2020 年春季学期
计算学部《机器学习》课程

Lab 2 实验报告

姓名	余涛
学号	1180300829
班号	1803202
电子邮件	1063695334@qq.com
手机号码	15586430583

目录

1 实验内容

1.1 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。

1.2 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。

1.3 实验环境

Windows 10 专业版；python 3.8.6；PyCharm Community Edition 2020.2.2 x64

2 实验设计思想

2.1 算法原理

2.1.1 手工生成 0/1 数据集（是否满足贝叶斯假设）

首先设置随机变量方差，然后设置两个维度的协方差，设置类别 1 的两个均值和类别 2 的两个均值，然后创建一个二维的点集和点集的分类集。对于二维点集，根据是否满足贝叶斯假设，对前一半创建第一类数据集，对后一半创建第二类数据集并且都加上一个 N(0,1)的高斯噪声，最后给分类集赋值 0 或者 1。

2.1.2 梯度下降法实现逻辑回归（无惩罚项和有惩罚项）

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 = <X1,⋯,Xn>)，根据朴素贝叶斯的方法，可以用贝叶斯概率公式，将上式转化为条件概率和类概率的乘积，即利用 P(Y)，P(X|Y)和各个维度之间的计算条件独立的假设来计算 P(Y|X)。本次实验却不需要这么复杂，而是直接求解这个概率。

对于 0/1 分类来说，可以有以下推导：

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

对分母 P(X)进行变换可得：

右边=

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

上下同除 P(Y=1)P(X|Y=1)得：

右边=

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

继续变换可得：

右边=

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

令

$\pi=P(Y=1)$, $1-\pi=P(Y=0)$

这时右边=

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

假设类条件分布服从正态分布并且方差不依赖于 k，则有

$$P(x|y_k)=\frac{1}{\sigma_i\sqrt{2\pi}}e^{\frac{-(x-\mu_{ik})^2}{2\sigma_i^2}}$$

此时右边=

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

=

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

=

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

=

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

令

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

则有：

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

sigmoid 函数：这个函数将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 的离散值：

$$S(x)=\frac{1}{1+e^{-x}}$$

其作用是将实数值映射到 0 到 1 之间的某个值，在 x=0 处函数值取 0.5，并且以 x=0 为界，函数值分别向 1 和 0 逼近。

此时的

$$P(Y=1|X)=\frac{1}{1+exp(\mathbf{w}^{\mathbf{T}}\mathbf{X})}$$

由于 P(Y = 0 | X)=1-P(Y=1 | X),则有：

$$P(Y=0|X)=\frac{exp(\mathbf{w}^{\mathbf{T}}\mathbf{X})}{1+exp(\mathbf{w}^{\mathbf{T}}\mathbf{X})}$$

两者相除=1 时即可以得到分类决策面(线)，即有：

$exp(\mathbf{w}^{\mathbf{T}}\mathbf{X})$=1，此时

$$w^Tx+b=0$$

即是分类决策面(线)，所以需要求解参数 w。

我们使用梯度下降法来获得逻辑回归的参数 w，参数为 wi，0≤i≤n，n 是样本集的维度。

需要计算 P(< X,Y>|w),可以转化为极大条件似然估计(MCLE)，只需要计算 P(X|Y, w)，有 MCLE 的推导式：

l(w)=

$$\sum_l Y^l lnP(Y^l=1|X^l,W)+(1-Y^l)lnP(Y^l=0|X^l,W)$$

=

$$\sum_l Y^l ln\frac{P(Y^l=1|X,W)}{P(Y^l=0|X^l,W)}+lnP(Y^l=0|X^l,W)$$

=

$$\sum_l Y^l(w_0+\sum_i^n w_iX_i^l)-ln(1+exp(w_0+\sum_i^n w_iX_i^l))$$

需要求得 MCLE 最大时对应的 w 值，即求解 $argmax_w l(w)$ 的 w 值，对 l(w)取反。此时就可以将-l(w)作为梯度下降法的损失函数，由于数据特别多的情况下，可能导致溢出，所以需要对-l(w)添加一个系数 1/L（L 为样本的总数），就可以得到此时的损失函数:

l(w)=$\frac{1}{L}$*

$$\sum_l Y^l(w_0+\sum_i^n w_i X_i^l)-ln(1+exp(w_0+\sum_i^n w_i X_i^l))$$

然后对损失函数求偏导数，有:

$$\frac{\partial l(w)}{\partial w_i}=\sum_l X_i^l(Y^l-\frac{1}{1+exp(-(w_0+\sum_i^n w_i X_i^l))})$$

可以得到梯度下降的迭代公式为:

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

为了避免出现过拟合的现象，可以增加惩罚项，增加惩罚项后的梯度下降的迭代公式为:

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

然后就可以通过迭代来完成对参数 w 的估计，进而得到分界决策面(线)，完成分类。

2.1.3 UCI 数据集样本太多时取部分数据进行试验

在进行 Skin_NonSkin.csv 进行逻辑回归试验时，由于该数据集含有几十万条数据，数据集的样本太大，需要将其先打乱顺序，然后按照比较大的步长取部分数据作为数据集。

2.1.4 判断测试集的准确率

在求解出系数 w 后，对于测试集中每个数据点 X，只需要将w^TX的结果与 0 进行判断，就能得出 X 预测的分类。在本实验中，若$w^TX>0$，则 X 属于分类 1。若$w^TX<0$，则 X 属于分类 0。然后将测试集预测的分类正确数除实际的数据总数，就可以得到测试集的准确率。

2.1.5 数据太大时防止 exp 溢出

由于在试验中，经常调用 exp(-x)，很容易由于数据过大导致溢出（在本人的试验中，blood.csv 和 heart.csv 都出现了 exp 溢出），为了防止 exp 溢出，可以对于样本集的所有维度的数据，计算其最大值和最小值的差作为极差，然后用最大值减当前数据，然后用得到的数据除极差作为新的数据，就可以将所有的数据缩减为[0,1]区间的数据，这样就能防止 exp 溢出。

2.2 算法实现

2.2.1 手工生成 0/1 数据集（是否满足贝叶斯假设）

作用：

```
'''
自己生成数据
根据是否满足贝叶斯假设，在范围[0,1]上生成 sample_number 个二维点集，并将其均匀分为两类
并添加一个 N(0,1)的高斯噪声
'''
```

具体实现：

首先设置随机变量方差为 0.2，然后设置两个维度的协方差为 0.01，设置类别 1 的两个均值为-0.7 和-0.3，设置类别 2 的两个均值为 0.7 和 0.3，然后创建一个二维的点集 train_point_X 和点集的分类集 classification_Y。对于二维点集 train_point_X，根据是否满足贝叶斯假设，对前一半创建第一类数据集 0，对后一半创建第二类数据集 1 并且都加上一个 N(0,1)的高斯噪声，最后给分类集赋值 0 或者 1。以注释形式给出。

```
def create_datas(sample_number, satisfy_naive):
    half = np.ceil(sample_number / 2).astype(np.int32)  # sample_number 的一半
    variance = 0.2  # 随机变量方差
    covariance_xy = 0.01  # 两个维度的协方差
    point_mean1 = [-0.7, -0.3]  # 类别 1 的均值
    point_mean2 = [0.7, 0.3]  # 类别 2 的均值
    train_point_X = np.zeros((sample_number, 2))  # 二维的点集
    classification_Y = np.zeros(sample_number)  # 点集的分类
    if satisfy_naive:  # 满足朴素贝叶斯假设
        train_point_X[:half, :] = np.random.multivariate_normal(point_mean1, [[variance, 0], [0, variance]],
                                                              size=half)  # 生成 half 个类别 1 的 1*2 数组，每个数组含有两个维度
        train_point_X[half:, :] = np.random.multivariate_normal(point_mean2, [[variance, 0], [0, variance]],
                                                              size=sample_number - half)  # 生成 half 个类别 2 的 1*2 数组，每个数组含有两个维度
        classification_Y[:half] = 0  # 将前 half 个数组标记为类别 1
        classification_Y[half:] = 1  # 将后 half 个数组标记为类别 2
    else:  # 不满足朴素贝叶斯假设
```

```
        train_point_X[:half, :] =
np.random.multivariate_normal(point_mean1,
                                                              [[variance,
covariance_xy], [covariance_xy, variance]],
                                                              size=half)  # 生
成 half 个类别 1 的 1*2 数组，每个数组含有两个维度
        train_point_X[half:, :] =
np.random.multivariate_normal(point_mean2,
                                                              [[variance,
covariance_xy], [covariance_xy, variance]],

size=sample_number - half)  # 生成 half 个类别 2 的 1*2 数组，每个数组含有两个
维度
        classification_Y[:half] = 0  # 将前 half 个数组标记为类别 1
        classification_Y[half:] = 1  # 将后 half 个数组标记为类别 2
    return train_point_X, classification_Y  # 返回生成的所有点及类别
```

