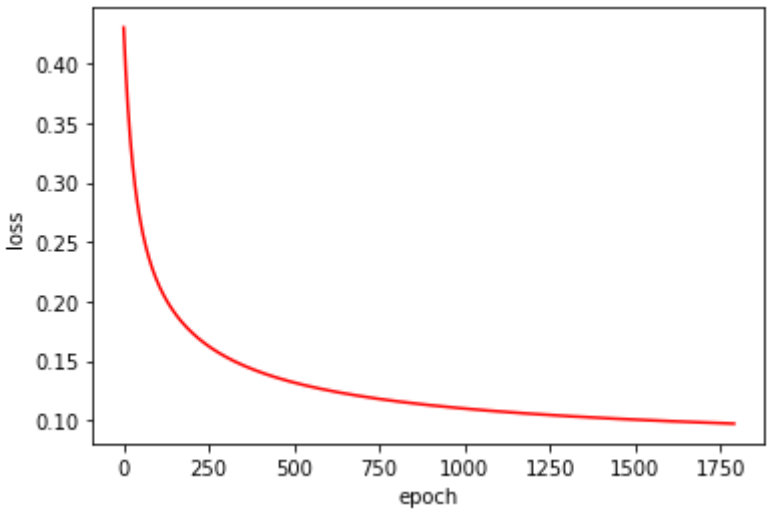
In [329]:

```
naive = True
custom_exp(200, 0, naive)
```

```
0 , loss= [0.4304008] , weight= [[0.98125076 1.01630604 1.00874562]] gradient=
[[ 0.18749236 -0.16306042 -0.08745617]]
100 , loss= [0.21579879] , weight= [[0.15375089 1.98183728 1.5434698 ]]
gradient= [[ 0.03386898 -0.06460842 -0.03764196]]
200 , loss= [0.17392498] , weight= [[-0.04790915  2.50283457  1.8585775 ]]
gradient= [[ 0.01208833 -0.04299544 -0.02701478]]
300 , loss= [0.1533995] , weight= [[-0.13546027  2.87486861  2.09959679]]
gradient= [[ 0.00652745 -0.03251322 -0.02172079]]
400 , loss= [0.14064186] , weight= [[-0.18926273  3.16569717  2.29904319]]
gradient= [[ 0.00454145 -0.02617619 -0.01842447]]
500 , loss= [0.13179526] , weight= [[-0.22956812  3.40466265  2.47111642]]
gradient= [[ 0.00362854 -0.02191251 -0.01613682]]
600 , loss= [0.12523932] , weight= [[-0.26305068  3.60752344  2.62351939]]
gradient= [[ 0.00311444 -0.0188446  -0.01443713]]
700 , loss= [0.12015493] , weight= [[-0.29238875  3.78377948  2.7609498 ]]
gradient= [[ 0.00277655 -0.01653082 -0.01311289]]
800 , loss= [0.11607851] , weight= [[-0.31884905  3.93961222  2.88650552]]
gradient= [[ 0.00252907 -0.01472363 -0.01204449]]
900 , loss= [0.11272622] , weight= [[-0.34312208  4.07927295  3.00234953]]
gradient= [[ 0.00233437 -0.01327328 -0.01115918]]
1000 , loss= [0.10991369] , weight= [[-0.36563198  4.20581209  3.11005965]]
gradient= [[ 0.0021739  -0.01208381 -0.01041003]]
```

```
1100 , loss= [0.10751556] , weight= [[-0.3866652   4.32149304  3.21082759]]
gradient= [[ 0.00203753 -0.01109085 -0.00976533]]
1200 , loss= [0.10544331] , weight= [[-0.40642963  4.42804169  3.30557911]]
gradient= [[ 0.00191917 -0.01024958 -0.0092028 ]]
1300 , loss= [0.10363255] , weight= [[-0.42508422  4.52680404  3.39505036]]
gradient= [[ 0.00181489 -0.00952784 -0.00870632]]
1400 , loss= [0.10203516] , weight= [[-0.44275527  4.61885002  3.47983827]]
gradient= [[ 0.00172197 -0.00890195 -0.00826386]]
1500 , loss= [0.10061446] , weight= [[-0.45954596  4.70504396  3.56043501]]
gradient= [[ 0.00163843 -0.00835407 -0.0078663 ]]
1600 , loss= [0.09934191] , weight= [[-0.4755423   4.786094    3.63725218]]
gradient= [[ 0.0015628  -0.00787053 -0.00750653]]
1700 , loss= [0.09819494] , weight= [[-0.49081721  4.86258742  3.71063836]]
gradient= [[ 0.00149391 -0.00744064 -0.00717895]]
The predict discriminant function: y =
-1.306 x + 0.1336
```

