哈尔滨工业大学计算学部

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：**1190201215**
姓名： 冯开来

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（梯度下降法/共轭梯度法/牛顿法）。

二、实验要求及实验环境

2.1 实验要求

1. 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）
2. 采用梯度下降、共轭梯度或者牛顿法等。

2.2 实验环境

Windows10; python3.9;PyCharm 2021.2.2

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

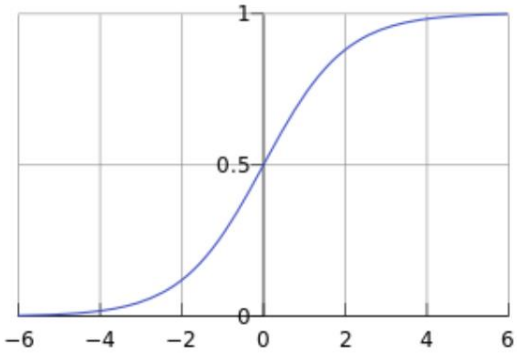
本次实验是通过建立逻辑回归模型，实现线性分类，从而进行分类(预测)任务。

1.1 逻辑回归模型原理

逻辑回归主要是针对二分类预测问题，通过直接计算后验概率来对类别进行判断，当计算结果大于 0.5 时，判别结果为 1，结果小于 0.5 则判别为 0；因此，逻辑回归实际是一种概率估计。

为了能将数据信息映射到 0 到 1 之间，我们需要寻找一种函数，这就是 sigmoid 函数，也就是判别函数：$g(z)=\frac{1}{1+\exp(-z)}$

该函数图像为：



当 z 为 0 时，函数值为 0.5；当 z 趋近正无穷时，函数值逼近 1；当 z 趋近负无穷时，函数值逼近 0；

该函数的输入便是参数 z，为了能够将多维数据映射到概率值上，做出线性分类器，我们将每一维的特征乘以一个系数并且累加，作为输入的参数，即：

$$z = w_0x_0 + w_1x_1 + w_2x_2 + \cdots w_nx_n = W^{\mathrm{T}}X$$

其中ω_0是作为一种截距/偏置，因此始终x_0为 1；

因此，只要通过训练样本对模型进行训练，计算出向量参数 W 的估计值，就可以利用判别函数来对数据进行分类。

逻辑回归中我们有这样几个假设：各维特征独立且满足高斯分布(类条件分布服从高斯分布)、每个类别的协方差阵相等，即方差与标签无关。

实验中，我们将参数 z 前的负号拿进来，由于这只是 W 加一个负号就可以解决，因此我们有 $z = W^T X$；

在预测时，就是要计算 P(Y|X)的值，从而进行类别预测，即 sigmoid 函数为

$$h(W) = P(Y = 1 | X =< X_1, X_2, ..., X_n >) = \frac{1}{1 + \exp(\omega_0 + \sum_i \omega_i X_i)}$$

为了能够实现矩阵形式，我们有如下定义：

$$W = [\omega_0, \omega_1, ..., \omega_n]^T, \text{其中} n \text{表示特征维数}$$

$$X^l = [1 \ X_1^l \ X_2^l \ ... X_n^l]$$

$$X = [X^1 \ X^2 \ ... X^l]$$

$$Y = [y_1 \ y_2 \ ... y_l]^T$$

那么，有 $h(W) = P(Y = 1 | X^l) = \frac{1}{1 + \exp(W^T X^l)}$。

又因各维特征独立，由此我们建立函数为：

$$MCLE = \prod_l P(Y^l | X^l, W)$$

为了防止计算过程中发生溢出，我们对函数取对数：

$$\begin{aligned} L(W) &= In \prod_l P(Y^l | X^l, W) \\ &= \sum_l P(Y^l | X^l, W) \\ &= \sum_l (Y^l InP(Y^l = 1 | X^l, W) + (1 - Y^l) InP(Y^l = 0 | X^l, W)) \\ &= \sum_l (Y^l In \frac{P(Y^l = 1 | X^l, W)}{P(Y^l = 0 | X^l, W)} + InP(Y^l = 0 | X^l, W)) \end{aligned}$$

而

$$\cos t(h_W(x), y) = \begin{cases} -\log(h_W(x)) & y = 1 \\ -\log(1 - h_W(x)) & y = 0 \end{cases}$$

$$loss(w) = -\frac{1}{m}\sum_{l}(y^l \ln\left(\frac{1}{1+e^{W^T X^l}}\right) + (1-y^l)\ln\left(1-\frac{1}{1+e^{W^T X^l}}\right)$$