2.2.2 梯度下降法实现逻辑回归（无惩罚项和有惩罚项）

作用：

```
'''
加惩罚项的梯度下降法
对于 train_point_X 和 classification_Y 对参数 w 做梯度下降，对损失函数使用梯度下
降法，当误差函数收敛到期望的最小值时，得到此时的 w 并返回 w
参数中：
迭代最大次数为 cycle_times
下降步长 descending_step_size
迭代误差 iteration_error
数据点集维度 dimension
惩罚项参数 lamda
'''
```

具体实现：在上面原理中已经给出损失函数和迭代公式，通过对迭代公式

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

不断迭代，当此时的迭代误差小于规定的迭代误差时终止迭代，如果迭代后的新的迭代误差大于原迭代误差，说明此时的迭代值在解析解左右摇摆，此时需要将步长变为原来的一半。对于不需要惩罚项的迭代，只需要将惩罚项的系数传为 0 即可。以注释形式给出。

```
def descent_gradient_add_errorfunction(train_point_X,
classification_Y, cycle_times, descending_step_size,
                                        iteration_error, dimension, lamda):
    total_points = np.size(train_point_X, axis=0)  # 得到 train_point_X
的行数，即点集个数
    w = np.ones((1, dimension + 1))  # 生成系数矩阵 w，一个列数为 dimension
```

```
+ 1，行数为 1 的矩阵，元素值全为 1
    cycle_times_list = np.zeros(cycle_times)  # 迭代次数统计
    loss_list = np.zeros(cycle_times)  # 迭代次数对应的损失函数值统计
    for i in range(cycle_times):
        old_loss = - 1 / total_points *
maximum_conditional_likelihood(train_point_X, classification_Y, w)  #
原先损失函数的值
        t = np.zeros((total_points, 1))
        for j in range(total_points):
            t[j] = w.dot(train_point_X[j].T)  # 极大条件似然中的 exp 中的部
分，即求和 wi * Xl
        gradient = - 1 / total_points * (classification_Y -
sigmoid(t.T)).dot(train_point_X)
        w = w - descending_step_size * lamda * w -
descending_step_size * gradient  # 梯度下降加惩罚项的迭代公式
        new_loss = - 1 / total_points *
maximum_conditional_likelihood(train_point_X, classification_Y, w)  #
新的损失函数的值
        cycle_times_list[i] = i  # 储存迭代次数
        loss_list[i] = new_loss  # 储存每次迭代对应的损失函数值
        if abs(new_loss - old_loss) < iteration_error:  # 如果新的误差函
数值与旧的误差函数值的差小于迭代误差则终止迭代
            cycle_times_list = cycle_times_list[:i + 1]
            loss_list = loss_list[:i + 1]
            print('迭代次数=', i, ', 对应的损失函数值=', new_loss, ', 对应的
系数 w=', w, '对应的梯度=', gradient)
            break
        if new_loss > old_loss:  # 当新的误差函数值大于旧的误差函数值时，将步
长变为原来的一半
            descending_step_size *= 0.5
    return w, cycle_times_list, loss_list
```

2.2.3 UCI 数据集样本太多时取部分数据进行试验

作用：

```
'''
读入 Skin_NonSkin.csv 文件，对数据进行拆分，拆分成训练集合测试集
由于原文件中数据量巨大，所以对数据集以 50 步长取部分数据作为数据集
'''
```

具体实现：该数据集含有几十万条数据，数据集的样本太大，需要将其先打乱顺序，然后按照 50 的步长取部分数据作为数据集。然后将数据集分为训练集和测试集。测试集所占比例

为 20%。以注释形式给出。

```
def Skin_NonSkin_getdata():
    all_data = np.loadtxt(open('./Skin_NonSkin.csv'), delimiter=",",
skiprows=0)  # 读取文件中的所有数据
    np.random.shuffle(all_data)  # 将原数据集打乱，分成训练集和测试集
    test_rate = 0.2  # 测试集所占比例
    all_data_size = np.size(all_data, axis=0)  # 总数据集数据数量
    train_data_X = all_data[:int(test_rate * all_data_size), :]  # 训练
集数据
    test_data_x = all_data[int(test_rate * all_data_size):, :]  # 测试
集数据
    dimension = np.size(all_data, axis=1) - 1  # 训练集样本维度
    step = 50  # 由于 Skin_NonSkin 的数据集太大，所以采用步长为 50 的方式取数据
    train_point_X = train_data_X[:, 0:dimension]  # 将所有数据集赋给
train_point_X
    train_point_X = train_point_X[::step]  # 以 step 为间隔取数据
    train_point_X = train_point_X - 100  # 对样本点进行坐标平移，方便在 3D
图中显示
    train_classification_Y = train_data_X[:, dimension:dimension + 1]
- 1  # 因为数据集的分类是 1/2,需要减 1 变成 0/1
    train_classification_Y = train_classification_Y[::step]  # 以 step
为间隔取数据
    train_size = np.size(train_point_X, axis=0)  # 训练集数据总数
    train_classification_Y =
train_classification_Y.reshape(train_size)  # 将矩阵转化为行向量
    test_point_X = test_data_x[:, 0:dimension]  # 将所有数据集赋给
test_point_X
    test_point_X = test_point_X[::step] - 100  # 对样本点进行坐标平移，方
便在 3D 图中显示
    test_classification_Y = test_data_x[:, dimension:dimension + 1] -
1  # 因为数据集的分类是 1/2,需要减 1 变成 0/1
    test_classification_Y = test_classification_Y[::step]  # 以 step 为
间隔取数据
    test_size = np.size(test_point_X, axis=0)  # 测试集数据总数
    test_classification_Y = test_classification_Y.reshape(test_size)
# 将矩阵转化为行向量
    return train_point_X, train_classification_Y, test_point_X,
test_classification_Y
```

2.2.4 判断测试集的准确率

作用：

得到测试集命中的准确率。

具体实现：在求解出系数 w 后，对于测试集中每个数据点 X，只需要将$w^T X$的结果与 0 进行判断，就能得出 X 预测的分类。在本实验中，若$w^T X > 0$，则 X 属于分类 1。若$w^T X < 0$，则 X 属于分类 0。然后将测试集预测的分类正确数除实际的数据总数，就可以得到测试集的准确率。以注释形式给出。

```python
for i in range(test_size):  # 对每种预测给 label 进行赋值
    if w.dot(test_all[i].T) >= 0:
        label[i] = 1
    else:
        label[i] = 0
for i in range(test_size):
    if label[i] == test_classification_Y[i]:  # 如果预测的结果与真实结果相同计数加一
        hit_count += 1
hit_rate = hit_count / test_size
print('数据的测试集的准确率为: ', hit_rate)
```

2.2.5 数据太大时防止 exp 溢出

作用：

防止调用 exp(-x)发生 exp 溢出。exp 溢出如下：

```
C:\Users\Administrator\PycharmProjects\MachineLearning_Lab2\Lab2\descent_gradient_add_errorfunction.py:53: RuntimeWarning: overflow encountered in exp
  t += np.log(1 + np.exp(predict[i]))  # 极大条件似然中ln的部分，即ln ( 1 + exp ( 求和wi * Xi ) )
C:\Users\Administrator\PycharmProjects\MachineLearning_Lab2\Lab2\descent_gradient_add_errorfunction.py:7: RuntimeWarning: overflow encountered in exp
  return 1 / (1 + np.exp(-x))
```

具体实现：可以对于样本集的所有维度的数据，计算其最大值和最小值的差 max(train_data_X[:, i]) - min(train_data_X[:, i])作为极差 d_length，然后用最大值 max 减当前数据，然后用得到的数据除极差(max(train_data_X[:, i]) - train_data_X[j, i]) / d_length 作为新的数据，就可以将所有的数据缩减为[0,1]区间的数据，这样就能防止 exp 溢出。以注释形式给出。

```python
# 消除 exp 溢出，防止数据太大导致 exp 溢出
for i in range(dimension):  # 对于样本的所有维度
    d_length = max(train_data_X[:, i]) - min(train_data_X[:, i])  # 计算最大值和最小值之间的极差
    for j in range(np.size(train_data_X, axis=0)):  # 对于每一维度的所有数
        train_data_X[j, i] = (max(train_data_X[:, i]) - train_data_X[j, i]) / d_length  # 将其化为[0,1]之间的值，防止 exp 溢出
```

2.2.6 绘制逻辑回归图像

作用：

```
'''
对于 train_point_X 中，含有二维数据，分别对应横坐标 x 和纵坐标 y
对于 classification_Y,为 train_point_X 中两种数据的点对应的类别
根据 train_point_X 和 classification_Y 画出二维坐标下的点图，然后画出分界判别函数 boundary_check_function
'''
'''
Skin_NonSkin 数据集是三个维度的，所以可以画出三维的图像。
'''
```