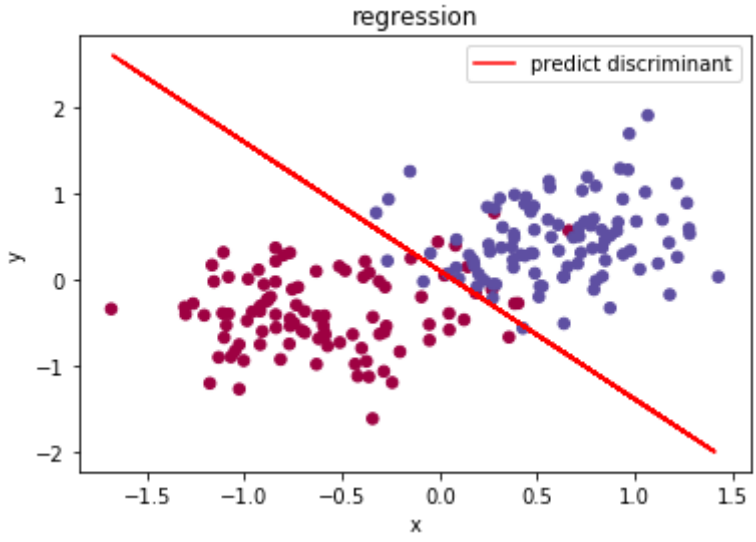


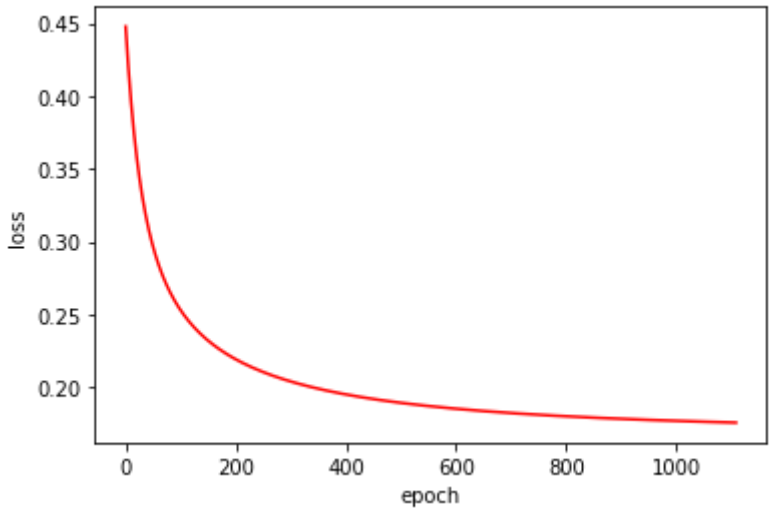


In [330]:

```
custom_exp(200, 0, not naive)
```

```
0 , loss= [0.44767362] , weight= [[0.98134961 1.0149551  1.00886388]] gradient=
[[ 0.18650389 -0.14955098 -0.08863884]]
100 , loss= [0.25277418] , weight= [[0.17099959 1.9038858  1.5197312 ]]
gradient= [[ 0.03175935 -0.05918811 -0.03336301]]
200 , loss= [0.21918262] , weight= [[-0.01093261  2.37830773  1.78552781]]
gradient= [[ 0.01031235 -0.03888295 -0.02166304]]
300 , loss= [0.20394597] , weight= [[-0.08281144  2.71256561  1.97111081]]
gradient= [[ 0.00514625 -0.02900305 -0.01605328]]
400 , loss= [0.19507939] , weight= [[-0.1242652   2.97020796  2.11345237]]
gradient= [[ 0.00343011 -0.02301869 -0.01269782]]
500 , loss= [0.18930075] , weight= [[-0.15437796  3.17884807  2.22845179]]
gradient= [[ 0.00268614 -0.01898826 -0.01045999]]
600 , loss= [0.18526553] , weight= [[-0.1790047   3.35334666  2.32456196]]
gradient= [[ 0.00227633 -0.01608608 -0.00885982]]
700 , loss= [0.18231064] , weight= [[-0.20031526  3.50266831  2.40682382]]
gradient= [[ 0.00200353 -0.01389552 -0.00765763]]
800 , loss= [0.18007005] , weight= [[-0.21927072  3.63264514  2.47848539]]
gradient= [[ 0.00179774 -0.01218282 -0.00672042]]
900 , loss= [0.17832509] , weight= [[-0.23637914  3.74728646  2.54176256]]
gradient= [[ 0.00163073 -0.01080659 -0.00596855]]
1000 , loss= [0.17693717] , weight= [[-0.25195554  3.84946819  2.59823543]]
gradient= [[ 0.00148958 -0.0096763  -0.00535146]]
1100 , loss= [0.17581423] , weight= [[-0.26622086  3.9413244   2.6490713 ]]
gradient= [[ 0.00136748 -0.00873132 -0.00483555]]
The predict discriminant function: y =
-1.488 x + 0.1008
```