那么损失函数矩阵形式为：

$$loss(w) = -\frac{1}{m}(W^T XY + \sum_{l}\ln\left(1+e^{W^T X}\right)$$

$\frac{\partial loss(w)}{\partial w} = \frac{1}{m}X(Y - P(Y^l = 1|X^l, W)$ 归一化处理，不含正则项

$\frac{\partial loss(w)}{\partial w} = \frac{1}{m}X(Y - P(Y^l = 1|X^l, W) + \lambda W$ 含正则项

因此，只需要利用优化方法使得损失函数最小化，求得参数 W。

实验中，使用了三种方法进行求解：梯度下降法、共轭梯度法、牛顿法。

1.2 梯度下降法

不加正则项：

1）确定当前位置的损失函数梯度

2）表达式：

$$\frac{\partial loss(w)}{\partial w} = \frac{1}{m}X(Y - P(Y^l = 1|X^l, W)$$

3）用步长 α，乘以梯度，得到当前位置下降的距离

4）确定所有 w_i 梯度下降的距离都小于 ε，若满足，则算法终止

此时可以得到系数矩阵 W

加入正则项：

表达式变为：

$$\frac{\partial loss(w)}{\partial w} = \frac{1}{m}(X(Y - P(Y^l = 1|X^l, W) + \lambda W)$$

1.3 共轭梯度法

在实验一中，已经详细介绍了共轭梯度法。对于实验二来说，共轭梯度法的贺信在于考虑一个方程组 AX=B，其中矩阵 A 为正定对称阵，将方程组的求解转为求解一个二次泛函 $\phi(X) = \frac{1}{2}X^T AX - b^T X$ 的最小值问题，因为 $\frac{\partial \phi(X)}{\partial X} = AX - b$。

在实验中，首先需要构造这样的式子以及正定阵，我们发现由于类别标签只有两类，即 y 的值要么是 1 要么是 0，因此应该有对于任一个训练样本数据代入 $P(Y=1|X^l) = \frac{1}{1+\exp(W^T X^l)}$，若标签为 1，则值为 1，若标签为 0，则值为 0；

那么，当标签为 1 时，$\exp(W^T X)->0$；当标签为 0 时，$\exp(W^T X)->+\infty$；我们利用取近似值，即当标签为 1 时，$W^T X = In(10^{-20})$；当标签为 0 时，$W^T X = In(10^{20})$；

写成矩阵形式即为 $X^T W = b$，b 即是刚才近似思想下的近似值向量，两边同时左乘 X 矩阵；因此构造正定对称阵 $A = XX^T$，显然，A 是对称矩阵；而 $(X^T W)^T (X^T W) = W^T (XX^T) W > 0$,因此 A 是正定阵；

那么方程组就变成以下形式：

$$XX^T W = Xb$$

我们记 $B = Xb, A = X^T X$,那么方程组变为：

AW=B；

直观的来说，共轭梯度法是每次迭代在一个方向上进行一维搜索，寻找下一个使得函数值下降最快的方向，理论上迭代 N 次必能得到最优解；

(以下部分来自实验一)

在共轭梯度法中，我们要做两点改变，一是步长二是方向。因为梯度下降法是沿着一个方向不停下降，而共轭梯度是在众多方向中找到一个最快的下降方法。初始点的下降方向仍然是负梯度方，但后面的迭代方向是该点的负梯度方向和前一次的迭代方向（共轭方向）行程的凸锥中的一个方向（大概就是两个方向的线性组合），这样可以避免梯度下降的“锯齿”现象。