具体实现：对于自己手工生成的二维数据，可以生成二维图像。对于 Skin_NonSkin.csv，由于是三维数据，可以绘制三维图像。其他的 UCI 数据集由于是多维的，所以并没有绘制图像以注释形式给出。

```
def draw_picture(train_point_X, classification_Y, boundary_check_function):
    if boundary_check_function:  # 绘制分界判定函数 boundary_check_function
        d_length = max(train_point_X[:, 0]) - min(train_point_X[:, 0])  # 找到横坐标 x 的极差
        real_x = np.linspace(min(train_point_X[:, 0]), min(train_point_X[:, 0]) + d_length, 50)  # 在 x 的范围内均匀产生 50 个点
        real_y = boundary_check_function(real_x)  # 对 real_x 每个点调用 boundary_check_function 求解对应的 real_y
        plt.plot(real_x, real_y, 'r', label='boundary_check_function')  # 绘制图像
    plot = plt.scatter(train_point_X[:, 0], train_point_X[:, 1], s=30, c=classification_Y, marker='o',
                       cmap=plt.cm.Spectral)  # 绘制两种点集
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title('The regression curve')
    plt.show()

def Skin_NonSkin_draw_picture(train_point_X, classification_Y, function_coefficient):
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.set_title('3D of the regression curve')
    ax.scatter(train_point_X[:, 0], train_point_X[:, 1],
```

```
train_point_X[:, 2], c=classification_Y, cmap=plt.cm.Spectral)
    real_x = np.arange(np.min(train_point_X[:, 0]),
np.max(train_point_X[:, 0]), 1)
    real_y = np.arange(np.min(train_point_X[:, 1]),
np.max(train_point_X[:, 1]), 1)
    real_X, real_Y = np.meshgrid(real_x, real_y)
    real_z = function_coefficient[0] + function_coefficient[1] *
real_X + function_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.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    plt.show()
```

2.2.7 绘制迭代次数与损失函数值关系图像

作用：

```
'''
根据迭代次数 cycle_times_list 和对应的误差 loss_list 画出损失函数图像
参数中，cycle_times_list 为迭代次数表，loss_list 为对应的误差表
'''
```

具体实现：以注释形式给出。

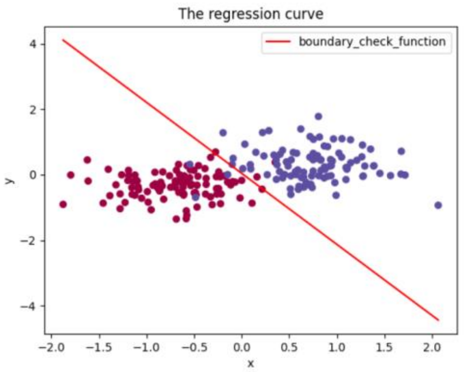
```
def draw_picture_loss(cycle_times_list, loss_list):
    plt.plot(cycle_times_list, loss_list, 'r', label='loss_function')
    plt.xlabel('cycle_times')
    plt.ylabel('loss')
    plt.legend(loc=1)
    plt.title('the loss_funciton')
    plt.show()
```

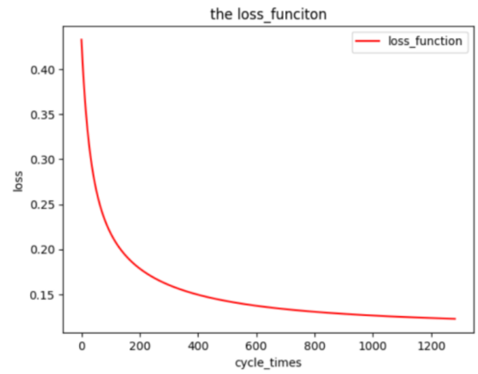
3 实验结果分析

3.1 生成满足朴素贝叶斯假设的数据

（1）不带惩罚项

生成的 0/1 数据比例为 1:1，生成 200 个测试用例，迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5

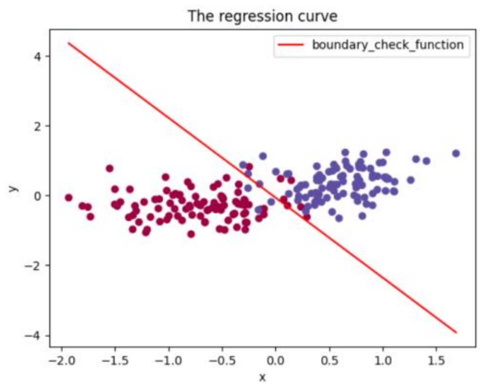


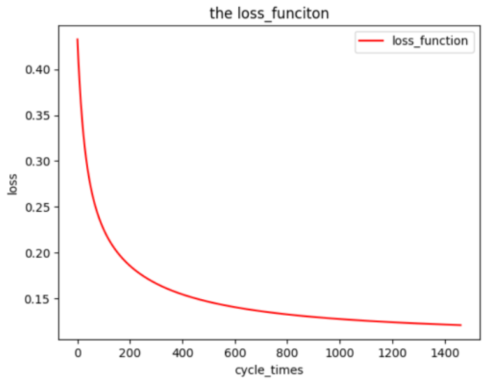



```
迭代次数= 1385
对应的损失函数值= [0.12415167]
对应的系数w= [[0.12258929 4.72804099 2.7577172 ]]
对应的梯度= [[-0.00022941 -0.00846942 -0.00531006]]
分界判别函数为： y =
-1.714 x - 0.04445
```

（2）带惩罚项

生成的 0/1 数据比例为 1:1，生成 200 个测试用例，惩罚项系数为 0.001 迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5






```
迭代次数= 1461
对应的损失函数值= [0.12102721]
对应的系数w= [[0.13064461 4.90338828 2.13986375]]
对应的梯度= [[-0.00013936 -0.01240028 -0.00390975]]
分界判别函数为： y =
-2.291 x - 0.06105
```

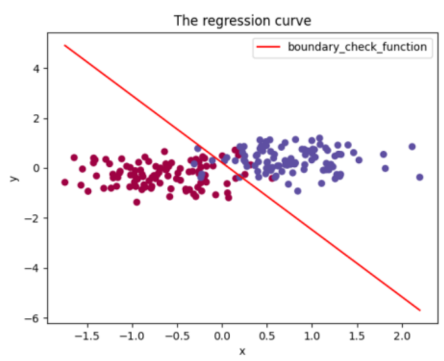
通过对图像进行分析可以得到：增加惩罚项几乎对准确率没有什么影响，但会增加一点准确率，更好的进行分类。

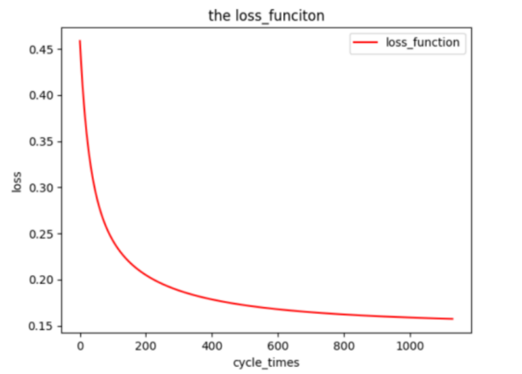
3.2 生成不满足朴素贝叶斯假设的数据

（1）不带惩罚项

生成的 0/1 数据比例为 1:1，生成 200 个测试用例，迭代最多次数为 1000000，初始步长

为 0.1，迭代误差为 1e-5

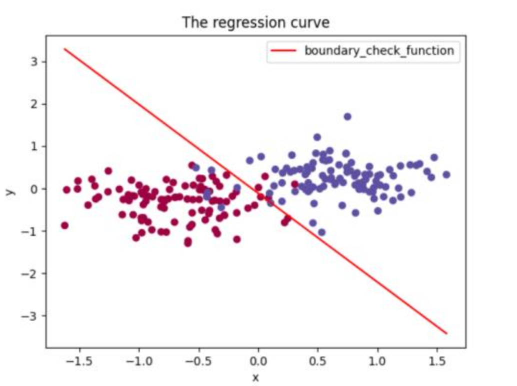


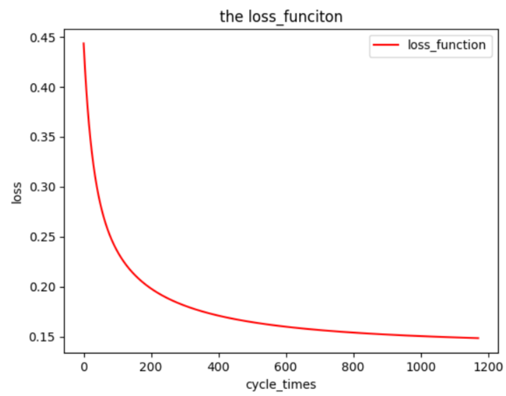



```
迭代次数= 1129
对应的损失函数值= [0.15733438]
对应的系数w= [[-0.36203409  4.48889695  1.67097651]]
对应的梯度= [[ 0.00125268 -0.00985926 -0.00110923]]
分界判别函数为： y =
-2.686 x + 0.2167
```

（2）带惩罚项

生成的 0/1 数据比例为 1:1，生成 200 个测试用例，惩罚项系数为 0.001 迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5