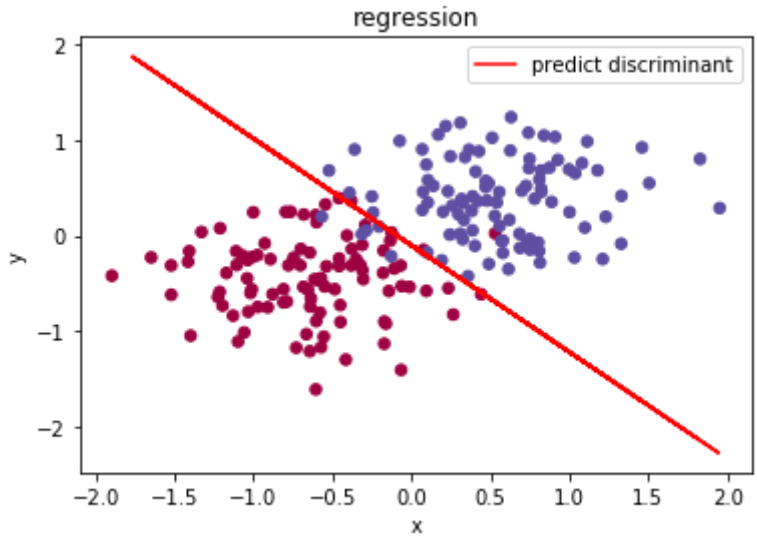


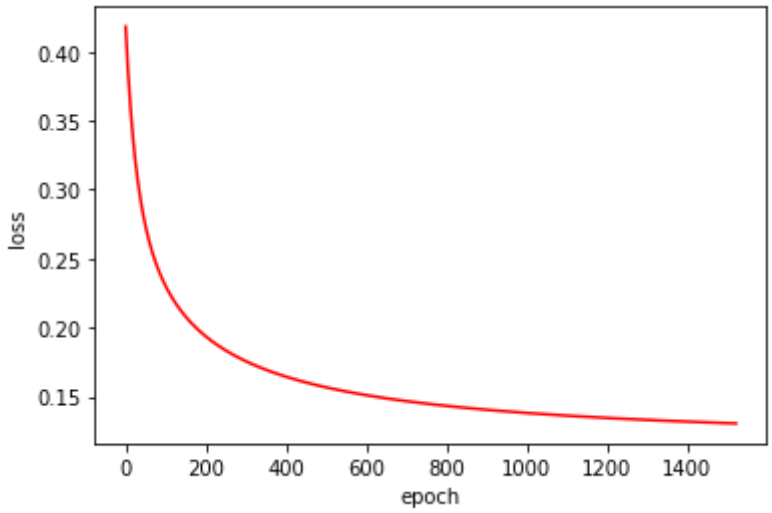


In [331]:

```
custom_exp(200, 0.001, naive)
```

```
0 , loss= [0.41819725] , weight= [[0.98325281 1.01554138 1.01049656]] gradient=
[[ 0.16647192 -0.15641375 -0.10596565]]
100 , loss= [0.22993378] , weight= [[0.32668909 1.91238265 1.63804164]]
gradient= [[ 0.02000762 -0.05999575 -0.04434853]]
200 , loss= [0.19363574] , weight= [[0.23719617 2.37096016 1.98763447]]
gradient= [[ 0.0023992  -0.03941058 -0.03131872]]
300 , loss= [0.17561461] , weight= [[0.23257258 2.68628258 2.24520772]]
gradient= [[-0.0010566  -0.02978066 -0.02509444]]
400 , loss= [0.16441877] , weight= [[0.24538918 2.92510422 2.45221576]]
gradient= [[-0.00178089 -0.0240893  -0.02131405]]
500 , loss= [0.15670602] , weight= [[0.26125366 3.11556363 2.62617054]]
gradient= [[-0.00184802 -0.0203175  -0.01872883]]
600 , loss= [0.15104385] , weight= [[0.27660927 3.27254182 2.77639282]]
gradient= [[-0.00175049 -0.01763607 -0.01682726]]
700 , loss= [0.14670015] , weight= [[0.29062957 3.40494946 2.90852163]]
gradient= [[-0.00162142 -0.01563619 -0.01535753]]
800 , loss= [0.14325831] , weight= [[0.30323937 3.51857222 3.02627004]]
gradient= [[-0.00149781 -0.01409154 -0.01418025]]
900 , loss= [0.14046255] , weight= [[0.31456343 3.61738917 3.13223643]]
gradient= [[-0.00138854 -0.01286615 -0.01321156]]
1000 , loss= [0.13814662] , weight= [[0.32476344 3.70425754 3.22832483]]
gradient= [[-0.00129417 -0.01187326 -0.01239776]]
1100 , loss= [0.13619749] , weight= [[0.33399084 3.78129915 3.31598165]]
gradient= [[-0.00121296 -0.0110548  -0.01170271]]
1200 , loss= [0.13453544] , weight= [[0.34237578 3.85013267 3.39633794]]
gradient= [[-0.00114283 -0.0103704  -0.01110109]]
1300 , loss= [0.13310255] , weight= [[0.35002715 3.91202056 3.47029935]]
gradient= [[-0.00108189 -0.00979113 -0.01057462]]
1400 , loss= [0.13185564] , weight= [[0.35703556 3.96796588 3.5386054 ]]
gradient= [[-0.00102858 -0.00929574 -0.01010966]]
1500 , loss= [0.13076185] , weight= [[0.36347668 4.01877817 3.60187012]]
gradient= [[-0.00098162 -0.00886822 -0.00969585]]
The predict discriminant function: y =
-1.115 x - 0.1009
```