解释一下共轭方向，假设 d_0 和 d_1 关于 A 共轭，那么满足 $d_0{}^T A d_1 = 0$。同理，n 个方向共轭的话，即两两满足上式。

那么我们同样设步长为 α，方向向量为 d，替代原式中的梯度，得到：

$$w = w - \alpha d$$

如何求由 d_k 得到 d_{k+1}？因为 d_k 和 d_{k+1} 既满足共轭，也满足和负梯度方向是线性关系，所以我们可以得到两个等式：

$$d_{k+1} = -gradient\_w + \beta d_k$$

$$d_{k+1}{}^T A d_k = 0$$

所以可以解出：

$$\alpha = -\frac{(gradient\_w)^T d_k}{d_k^T A d_k}$$

$$\beta = \frac{d_k{}^T A (gradient\_w)}{d_k^T A d_k}$$

这样带入 $w = w - \alpha d$ 完成迭代，迭代次数也很有意思，因为每一步都是朝着

最陡的方向下降，所以迭代次数就是 w 的维数。实事证明，迭代这么点次数，确实收敛。

（以上部分来自实验一）

算法过程：

1) 任取$W \in R^N$(实验中初始化为全 0)，计算$r^{(0)} = b - AW^{(0)}$，取 $p^{(0)} = r^{(0)}$；

2) 对 k=1,2,3...，计算

$$\alpha_k = \frac{r^{(k)^T} r^{(k)}}{p^{(k)^T} A p^{(k)}}$$

$$W^{(k+1)} = W^{(k)} + \alpha_k p^{(k)}$$

$$r^{(k+1)} = r^{(k)} - \alpha_k A p^{(k)}$$

$$\beta_k = \frac{r^{(k+1)^T} r^{(k+1)}}{r^{(k)^T} r^{(k)}}$$

$$p^{(k+1)} = r^{(k+1)} + \beta_k p^{(k)}$$

其中， $p^{(0)}, p^{(1)}, ... p^{(k)}$是共轭向量组， $r^{(k)} = B - AW^{(k)}$是剩余向量；

3) 判断终止条件$r^k = 0$，成立则终止算法，不成立则重复循环步骤 2；

最后算法终止便可得到系数矩阵 W；

1.4 牛顿法

牛顿法解决的是 f(X)=0 一类问题，利用在近似值点 x_0 泰勒展开：

$$f(x) = f(x_0) + (x - x_0) f'(x_0) = 0$$

$$=> x = x_0 - \frac{f(x_0)}{f'(x_0)}$$

这是我们得到的迭代公式，可以对解做不断地逼近；

实验中我们在求得损失函数最小值后，有导数为 0，即 $\frac{\partial J(W)}{\partial W} = \frac{1}{m} X(Y - P) = 0$，这是不含正则项，含有正则项时，$\frac{\partial J(W)}{\partial W} = \frac{1}{m}(X(Y - P) + \lambda W) = 0$，因此选择这个导数作为 f 函数；而 f 函数的导数即是损失函数的海森矩阵 H。

我们去掉常系数 1/m，则 $f(W) = X(Y - P) = 0$，$H = f'(W) = X^T W X$，这是不含正则项；

当含有正则项时，$f(W)=X(Y-P)+\lambda W=0$，

$H=f'(W)=X^TWX+\lambda E_{(n+1)*(n+1)}$；

算法过程：
1）初始化近似值 W 为 0 向量
2）利用迭代公式不断计算 W 的值
3）代入 loss 函数计算 loss 值，并与之前的 loss 进行比较，若插值小于一个极小值，则迭代停止，否则重复 1，2 步骤
注意：由于导数是海森矩阵，是(n+1)*(n+1)维的矩阵，因此在矩阵除法中，需要对其取逆，再与 $f(W)$ 矩阵相乘。

1.5 三种方法比较

1. 梯度下降法是线性收敛，收敛速度慢，且步长过大会导致不收敛，步长过小则收敛速度太慢；
2. 牛顿法是平方收敛，收敛速度比梯度下降快了很多，但是由于需要计算海森矩阵、导数等，计算较为复杂，占用空间较大。
3. 共轭梯度法是介于梯度下降法和牛顿法之间的，克服前者收敛速度慢，后者占用空间大、计算复杂的缺点。

2.代码实现

2.1 生成数据

2.1.1 生成满足朴素贝叶斯假设的两个分类类别的数据

由于朴素贝叶斯假设中要求的是各维特征独立且满足高斯分布，因此在生成数据时，假设特征维数为 n，则需要进行 n 次高斯分布生成数据，作为相应维特征的值，然后进行拼接，得出前面交代的 X 矩阵的形式，相应的也可以构造矩阵 Y，从而实现了生成满足朴素贝叶斯假设的数据；

例如，实验中，我们生成特征维数为 2 的样本数据：

```
def generate_data(train_sample, navie = True):
    pos_mean = [1, 1.2] #正例的两维度均值
    neg_mean = [-1, -1.2] #反例的两维度均值
    X = np.zeros((2 * train_sample, 2))
    Y = np.zeros((2 * train_sample, 1))
    if navie:
        cov = np.mat([[0.3, 0], [0, 0.4]])
        X[:train_sample, :] = np.random.multivariate_normal(pos_mean, cov, train_sample)
        X[train_sample:, :] = np.random.multivariate_normal(neg_mean, cov, train_sample)
        Y[:train_sample] = 1
        Y[train_sample:] = 0
```

2.1.2 生成不满足朴素贝叶斯假设的数据

当不满足朴素贝叶斯假设时，即各维并不是独立的，而是相关的；可以证明的是，当类条件分布满足高斯分布时，各维不相互独立等价于协方差矩阵非对角阵，然而这是一个特例，当类条件分布不满足高斯分布时，那么就不存在这个关系了。