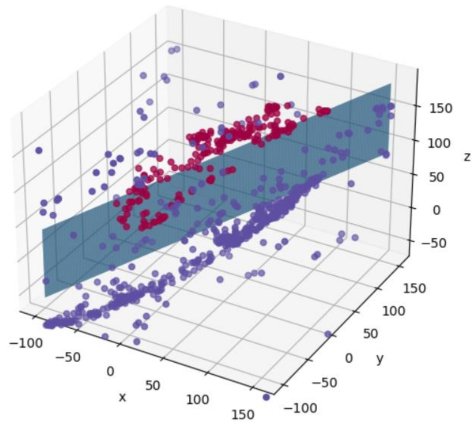

```
迭代次数= 1171
对应的损失函数值= [0.14838599]
对应的系数w= [[0.24057152 4.56280514 2.17736684]]
对应的梯度= [[-0.00084443 -0.009291    -0.00358115]]
分界判别函数为： y =
-2.096 x - 0.1105
```

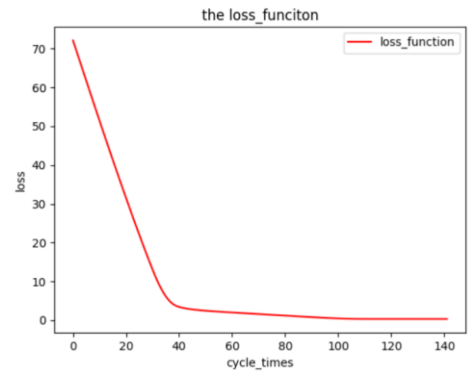
通过对图像进行分析可以得到：若数据不满足朴素贝叶斯假设，损失函数值会变大，即准确率会降低。

3.3 使用 UCI 上的 Skin_NonSkin.csv 进行测试（三维）

（1）不带惩罚项

迭代最多次数为 1000000，初始步长为 0.001，迭代误差为 1e-5

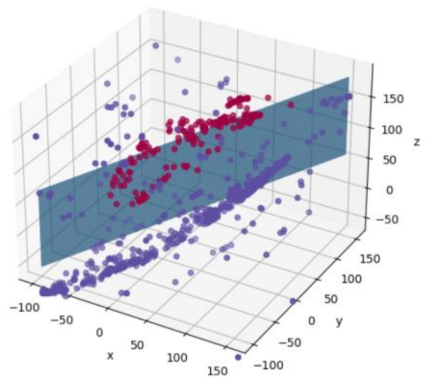


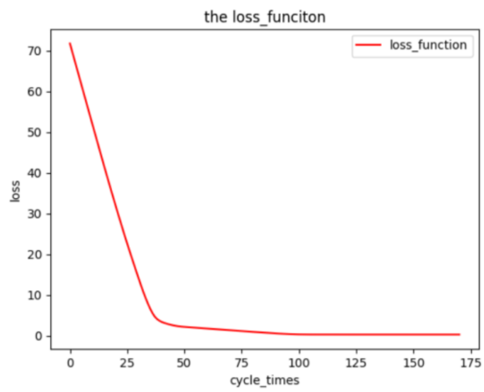



```
迭代次数= 141
对应的损失函数值= [0.32482903]
对应的系数w= [[ 1.01334817  0.01509077  0.0104482  -0.02648057]]
对应的梯度= [[-0.08592742 -0.02920346  0.03898056 -0.01128975]]
数据的测试集的准确率为： 0.9405763835756185
```

（2）带惩罚项

惩罚项系数为 0.01，迭代最多次数为 1000000，初始步长为 0.001，迭代误差为 1e-5



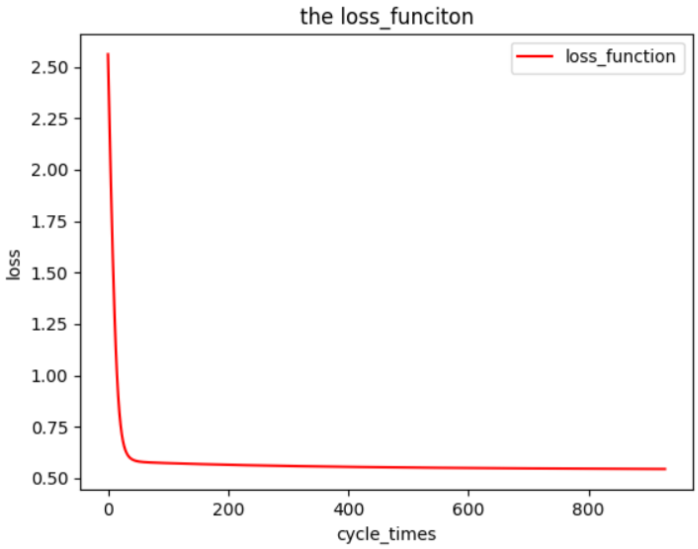



```
迭代次数= 170
对应的损失函数值= [0.31597635]
对应的系数w= [[ 1.0170184   0.01180051  0.01265125 -0.02564327]]
对应的梯度= [[-0.09959118 -0.01868434  0.02554739 -0.00766577]]
数据的测试集的准确率为： 0.9474623820453966
```

3.4 使用 UCI 上的 blood.csv 进行测试

（1）不带惩罚项

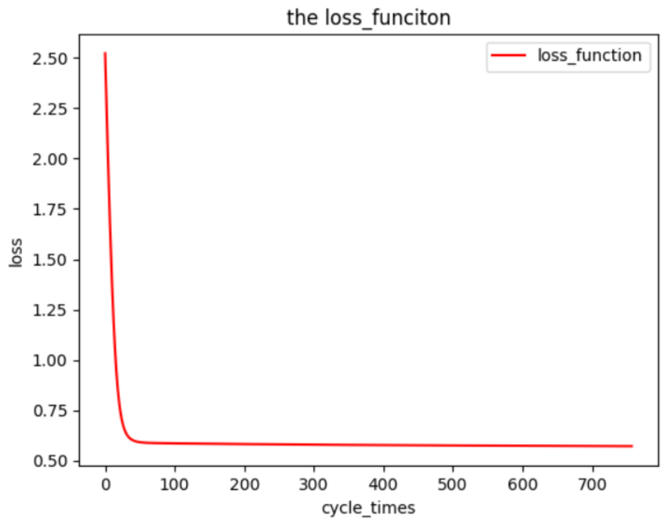
迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5




```
迭代次数= 927
对应的损失函数值= [0.54486954]
对应的系数w= [[-1.09677922  1.25319852 -0.79845115 -0.79845115  0.72541166]]
对应的梯度= [[ 0.00423285 -0.00617134  0.00259475  0.00259475 -0.00552493]]
数据的测试集的准确率为： 0.7712854757929883
```

（2）带惩罚项

惩罚项系数为 0.01，迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5

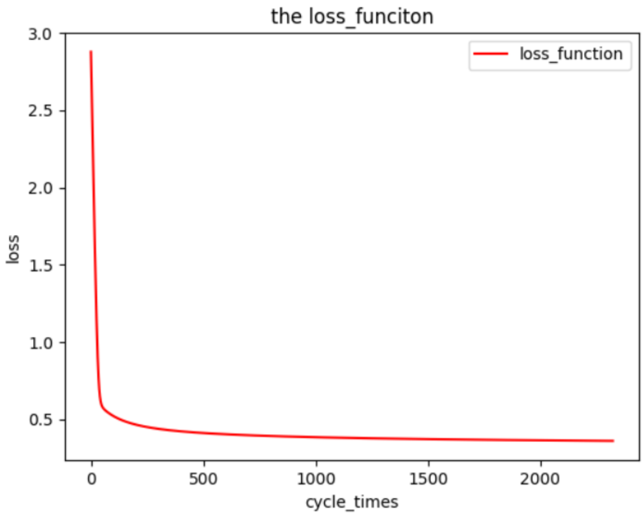



```
迭代次数= 757
对应的损失函数值= [0.57288786]
对应的系数w= [[-0.61810035  0.21807455 -0.58717378 -0.58717378  0.69180802]]
对应的梯度= [[ 0.00633733 -0.00328306  0.00828441  0.00828441 -0.01165463]]
数据的测试集的准确率为： 0.7729549248747913
```