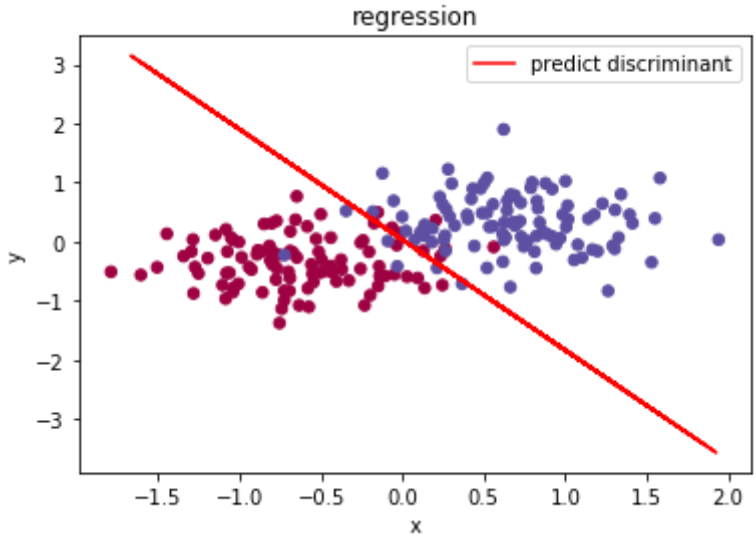


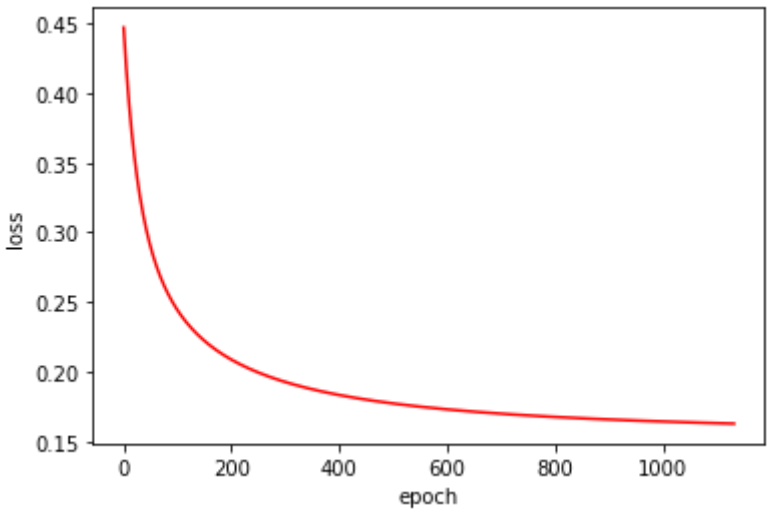
In [332]:

```
custom_exp(200, 0.001, not naive)
```

```
0 , loss= [0.44684712] , weight= [[0.98151464 1.01674229 1.00736259]] gradient=
[[ 0.18385361 -0.16842293 -0.07462593]]
100 , loss= [0.2441195] , weight= [[0.19927722 2.00899531 1.40994104]] gradient=
[[ 0.02885945 -0.06742307 -0.02635343]]
200 , loss= [0.20875728] , weight= [[0.04125889 2.52724515 1.60478908]]
gradient= [[ 0.008093   -0.04444549 -0.0172276 ]]
300 , loss= [0.19272943] , weight= [[-0.01000715  2.88367967  1.73745912]]
gradient= [[ 0.003186   -0.03345141 -0.01311828]]
400 , loss= [0.18342459] , weight= [[-0.03249225  3.15244185  1.83772451]]
gradient= [[ 0.00163373 -0.02690105 -0.01072087]]
500 , loss= [0.177371] , weight= [[-0.04498316  3.36550819  1.91766404]]
gradient= [[ 0.00103331 -0.02254515 -0.00913766]]
600 , loss= [0.17314957] , weight= [[-0.05327488  3.53993549  1.98354454]]
gradient= [[ 0.00075801 -0.01944124 -0.00801041]]
700 , loss= [0.17006162] , weight= [[-0.05947835  3.6859632   2.0390772 ]]
gradient= [[ 0.00061048 -0.01712068 -0.00716576]]
```

```
800 , loss= [0.16772179] , weight= [[-0.06447239  3.81024314  2.08665708]]
gradient= [[ 0.00051966 -0.0153234  -0.00650902]]
900 , loss= [0.1659001] , weight= [[-0.06867018  3.91735152  2.12792752]]
gradient= [[ 0.00045712 -0.01389329 -0.00598394]]
1000 , loss= [0.16445088] , weight= [[-0.07229015  4.01057738  2.16406941]]
gradient= [[ 0.00041035 -0.0127309  -0.00555489]]
1100 , loss= [0.16327752] , weight= [[-0.07546132  4.09236754  2.19596239]]
gradient= [[ 0.00037335 -0.01176982 -0.00519815]]
The predict discriminant function: y =
-1.866 x + 0.03463
```