因此为了普遍性，我们仍然是先进行 n 次高斯分布生成数据，作为相应维特征的值，然后进行拼接，接着为了让各维不独立。

例如，实验中，我们生成不独立的数据，采用的是协方差矩阵不是正定阵：

```
        Y[train_sample:] = 0
    else:
        cov = np.mat([[0.3, 0.5], [0.5, 0.4]])
        X[:train_sample, :] = np.random.multivariate_normal(pos_mean, cov, train_sample)
        X[train_sample:, :] = np.random.multivariate_normal(neg_mean, cov, train_sample)
        Y[:train_sample] = 1
        Y[train_sample:] = 0
```

2.2 梯度下降法

梯度下降中的步长需要通过不断调优，找到合适的步长；
同时，实验中我们初始化参数矩阵 W 为全 0；
其中 λ 不等于 0 的时候就是不加正则项的情况。

```
#梯度下降法
def gradient_descent(X, Y, w, lamuda, alpha=0.1, epsilon=0.1):
    cnt = 0 #记录迭代次数
    size = X.shape[1] #size为数据量
    new_loss = loss(X, Y, w, lamuda) #计算损失函数的值，通过比较损失函数的差结束迭代
    # print(new_loss)
    while True:
        cnt += 1
        old_loss = new_loss
        wX = np.zeros((size, 1))
        wX = np.dot(w, X) #初始化wX，方便后续计算sigmoid函数
        gradient_w = - np.dot(X, (Y - sigmoid(wX))) /size #归一化处理 3x1
        old_w = w
        w = w - alpha * lamuda * w - alpha * gradient_w.T #迭代w的值
        new_loss = loss(X, Y, w, lamuda)
        if old_loss < new_loss: #步长过大，下降不了
            w = old_w
            alpha /= 2
            continue
        if old_loss - new_loss < epsilon: #结束迭代
        #if np.linalg.norm(gradient_w) <= epsilon:
            break
    print(cnt)
    return w
```

2.3 共轭梯度法

我先根据实验一写了一般共轭梯度法，但是正定阵没有弄明白，所以学习效果并不好：

```
# cnt = 0 #限制迭代次数
# size = X.shape[1]
# wX = np.zeros((size, 1))
# wX = np.dot(w, X)
# A = np.dot(X, X.T) #方便计算，为原式的正定矩阵 3x3
# gradient_w = derivative(w, X, Y, lamuda) #计算第一次梯度方向 3x1
# d = - gradient_w #第一次迭代方向为负梯度方向 3x1
# alpha = - np.dot(d.T, gradient_w) / np.dot(d.T, A).dot(d) #初始化步长
# while cnt<=2*size: #控制迭代次数为w的维数
#     alpha = - np.dot(d.T, gradient_w) / np.dot(d.T, A).dot(d) #更新步长，使损失函数达到最小的步长
#     w = w + alpha * d.T #更新w矩阵，沿着共轭方向下降
#     gradient_w = np.dot(X, (Y - sigmoid(wX))) #更新梯度，计算共轭方向和步长需要
#     beta = np.dot(d.T, A).dot(gradient_w) / np.dot(d.T, A).dot(d) #计算共轭方向需要的线性关系系数
#     d = -gradient_w + beta * d #得到共轭方向
#     cnt = cnt + 1
# print(cnt)
```

后来我又写了一版，首先为了满足共轭梯度法的条件，我们对方程组的形式做了前文所提及的矩阵变换。

实验中 W 初始化为全 0 向量；

迭代过程，我们设定算法终止的条件是$r^k=0$，但由于浮点运算本身就具有不准确性，因此只要小于 10^-10 即可，但是这样写，迭代时间太长了。根据共轭梯度法下降的维度是有限的性质，我们让这个成为迭代的次数，实事证明，分类效果还是不错的。

```
# 共轭梯度法
def conjugate_gradient(X, Y, w, epsilon):
    cnt = 0
    Q = np.dot(X, X.T) + lamuda * np.eye(X.shape[0])
    w = np.zeros((1, X.shape[0]))
    gradient_w = derivative(w, X, Y, lamuda)
    # gradient_w = np.dot(w, X).dot(X.T) - np.dot(X, Y).T + lamuda * w # 3x2n,2nx3,1x3
    r = -gradient_w
    d = r
    # for i in range(X.shape[0]):
    # while np.linalg.norm(gradient_w) >= 0.1 :
    while cnt < X.shape[1]:
        cnt += 1
        alpha = np.dot(r, r.T) / np.dot(d, Q).dot(d.T)
        r_old = r
        w = w + alpha * d
        r = r - alpha * np.dot(d, Q)
        beta = np.dot(r, r.T) / np.dot(r_old, r.T)
        d = r + beta * d
    print(cnt)
```