3.5 使用 UCI 上的 heart.csv 进行测试

（1）不带惩罚项

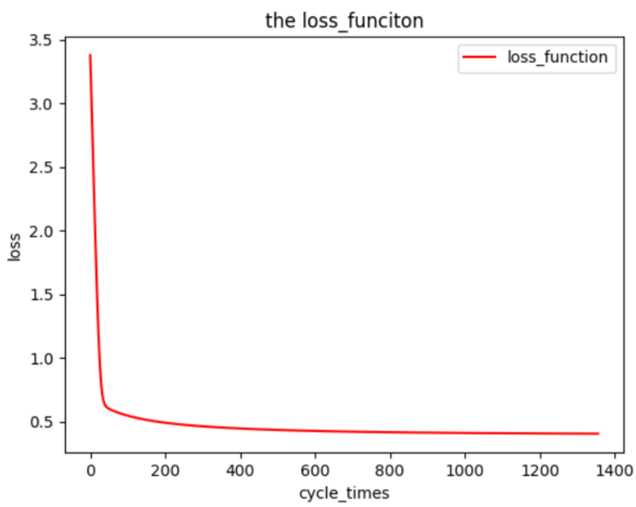
迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5




```
迭代次数= 1314
对应的损失函数值= [0.44041108]
对应的系数w= [[-1.68176465 -0.2245306   1.3685102  -1.61514898  0.47559623  0.50113866
  -0.45590281  0.28699917 -0.91086881  1.53812834  1.68398449 -1.85313309
   0.93059536  1.50573261]]
对应的梯度= [[ 2.86646102e-03  2.77096224e-03 -2.39526190e-03  1.19452680e-03
  -5.76075824e-04 -1.03283853e-03  9.51880422e-04 -7.74411200e-05
   4.22441495e-03 -1.78351191e-03 -3.77069185e-03  3.14637807e-03
  -3.18842514e-03 -4.39388054e-03]]
数据的测试集的准确率为：  0.46502057613168724
```

（2）带惩罚项

惩罚项系数为 0.01，迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5

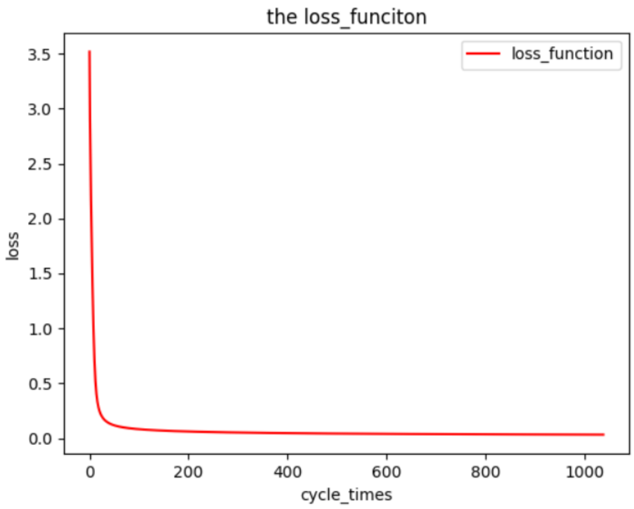


```
迭代次数= 1183
对应的损失函数值= [0.37935122]
对应的系数w= [[-0.5621293   0.16646794  2.03304325 -1.26233591  0.08569998  0.38130998
  -0.12582791  0.25856945 -0.23843592  0.83161769  0.73807181 -1.70935519
  -0.21277727  0.76346485]]
对应的梯度= [[ 0.00477933 -0.00111087 -0.02119195  0.01420152 -0.0006597  -0.00532825
   0.00049549 -0.00353744  0.00288388 -0.00803394 -0.00722893  0.01998897
   0.00296622 -0.00679806]]
数据的测试集的准确率为： 0.5555555555555556
```

3.6 使用 UCI 上的 data_banknote_authentication.csv 进行测试

（1）不带惩罚项

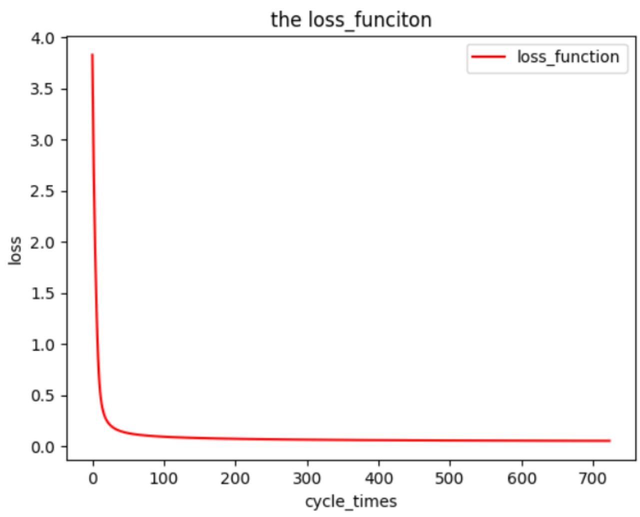
迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5




```
迭代次数= 1039
对应的损失函数值= [0.03317917]
对应的系数w= [[ 2.50203025 -2.07754665 -1.22787465 -1.46339878 -0.03038967]]
对应的梯度= [[-0.00752398  0.00492056  0.00253372  0.00345335 -0.00086916]]
数据的测试集的准确率为： 0.9899817850637522
```

（2）带惩罚项

惩罚项系数为 0.01，迭代最多次数为 1000000，初始步长为 0.1，迭代误差为 1e-5




```
迭代次数= 723
对应的损失函数值= [0.05508854]
对应的系数w= [[ 1.73287967 -1.47037375 -0.9073846  -1.04489277 -0.08294074]]
对应的梯度= [[-0.02050091  0.0157774   0.00974001  0.01143956  0.00039289]]
数据的测试集的准确率为： 0.9908925318761385
```

4 结论

1. 在数据量特别大时，使用梯度下降法，使用惩罚项对准确率提升影响很小，几乎可以忽略。可能原因是过拟合现象几乎可以忽略。

2. 数据量小时使用惩罚项可以提高准确率。

3. 类条件分布如果满足朴素贝叶斯假设时比不满足的情况准确率高。

4. 数据集的数据分布会影响数据测试集的准确率，blood.csv和heart.csv的准确率不是很高，而Skin_NonSkin.csv和data_banknote_authentication.csv的准确率很高，一方面的原因可能来自数据集的分布规律，另外一个原因可能是因为blood.csv和heart.csv的数据太大，为了防止exp溢出进行了归一化。

5. 逻辑回归可以解决简单的数据集分类问题，梯度下降法的迭代次数也不是很大。

5 源代码（含详细注释）

Lab2_1180300829.py

```
from Lab2.Skin_NonSkin import Skin_NonSkin_experiment
from Lab2.blood import blood_exp
from Lab2.data_banknote_authentication import
data_banknote_authentication_exp
from Lab2.design_experiment import design_experiment
```

```
from Lab2.heart import heart_exp

cycle_times = 1000000
descending_step_size = 0.1
iteration_error = 1e-5
design_experiment(200, 0, True, cycle_times, descending_step_size,
iteration_error)
design_experiment(200, 0.001, True, cycle_times,
descending_step_size, iteration_error)
design_experiment(200, 0, False, cycle_times, descending_step_size,
iteration_error)
design_experiment(200, 0.001, False, cycle_times,
descending_step_size, iteration_error)

cycle_times1 = 1000000
descending_step_size1 = 0.001
iteration_error1 = 1e-5
Skin_NonSkin_experiment(0, cycle_times1, descending_step_size1,
iteration_error1)
Skin_NonSkin_experiment(0.01, cycle_times1, descending_step_size1,
iteration_error1)

cycle_times3 = 1000000
descending_step_size3 = 0.1
iteration_error3 = 1e-5
blood_exp(0, cycle_times3, descending_step_size3, iteration_error3)
blood_exp(0.01, cycle_times3, descending_step_size3,
iteration_error3)



cycle_times4 = 1000000
descending_step_size4 = 0.1
iteration_error4 = 1e-5
heart_exp(0, cycle_times4, descending_step_size4, iteration_error4)
heart_exp(0.01, cycle_times4, descending_step_size4,
iteration_error4)

cycle_times2 = 1000000
descending_step_size2 = 0.1
iteration_error2 = 1e-5
data_banknote_authentication_exp(0, cycle_times2,
descending_step_size2, iteration_error2)
data_banknote_authentication_exp(0.01, cycle_times2,
descending_step_size2, iteration_error2)
```