In [333]:

```
'''
    生成skin数据集数据
'''
def skin_gen_data():
    load_data = np.loadtxt('./Skin_NonSkin.txt', dtype=np.int32)
    np.random.shuffle(load_data) # 打乱原数据，以便分成训练集和测试集
    test_data_rate = 0.2 # 测试集比例
    load_data_size = np.size(load_data, axis=0)
    train_data = load_data[:int(test_data_rate * load_data_size), :] # 训练集数据
```

```
    test_data = load_data[int(test_data_rate * load_data_size):, :] # 测试集数据
    dim = np.size(load_data, axis=1) - 1 # 训练集样本维度

    step = 50 # 本数据集太大，采用步长为50的方式，选取打乱的数据
    train_x = train_data[:,0:dim]
    train_x = train_x[::step]
    train_x = train_x - 100 # 对样本点进行坐标平移
    train_y = train_data[:,dim:dim + 1] - 1
    train_y = train_y[::step]
    train_size = np.size(train_x, axis=0)
    train_y = train_y.reshape(train_size) # 矩阵转化为行向量

    test_x = test_data[:,0:dim]
    test_x = test_x[::step] - 100  # 对样本点进行坐标平移
    test_y = test_data[:,dim:dim + 1] - 1
    test_y = test_y[::step]
    test_size = np.size(test_x, axis=0)
    test_y = test_y.reshape(test_size) # 矩阵转化为行向量
    return train_x, train_y, test_x, test_y
```

In [334]:

```
'''
    由于skin数据集是三维的，可以画出三维立体图像
'''
from mpl_toolkits.mplot3d import Axes3D
def skin_show_3D(train_x, train_y, coefficient):
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.scatter(train_x[:,0], train_x[:,1], train_x[:,2], c=train_y,
cmap=plt.cm.Spectral)
    real_x = np.arange(np.min(train_x[:,0]), np.max(train_x[:,0]), 1)
    real_y = np.arange(np.min(train_x[:,1]), np.max(train_x[:,1]), 1)
    real_X, real_Y = np.meshgrid(real_x, real_y)
    real_z = coefficient[0] + coefficient[1] * real_X + coefficient[2] * real_Y
    ax.plot_surface(real_x, real_y, real_z, rstride=1, cstride=1)
    ax.set_zlim(np.min(real_z) - 10, np.max(real_z) + 10)
    ax.legend(loc='best')
    plt.show()
```

In [335]:

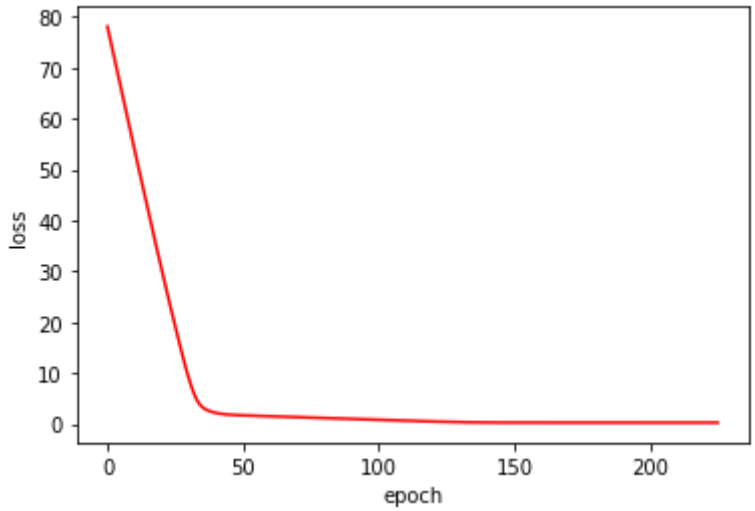
```
'''
    用uci上Skin_NonSkin.txt数据集实验
'''
def skin_exp(lam):
    train_x, train_y, test_x, test_y = skin_gen_data()
    train_size = np.size(train_x, axis=0) # 训练样本数量
    test_size = np.size(test_x, axis=0) # 测试样本数量
    dim = np.size(train_x, axis=1) # 样本维度

    # 构造训练集矩阵
    train_X = np.ones((train_size, dim + 1))
    train_X[:,1] = train_x[:,0]
    train_X[:,2] = train_x[:,1]
    train_X[:,3] = train_x[:,2]
```

```
    # 设置参数，开始训练
    epoch = 1000000
    eta = 0.001
    eps = 1e-5
    weight, epoch_list, loss_list = descent_gradient(train_X, train_y, epoch,
eta, eps, dim, lam)
    weight = weight.reshape(dim + 1) # 训练结果参数重塑
    coefficient = - (weight / weight[dim])[0:dim] # 得到决策面方程系数
    # 画图
    plt_show_loss(epoch_list, loss_list)
    skin_show_3D(train_x, train_y, coefficient)
    # 计算测试集的准确率
    label = np.ones(test_size)
    correct_count = 0
    test_X = np.ones((test_size, dim + 1))
    test_X[:,1] = test_x[:,0]
    test_X[:,2] = test_x[:,1]
    test_X[:,3] = test_x[:,2]
    for i in range(test_size):
        if np.dot(weight, test_X[i].T) >= 0:
            label[i] = 1
        else:
            label[i] = 0
        if label[i] == test_y[i]:
            correct_count += 1
    correct_rate = correct_count / test_size
    print(correct_rate)
```