blood.py

```
import numpy as np

from Lab2.descent_gradient_add_errorfunction import descent_gradient_add_errorfunction
from Lab2.draw import draw_picture_loss

'''
读入 blood.csv 文件，对数据进行拆分，拆分成训练集合测试集
'''
def blood_getdata():
    all_data = np.loadtxt(open('./blood.csv'), delimiter=",", skiprows=0)  # 读取文件中的所有数据
    np.random.shuffle(all_data)  # 将原数据集打乱，分成训练集和测试集
    test_rate = 0.2  # 测试集所占比例
    all_data_size = np.size(all_data, axis=0)  # 总数据集数据数量
    train_data_X = all_data[:int(test_rate * all_data_size), :]  # 训练集数据
    test_data_x = all_data[int(test_rate * all_data_size):, :]  # 测试集数据
    dimension = np.size(all_data, axis=1) - 1  # 数据集样本维度
    # 消除 exp 溢出，防止数据太大导致 exp 溢出
    for i in range(dimension):  # 对于样本的所有维度
        d_length = max(train_data_X[:, i]) - min(train_data_X[:, i])  # 计算最大值和最小值之间的极差
        for j in range(np.size(train_data_X, axis=0)):  # 对于每一维度的所有数
            train_data_X[j, i] = (max(train_data_X[:, i]) - train_data_X[j, i]) / d_length  # 将其化为[0,1]之间的值，防止 exp 溢出
    train_point_X = train_data_X[:, 0:dimension]  # 将所有数据集赋给 train_point_X
    train_classification_Y = train_data_X[:, dimension:dimension + 1]  # 为 train_classification_Y 赋值为 0/1
    train_size = np.size(train_point_X, axis=0)  # 训练集数据总数
    train_classification_Y = train_classification_Y.reshape(train_size)  # 将矩阵转化为行向量
    test_point_X = test_data_x[:, 0:dimension]  # 将所有数据集赋给 test_point_X
    test_classification_Y = test_data_x[:, dimension:dimension + 1]  # 为 test_classification_Y 赋值为 0/1
    test_size = np.size(test_point_X, axis=0)  # 测试集数据总数
    test_classification_Y = test_classification_Y.reshape(test_size)  # 将矩阵转化为行向量
    return train_point_X, train_classification_Y, test_point_X,
```

```
test_classification_Y



'''
使用 blood.csv 上的数据进行试验
参数中 lamda 为惩罚项系数，cycle_times 为梯度下降迭代最大次
数,descending_step_size 为梯度下降下降步长,iteration_error 为梯度下降迭代误
差
'''
def blood_exp(lamda, cycle_times, descending_step_size,
iteration_error):
    train_point_X, train_classification_Y, test_point_X,
test_classification_Y = blood_getdata()
    train_size = np.size(train_point_X, axis=0)  # 训练集样本数量
    test_size = np.size(test_point_X, axis=0)  # 测试集样本数量
    dimension = np.size(train_point_X, axis=1)  # 样本维度
    # 构造训练集样本矩阵
    train_all = np.ones((train_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 train_all
    for i in range(dimension):  # 依次将训练集样本的每一个维度放入 train_all
的下一个维度
        train_all[:, i + 1] = train_point_X[:, i]
    w, cycle_times_list, loss_list =
descent_gradient_add_errorfunction(train_all, train_classification_Y,
cycle_times,

descending_step_size, iteration_error,

dimension, lamda)
    w = w.reshape(-1)  # 得到的 w 是一个一行 dimension + 1 列的矩阵,需要先将 w
改成行向量
    function_coefficient = - (w / w[dimension])[0:dimension]  # w 整体
除 y 的系数然后移项得到决策面系数
    draw_picture_loss(cycle_times_list, loss_list)
    # 计算测试集准确率
    label = np.ones(test_size)
    hit_count = 0
    test_all = np.ones((test_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 test_all
    for i in range(dimension):  # 依次将测试集样本的每一个维度放入 train_all
的下一个维度
        test_all[:, i + 1] = test_point_X[:, i]
    for i in range(test_size):  # 对每种预测给 label 进行赋值
        if np.dot(w, test_all[i].T) >= 0:
```

```
            label[i] = 1
        else:
            label[i] = 0
    for i in range(test_size):
        if label[i] == test_classification_Y[i]:  # 如果预测的结果与真实结果相同计数加一
            hit_count += 1
    hit_rate = hit_count / test_size
    print('数据的测试集的准确率为：', hit_rate)
```

data_banknote_authentication.py

```
import numpy as np

from Lab2.descent_gradient_add_errorfunction import descent_gradient_add_errorfunction
from Lab2.draw import draw_picture_loss

'''
读入 data_banknote_authentication.csv 文件，对数据进行拆分，拆分成训练集合测试集
'''
def data_banknote_authentication_getdata():
    all_data = np.loadtxt(open('./data_banknote_authentication.csv'), delimiter=",", skiprows=0)  # 读取文件中的所有数据
    np.random.shuffle(all_data)  # 将原数据集打乱，分成训练集和测试集
    test_rate = 0.2  # 测试集所占比例
    all_data_size = np.size(all_data, axis=0)  # 总数据集数据数量
    train_data_X = all_data[:int(test_rate * all_data_size), :]  # 训练集数据
    test_data_x = all_data[int(test_rate * all_data_size):, :]  # 测试集数据
    dimension = np.size(all_data, axis=1) - 1  # 训练集样本维度
    train_point_X = train_data_X[:, 0:dimension]  # 将所有数据集赋给 train_point_X
    train_classification_Y = train_data_X[:, dimension:dimension + 1]  # 为 train_classification_Y 赋值为 0/1
    train_size = np.size(train_point_X, axis=0)  # 训练集数据总数
    train_classification_Y = train_classification_Y.reshape(train_size)  # 将矩阵转化为行向量
    test_point_X = test_data_x[:, 0:dimension]  # 将所有数据集赋给 test_point_X
    test_classification_Y = test_data_x[:, dimension:dimension + 1]  # 为 test_classification_Y 赋值为 0/1
    test_size = np.size(test_point_X, axis=0)  # 测试集数据总数
    test_classification_Y = test_classification_Y.reshape(test_size)
```

```
# 将矩阵转化为行向量
    return train_point_X, train_classification_Y, test_point_X,
test_classification_Y



'''
使用 data_banknote_authentication.csv 上的数据进行试验
参数中 lamda 为惩罚项系数，cycle_times 为梯度下降迭代最大次
数,descending_step_size 为梯度下降下降步长,iteration_error 为梯度下降迭代误
差
'''
def data_banknote_authentication_exp(lamda, cycle_times,
descending_step_size, iteration_error):
    train_point_X, train_classification_Y, test_point_X,
test_classification_Y = data_banknote_authentication_getdata()
    train_size = np.size(train_point_X, axis=0)  # 训练集样本数量
    test_size = np.size(test_point_X, axis=0)  # 测试集样本数量
    dimension = np.size(train_point_X, axis=1)  # 样本维度
    # 构造训练集样本矩阵
    train_all = np.ones((train_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 train_all
    for i in range(dimension):  # 依次将训练集样本的每一个维度放入 train_all
的下一个维度
        train_all[:, i + 1] = train_point_X[:, i]
    w, cycle_times_list, loss_list =
descent_gradient_add_errorfunction(train_all, train_classification_Y,
cycle_times,

descending_step_size, iteration_error,

dimension, lamda)
    w = w.reshape(-1)  # 得到的 w 是一个一行 dimension + 1 列的矩阵,需要先将 w
改成行向量
    function_coefficient = - (w / w[dimension])[0:dimension]  # w 整体
除 y 的系数然后移项得到决策面系数
    draw_picture_loss(cycle_times_list, loss_list)
    # 计算测试集准确率
    label = np.ones(test_size)
    hit_count = 0
    test_all = np.ones((test_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 test_all
    for i in range(dimension):  # 依次将测试集样本的每一个维度放入 train_all
的下一个维度
        test_all[:, i + 1] = test_point_X[:, i]
```

```
    for i in range(test_size):  # 对每种预测给 label 进行赋值
        if np.dot(w, test_all[i].T) >= 0:
            label[i] = 1
        else:
            label[i] = 0
    for i in range(test_size):
        if label[i] == test_classification_Y[i]:  # 如果预测的结果与真实结
果相同计数加一
            hit_count += 1
    hit_rate = hit_count / test_size
    print('数据的测试集的准确率为：', hit_rate)
```