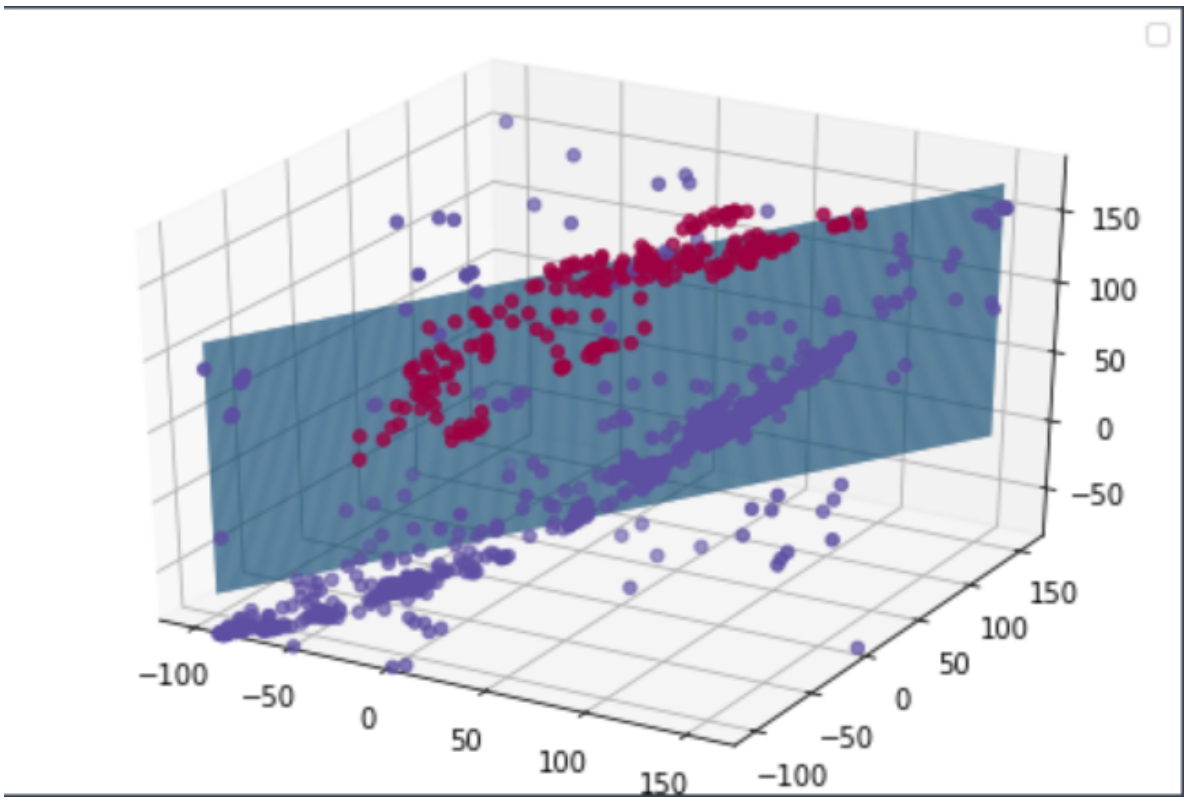
In [336]:

```
skin_exp(0)
```

```
0 , loss= [78.0732635] , weight= [[1.00006045 0.98252067 0.97690828 0.96000533]]
gradient= [[-0.06045196 17.47932615 23.09172081 39.99466543]]
100 , loss= [0.80464362] , weight= [[ 1.00863631  0.09626433  0.08062129
-0.16601919]] gradient= [[-0.07696413  2.61200237  0.66825174 -3.13056851]]
200 , loss= [0.25784807] , weight= [[ 1.01549561  0.00858208  0.0303685
-0.04090911]] gradient= [[-0.06732764 -0.07439327  0.11880689 -0.04801726]]
```




```
No handles with labels found to put in legend.
```