descent_gradient_add_errorfunction.py

```
import numpy as np

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


'''
自己生成数据
根据是否满足贝叶斯假设，在范围[0,1]上生成 sample_number 个二维点集，并将其均匀分
为两类
并添加一个 N(0,1)的高斯噪声
'''
def create_datas(sample_number, satisfy_naive):
    half = np.ceil(sample_number / 2).astype(np.int32)  #
sample_number 的一半
    variance = 0.2  # 随机变量方差
    covariance_xy = 0.01  # 两个维度的协方差
    point_mean1 = [-0.7, -0.3]  # 类别 1 的均值
    point_mean2 = [0.7, 0.3]  # 类别 2 的均值
    train_point_X = np.zeros((sample_number, 2))  # 二维的点集
    classification_Y = np.zeros(sample_number)  # 点集的分类
    if satisfy_naive:  # 满足朴素贝叶斯假设
        train_point_X[:half, :] =
np.random.multivariate_normal(point_mean1, [[variance, 0], [0,
variance]],
                                                   size=half)  # 生
成 half 个类别 1 的 1*2 数组，每个数组含有两个维度
        train_point_X[half:, :] =
np.random.multivariate_normal(point_mean2, [[variance, 0], [0,
```

```
variance]],

size=sample_number - half)  # 生成 half 个类别 2 的 1*2 数组，每个数组含有两个维度
        classification_Y[:half] = 0  # 将前 half 个数组标记为类别 1
        classification_Y[half:] = 1  # 将后 half 个数组标记为类别 2
    else:  # 不满足朴素贝叶斯假设
        train_point_X[:half, :] =
np.random.multivariate_normal(point_mean1,
                                                                [[variance,
covariance_xy], [covariance_xy, variance]],
                                                                size=half)  # 生成 half 个类别 1 的 1*2 数组，每个数组含有两个维度
        train_point_X[half:, :] =
np.random.multivariate_normal(point_mean2,
                                                                [[variance,
covariance_xy], [covariance_xy, variance]],

size=sample_number - half)  # 生成 half 个类别 2 的 1*2 数组，每个数组含有两个维度
        classification_Y[:half] = 0  # 将前 half 个数组标记为类别 1
        classification_Y[half:] = 1  # 将后 half 个数组标记为类别 2
    return train_point_X, classification_Y  # 返回生成的所有点及类别



'''
根据公式得到点集的极大条件似然得到极大条件似然 l(W)
'''
def maximum_conditional_likelihood(train_point_X, classification_Y,
w):
    total_points = np.size(train_point_X, axis=0)  # 得到 train_point_X 的行数，即点集个数
    predict = np.zeros((total_points, 1))
    for i in range(total_points):
        predict[i] = w.dot(train_point_X[i].T)  # 极大条件似然中的 exp 中的部分，即求和 wi * Xi
    t = 0
    for i in range(total_points):
        t += np.log(1 + np.exp(predict[i]))  # 极大条件似然中 ln 的部分，即 ln ( 1 + exp ( 求和 wi * Xi ) )
    MCLE = classification_Y.dot(predict) - t  # 得到极大条件似然 l(w)
    return MCLE
```

```
'''
加惩罚项的梯度下降法
对于 train_point_X 和 classification_Y 对参数 w 做梯度下降，对损失函数使用梯度下降法，当误差函数收敛到期望的最小值时，得到此时的 w 并返回 w
参数中：
迭代最大次数为 cycle_times
下降步长 descending_step_size
迭代误差 iteration_error
数据点集维度 dimension
惩罚项参数 lamda
'''
def descent_gradient_add_errorfunction(train_point_X, classification_Y, cycle_times, descending_step_size,
                                       iteration_error, dimension, lamda):
    total_points = np.size(train_point_X, axis=0)  # 得到 train_point_X 的行数，即点集个数
    w = np.ones((1, dimension + 1))  # 生成系数矩阵 w，一个列数为 dimension + 1，行数为 1 的矩阵，元素值全为 1
    cycle_times_list = np.zeros(cycle_times)  # 迭代次数统计
    loss_list = np.zeros(cycle_times)  # 迭代次数对应的损失函数值统计
    for i in range(cycle_times):
        old_loss = - 1 / total_points * maximum_conditional_likelihood(train_point_X, classification_Y, w)  # 原先损失函数的值
        t = np.zeros((total_points, 1))
        for j in range(total_points):
            t[j] = w.dot(train_point_X[j].T)  # 极大条件似然中的 exp 中的部分，即求和 wi * Xl
        gradient = - 1 / total_points * (classification_Y - sigmoid(t.T)).dot(train_point_X)
        w = w - descending_step_size * lamda * w - descending_step_size * gradient  # 梯度下降加惩罚项的迭代公式
        new_loss = - 1 / total_points * maximum_conditional_likelihood(train_point_X, classification_Y, w)  # 新的损失函数的值
        cycle_times_list[i] = i  # 储存迭代次数
        loss_list[i] = new_loss  # 储存每次迭代对应的损失函数值
        if abs(new_loss - old_loss) < iteration_error:  # 如果新的误差函数值与旧的误差函数值的差小于迭代误差则终止迭代
            cycle_times_list = cycle_times_list[:i + 1]
            loss_list = loss_list[:i + 1]
            print('迭代次数=', i, '\n 对应的损失函数值=', new_loss, '\n 对应的系数 w=', w, '\n 对应的梯度=', gradient)
```

```
            break
        if new_loss > old_loss:  # 当新的误差函数值大于旧的误差函数值时，将步
长变为原来的一半
            descending_step_size *= 0.5
    return w, cycle_times_list, loss_list
```

design_experiment.py

```
import numpy as np

from Lab2.descent_gradient_add_errorfunction import create_datas,
descent_gradient_add_errorfunction
from Lab2.draw import draw_picture, draw_picture_loss

'''
自定义二维点集进行试验并绘制图像
参数中：sample_number 为点集中点的个数，lamda 为惩罚项系数（可以为 0，此时没有惩
罚项），satisfy_naive 为是否满足朴素贝叶斯假设
cycle_times 为梯度下降迭代最大次数,descending_step_size 为梯度下降下降步
长,iteration_error 为梯度下降迭代误差
'''
def design_experiment(sample_number, lamda, satisfy_naive,
cycle_times, descending_step_size, iteration_error):
    train_point_X, classification_Y = create_datas(sample_number,
satisfy_naive)  # 生成 sample_number 个二维点集数据
    train_all = np.ones((sample_number, 3))  # 创建行为 sample_number,
列为 3 的矩阵 train_all
    train_all[:, 1] = train_point_X[:, 0]  # 将生成点集的第一个维度放入
train_all 的第二列
    train_all[:, 2] = train_point_X[:, 1]  # 将生成点集的第二个维度放入
train_all 的第三列
    dimension = np.size(train_point_X, axis=1)
    w, cycle_times_list, loss_list =
descent_gradient_add_errorfunction(train_all, classification_Y,
cycle_times,

descending_step_size, iteration_error,

dimension, lamda)
    w = w.reshape(-1)  # 得到的 w 是一个一行三列的矩阵,需要先将 w 改成行向量
    function_coefficient = -(w / w[2])[0:2]  # w 整体除 y 的系数然后移项得
到决策面系数
    boundary_check_function = np.poly1d(function_coefficient[::-1])  #
将 function_coefficient 从后往前倒序然后调用 poly1d 得到多项式函数
    print('分界判别函数为: y = ', boundary_check_function)
    draw_picture(train_point_X, classification_Y,
```

```
boundary_check_function)
    draw_picture_loss(cycle_times_list, loss_list)
```

draw.py

```
import numpy as np
import matplotlib.pyplot as plt

'''
对于 train_point_X 中，含有二维数据，分别对应横坐标 x 和纵坐标 y
对于 classification_Y,为 train_point_X 中两种数据的点对应的类别
根据 train_point_X 和 classification_Y 画出二维坐标下的点图，然后画出分界判别函
数 boundary_check_function
'''
def draw_picture(train_point_X, classification_Y,
boundary_check_function):
    if boundary_check_function:  # 绘制分界判定函数
boundary_check_function
        d_length = max(train_point_X[:, 0]) - min(train_point_X[:, 0])
# 找到横坐标 x 的极差
        real_x = np.linspace(min(train_point_X[:, 0]),
min(train_point_X[:, 0]) + d_length, 50)  # 在 x 的范围内均匀产生 50 个点
        real_y = boundary_check_function(real_x)  # 对 real_x 每个点调用
boundary_check_function 求解对应的 real_y
        plt.plot(real_x, real_y, 'r', label='boundary_check_function')
# 绘制图像
    plot = plt.scatter(train_point_X[:, 0], train_point_X[:, 1],
s=30, c=classification_Y, marker='o',
                      cmap=plt.cm.Spectral)  # 绘制两种点集
    plt.xlabel('x')
    plt.ylabel('y')
    plt.legend(loc=1)
    plt.title('The regression curve')
    plt.show()


'''
根据迭代次数 cycle_times_list 和对应的误差 loss_list 画出损失函数图像
参数中，cycle_times_list 为迭代次数表，loss_list 为对应的误差表
'''
def draw_picture_loss(cycle_times_list, loss_list):
    plt.plot(cycle_times_list, loss_list, 'r', label='loss_function')
    plt.xlabel('cycle_times')
    plt.ylabel('loss')
    plt.legend(loc=1)
```

```
    plt.title('the loss_funciton')
    plt.show()
```