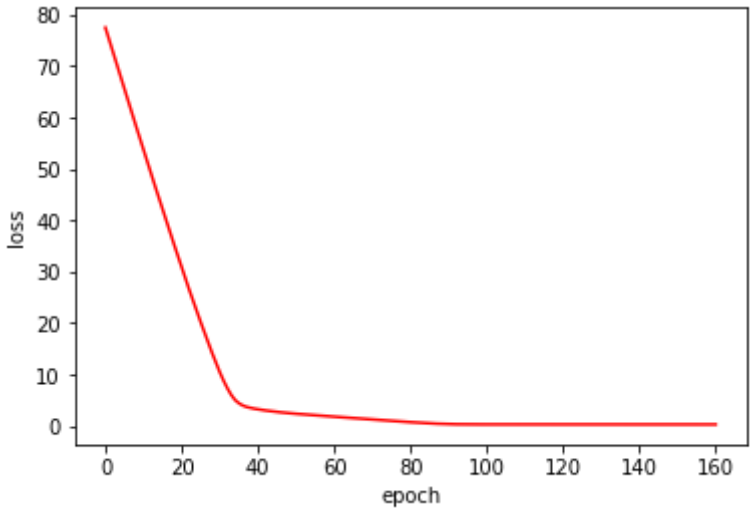



```
0.9466972711043101
```

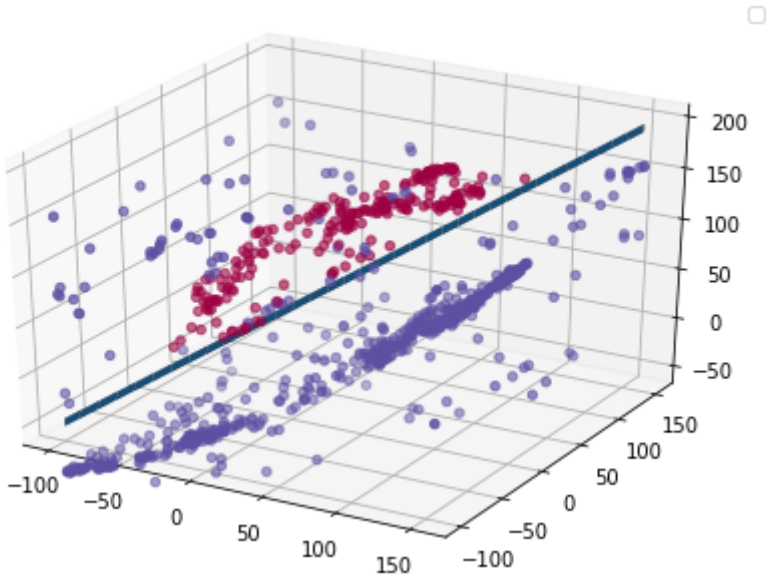
In [337]:

```
skin_exp(0.01)
```

```
0 , loss= [77.38886887] , weight= [[1.00004208 0.98305931 0.97690587
0.96016225]] gradient= [[-0.05208352 16.93068734 23.08412807 39.82774965]]
100 , loss= [0.34776756] , weight= [[ 1.00989463  0.03055627 -0.01119228
-0.02029575]] gradient= [[-0.11392781  0.81555458 -0.46662685 -0.24005138]]
```




```
No handles with labels found to put in legend.
```




```
0.9380260137719969
```

In [338]:

```
'''
    生成banknote数据集数据
'''
def banknote_gen_data():
```

```
    load_data = np.loadtxt('./data_banknote_authentication.txt',
dtype=np.float32)
    np.random.shuffle(load_data) # 打乱原数据，以便分成训练集和测试集
    test_data_rate = 0.2 # 测试集比例
    load_data_size = np.size(load_data, axis=0) # 数据数量
    train_data = load_data[:int(test_data_rate * load_data_size), :] # 训练集数据
    test_data = load_data[int(test_data_rate * load_data_size):, :] # 测试集数据
    dim = np.size(load_data, axis=1) - 1 # 训练集样本维度

    train_x = train_data[:,0:dim]
    train_y = train_data[:,dim:dim + 1] - 1
    train_size = np.size(train_x, axis=0)
    train_y = train_y.reshape(train_size) # 矩阵转化为行向量

    test_x = test_data[:,0:dim]
    test_y = test_data[:,dim:dim + 1] - 1
    test_size = np.size(test_x, axis=0)
    test_y = test_y.reshape(test_size) # 矩阵转化为行向量
    return train_x, train_y, test_x, test_y
```

In [339]:

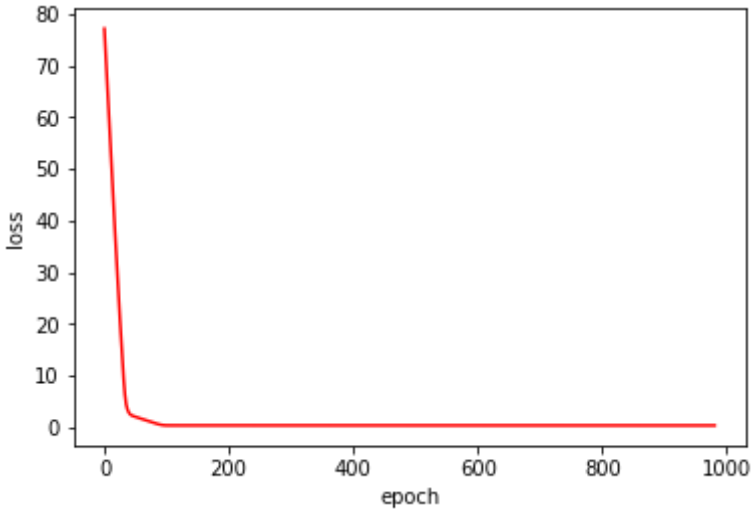
```
'''
    根据banknote数据集实验
'''
def banknote_exp(lam):
    train_x, train_y, test_x, test_y = uci_gen_data()
    train_size = np.size(train_x, axis=0) # 训练集大小
    test_size = np.size(test_x, axis=0) # 测试集大小
    dim = np.size(train_x, axis=1) # 训练集样本维度
    # 构造训练集样本矩阵
    train_X = np.ones((train_size, dim + 1))
    for i in range(dim):
        train_X[:,i + 1] = train_x[:, i]
    # 设置参数，开始训练
    epoch = 1000000
    eta = 0.001
    eps = 1e-5
    weight, epoch_list, loss_list = descent_gradient(train_X, train_y, epoch,
eta, eps, dim, lam)
    weight = weight.reshape(dim + 1)
    coefficient = - (weight / weight[dim])[0:dim] # 得到决策面方程系数
    # 画损失函数图像
    plt_show_loss(epoch_list, loss_list)
    # 计算测试集准确率
    label = np.ones(test_size)
    correct_count = 0
    test_X = np.ones((test_size, dim + 1))
    for i in range(dim):
        test_X[:,i + 1] = test_x[:,i]
    for i in range(test_size):
        if np.dot(weight, test_X[i].T) >= 0:
            label[i] = 1
        else:
            label[i] = 0
        if label[i] == test_y[i]:
            correct_count += 1
```

```
    correct_rate = correct_count / test_size
    print(correct_rate)
```

In [340]:

```
banknote_exp(0)
```

```
0 , loss= [77.13798766] , weight= [[1.0001202  0.98208272 0.97701579
0.96130074]] gradient= [[-0.12020095 17.91728485 22.98421046 38.69926347]]
100 , loss= [0.32413507] , weight= [[ 1.0121294   0.02252103  0.00370694
-0.02808576]] gradient= [[-0.11001286  1.11397612  0.40950939 -1.34369195]]
200 , loss= [0.31811913] , weight= [[ 1.02302295  0.0173383   0.00385723
-0.02385568]] gradient= [[-1.08002147e-01 -1.87628320e-05  4.89474639e-04
1.51656178e-04]]
300 , loss= [0.31696465] , weight= [[ 1.03376676  0.01740681  0.00372502
-0.02384976]] gradient= [[-0.10690092 -0.00085615  0.00153195 -0.0001108 ]]
400 , loss= [0.31583308] , weight= [[ 1.04440306  0.01749284  0.00357182
-0.02383889]] gradient= [[-1.05838525e-01 -8.58820787e-04  1.52524905e-03
-1.04923621e-04]]
500 , loss= [0.31472382] , weight= [[ 1.05493396  0.01757838  0.00342022
-0.02382884]] gradient= [[-1.04792670e-01 -8.52051066e-04  1.50698368e-03
-9.62876568e-05]]
600 , loss= [0.31363632] , weight= [[ 1.06536109  0.01766324  0.00327044
-0.02381963]] gradient= [[-1.03762813e-01 -8.45175745e-04  1.48886882e-03
-8.78871294e-05]]
700 , loss= [0.31257003] , weight= [[ 1.07568603  0.01774741  0.00312246
-0.02381126]] gradient= [[-1.02748658e-01 -8.38305308e-04  1.47103372e-03
-7.97475791e-05]]
800 , loss= [0.31152443] , weight= [[ 1.08591033  0.01783089  0.00297624
-0.02380368]] gradient= [[-1.01749918e-01 -8.31444891e-04  1.45347565e-03
-7.18610933e-05]]
900 , loss= [0.310499] , weight= [[ 1.09603552  0.01791369  0.00283177
-0.02379689]] gradient= [[-1.00766311e-01 -8.24598199e-04  1.43619043e-03
-6.42196595e-05]]
```

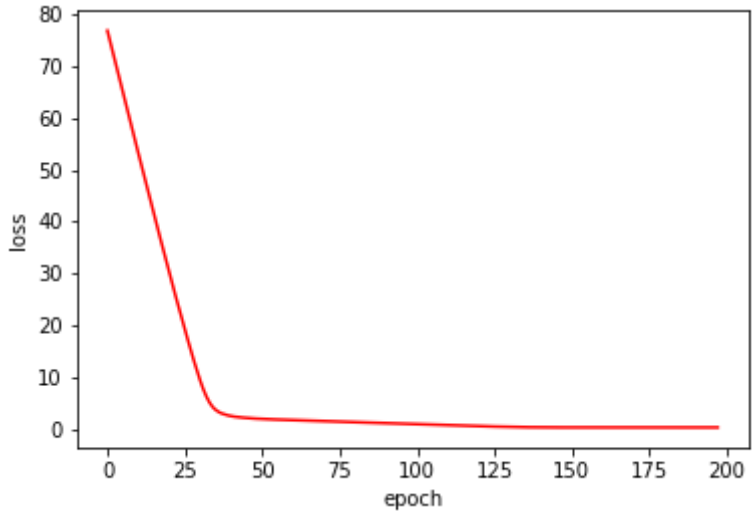


```
0.9382810507523591
```

In [341]:

```
banknote_exp(0.01)
```

```
0 , loss= [76.83953028] , weight= [[1.00004489 0.9832704  0.97733132 0.9600917
]] gradient= [[-0.05488676 16.71959601 22.65868104 39.89830429]]
100 , loss= [0.89789853] , weight= [[ 1.00753468  0.13236574  0.04148546
-0.15919396]] gradient= [[-0.07937962  2.44036901  1.22793437 -3.45650807]]
```




```
0.9410864575363428
```

In [ ]:

```

```