heart.py

```
import numpy as np

from Lab2.descent_gradient_add_errorfunction import
descent_gradient_add_errorfunction
from Lab2.draw import draw_picture_loss

'''
读入 heart.csv 文件，对数据进行拆分，拆分成训练集合测试集
'''
def heart_getdata():
    all_data = np.loadtxt(open('./heart.csv'), delimiter=",",
skiprows=0)  # 读取文件中的所有数据
    np.random.shuffle(all_data)  # 将原数据集打乱，分成训练集和测试集
    test_rate = 0.2  # 测试集所占比例
    all_data_size = np.size(all_data, axis=0)  # 总数据集数据数量
    train_data_X = all_data[:int(test_rate * all_data_size), :]  # 训练
集数据
    test_data_x = all_data[int(test_rate * all_data_size):, :]  # 测试
集数据
    dimension = np.size(all_data, axis=1) - 1  # 数据集样本维度
    # 消除 exp 溢出，防止数据太大导致 exp 溢出
    for i in range(dimension):  # 对于样本的所有维度
        d_length = max(train_data_X[:, i]) - min(train_data_X[:, i])
# 计算最大值和最小值之间的极差
        for j in range(np.size(train_data_X, axis=0)):  # 对于每一维度的
所有数
            train_data_X[j, i] = (max(train_data_X[:, i]) -
train_data_X[j, i]) / d_length  # 将其化为[0,1]之间的值，防止 exp 溢出
    train_point_X = train_data_X[:, 0:dimension]  # 将所有数据集赋给
train_point_X
    train_classification_Y = train_data_X[:, dimension:dimension + 1]
# 为 train_classification_Y 赋值为 0/1
    train_size = np.size(train_point_X, axis=0)  # 训练集数据总数
    train_classification_Y =
train_classification_Y.reshape(train_size)  # 将矩阵转化为行向量
    test_point_X = test_data_x[:, 0:dimension]  # 将所有数据集赋给
test_point_X
    test_classification_Y = test_data_x[:, dimension:dimension + 1]  #
为 test_classification_Y 赋值为 0/1
    test_size = np.size(test_point_X, axis=0)  # 测试集数据总数
    test_classification_Y = test_classification_Y.reshape(test_size)
```

```
# 将矩阵转化为行向量
    return train_point_X, train_classification_Y, test_point_X,
test_classification_Y



'''
使用 heart.csv 上的数据进行试验
参数中 lamda 为惩罚项系数，cycle_times 为梯度下降迭代最大次
数,descending_step_size 为梯度下降下降步长,iteration_error 为梯度下降迭代误
差
'''
def heart_exp(lamda, cycle_times, descending_step_size,
iteration_error):
    train_point_X, train_classification_Y, test_point_X,
test_classification_Y = heart_getdata()
    train_size = np.size(train_point_X, axis=0)  # 训练集样本数量
    test_size = np.size(test_point_X, axis=0)  # 测试集样本数量
    dimension = np.size(train_point_X, axis=1)  # 样本维度
    # 构造训练集样本矩阵
    train_all = np.ones((train_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 train_all
    for i in range(dimension):  # 依次将训练集样本的每一个维度放入 train_all
的下一个维度
        train_all[:, i + 1] = train_point_X[:, i]
    w, cycle_times_list, loss_list =
descent_gradient_add_errorfunction(train_all, train_classification_Y,
cycle_times,

descending_step_size, iteration_error,

dimension, lamda)
    w = w.reshape(-1)  # 得到的 w 是一个一行 dimension + 1 列的矩阵,需要先将 w
改成行向量
    function_coefficient = - (w / w[dimension])[0:dimension]  # w 整体
除 y 的系数然后移项得到决策面系数
    draw_picture_loss(cycle_times_list, loss_list)
    # 计算测试集准确率
    label = np.ones(test_size)
    hit_count = 0
    test_all = np.ones((test_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 test_all
    for i in range(dimension):  # 依次将测试集样本的每一个维度放入 train_all
的下一个维度
        test_all[:, i + 1] = test_point_X[:, i]
```

```
    for i in range(test_size):  # 对每种预测给 label 进行赋值
        if np.dot(w, test_all[i].T) >= 0:
            label[i] = 1
        else:
            label[i] = 0
    for i in range(test_size):
        if label[i] == test_classification_Y[i]:  # 如果预测的结果与真实结果相同计数加一
            hit_count += 1
    hit_rate = hit_count / test_size
    print('数据的测试集的准确率为：', hit_rate)
```

Skin_NonSkin.py

```
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from Lab2.descent_gradient_add_errorfunction import descent_gradient_add_errorfunction
from Lab2.draw import draw_picture_loss

'''
Skin_NonSkin 数据集是三个维度的，所以可以画出三维的图像。
'''
def Skin_NonSkin_draw_picture(train_point_X, classification_Y, function_coefficient):
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.set_title('3D of the regression curve')
    ax.scatter(train_point_X[:, 0], train_point_X[:, 1], train_point_X[:, 2], c=classification_Y, cmap=plt.cm.Spectral)
    real_x = np.arange(np.min(train_point_X[:, 0]), np.max(train_point_X[:, 0]), 1)
    real_y = np.arange(np.min(train_point_X[:, 1]), np.max(train_point_X[:, 1]), 1)
    real_X, real_Y = np.meshgrid(real_x, real_y)
    real_z = function_coefficient[0] + function_coefficient[1] * real_X + function_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.set_xlabel('x')
    ax.set_ylabel('y')
    ax.set_zlabel('z')
    plt.show()
```

```
'''
读入 Skin_NonSkin.csv 文件，对数据进行拆分，拆分成训练集合测试集
由于原文件中数据量巨大，所以对数据集以 50 步长取部分数据作为数据集
'''
def Skin_NonSkin_getdata():
    all_data = np.loadtxt(open('./Skin_NonSkin.csv'), delimiter=",", skiprows=0)  # 读取文件中的所有数据
    np.random.shuffle(all_data)  # 将原数据集打乱，分成训练集和测试集
    test_rate = 0.2  # 测试集所占比例
    all_data_size = np.size(all_data, axis=0)  # 总数据集数据数量
    train_data_X = all_data[:int(test_rate * all_data_size), :]  # 训练集数据
    test_data_x = all_data[int(test_rate * all_data_size):, :]  # 测试集数据
    dimension = np.size(all_data, axis=1) - 1  # 训练集样本维度
    step = 50  # 由于 Skin_NonSkin 的数据集太大，所以采用步长为 50 的方式取数据
    train_point_X = train_data_X[:, 0:dimension]  # 将所有数据集赋给 train_point_X
    train_point_X = train_point_X[::step]  # 以 step 为间隔取数据
    train_point_X = train_point_X - 100  # 对样本点进行坐标平移，方便在 3D 图中显示
    train_classification_Y = train_data_X[:, dimension:dimension + 1] - 1  # 因为数据集的分类是 1/2,需要减 1 变成 0/1
    train_classification_Y = train_classification_Y[::step]  # 以 step 为间隔取数据
    train_size = np.size(train_point_X, axis=0)  # 训练集数据总数
    train_classification_Y = train_classification_Y.reshape(train_size)  # 将矩阵转化为行向量
    test_point_X = test_data_x[:, 0:dimension]  # 将所有数据集赋给 test_point_X
    test_point_X = test_point_X[::step] - 100  # 对样本点进行坐标平移，方便在 3D 图中显示
    test_classification_Y = test_data_x[:, dimension:dimension + 1] - 1  # 因为数据集的分类是 1/2,需要减 1 变成 0/1
    test_classification_Y = test_classification_Y[::step]  # 以 step 为间隔取数据
    test_size = np.size(test_point_X, axis=0)  # 测试集数据总数
    test_classification_Y = test_classification_Y.reshape(test_size)  # 将矩阵转化为行向量
    return train_point_X, train_classification_Y, test_point_X, test_classification_Y
```

```
'''
使用 Skin_NonSkin.csv 上的数据进行试验
参数中 lamda 为惩罚项系数，cycle_times 为梯度下降迭代最大次
数,descending_step_size 为梯度下降下降步长,iteration_error 为梯度下降迭代误
差
'''
def Skin_NonSkin_experiment(lamda, cycle_times, descending_step_size,
iteration_error):
    train_point_X, train_classification_Y, test_point_X,
test_classification_Y = Skin_NonSkin_getdata()  # 得到
Skin_NonSkin.csv 上的训练集样本和测试集样本
    train_size = np.size(train_point_X, axis=0)  # 训练集样本数量
    test_size = np.size(test_point_X, axis=0)  # 测试集样本数量
    dimension = np.size(train_point_X, axis=1)  # 样本维度
    train_all = np.ones((train_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 train_all
    for i in range(dimension):  # 依次将训练集样本的每一个维度放入 train_all
的下一个维度
        train_all[:, i + 1] = train_point_X[:, i]
    w, cycle_times_list, loss_list =
descent_gradient_add_errorfunction(train_all, train_classification_Y,
cycle_times,

descending_step_size, iteration_error,

dimension, lamda)
    w = w.reshape(-1)  # 得到的 w 是一个一行 dimension + 1 列的矩阵,需要先将 w
改成行向量
    function_coefficient = - (w / w[dimension])[0:dimension]  # w 整体
除 y 的系数然后移项得到决策面系数
    Skin_NonSkin_draw_picture(train_point_X, train_classification_Y,
function_coefficient)
    draw_picture_loss(cycle_times_list, loss_list)
    # 计算测试集的准确率
    label = np.ones(test_size)
    hit_count = 0
    test_all = np.ones((test_size, dimension + 1))  # 创建行为
train_size，列为样本维度+1 的矩阵 test_all
    for i in range(dimension):  # 依次将测试集样本的每一个维度放入 train_all
的下一个维度
        test_all[:, i + 1] = test_point_X[:, i]
    for i in range(test_size):  # 对每种预测给 label 进行赋值
        if w.dot(test_all[i].T) >= 0:
            label[i] = 1
```

```
        else:
            label[i] = 0
    for i in range(test_size):
        if label[i] == test_classification_Y[i]:  # 如果预测的结果与真实结果相同计数加一
            hit_count += 1
    hit_rate = hit_count / test_size
    print('数据的测试集的准确率为：', hit_rate)
```