2.4 牛顿法

牛顿法的算法实现很容易，只需要按照前文的矩阵公式进行计算迭代，每次迭代后计算当前 loss 值，与前一次的 loss 进行比较，若差值小于一个阈值，则终止算法：
为了方便计算，我们让一阶导和二阶导变成函数，拿出来计算。
一阶导：

```
#一阶导
def derivative(w, X, Y, lamuda):
    result = np.zeros((1, X.shape[0]))
    for i in range(X.shape[1]):
        multi = np.dot(w, X[:, i])
        result += (Y[i] - math.exp(multi) / (1 + math.exp(multi))) * X[:, i].T
    return -1 * result + lamuda * w
```

二阶导（海瑟阵）

```
#二阶导（海瑟阵）
def second_derivative(w, X, lamuda):
    result = np.eye(X.shape[0]) * lamuda
    for i in range(X.shape[1]):
        matrix = X[:, i].T.reshape(1, X.shape[0])
        multi = np.dot(w, X[:, i])
        r = math.exp(multi) / (1+math.exp(multi))
        result += np.dot(matrix.T, matrix) * r * (1-r)
    return np.linalg.pinv(result)
```

牛顿法：

```
#牛顿法
def newton2(X, Y, w, epsilon, lamuda):
    cnt = 0
    while True:
        cnt += 1
        gradient = derivative(w, X, Y, lamuda)
        if np.linalg.norm(gradient) < epsilon:
            break
        w -= np.dot(gradient, second_derivative(w, X, lamuda))
    print(cnt)
    return w
```

四、实验结果与分析

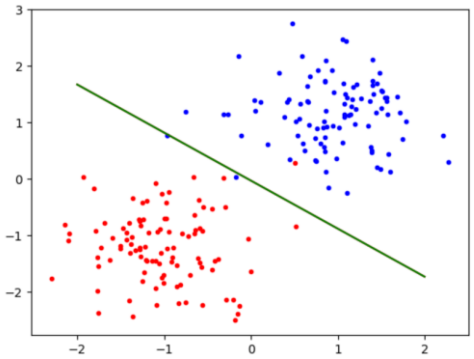
说明：由于当特征维数是二维时，决策面实际上是一条直线，可以在二维坐标系中画出；因此，当特征维数为 2 时，我们画出决策面以及 loss 曲线，而超过二维的只画 loss 曲线；

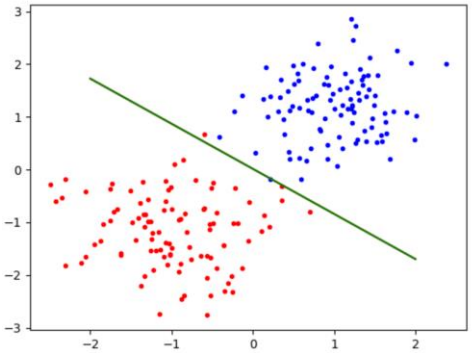
设训练样本数为 N；

4.1 满足朴素贝叶斯假设的数据

4.1.1 梯度下降法（左边为不带正则项，右边为带正则项）

N = 100

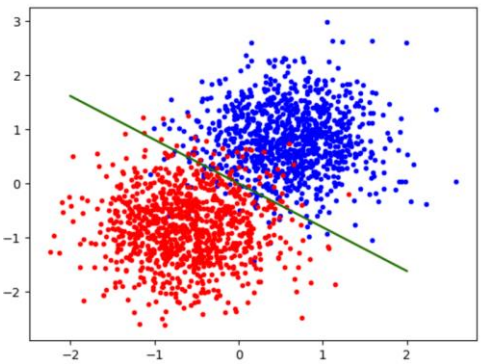


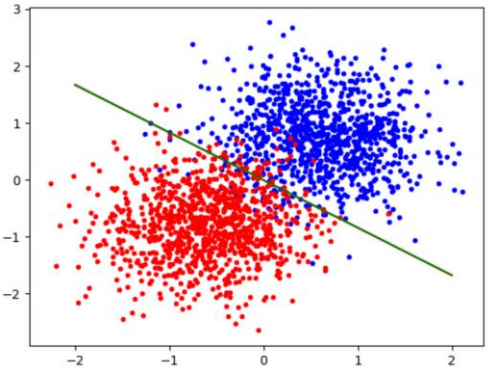


求解的 w，上面为不带正则项，下面为带正则项：

```
[0.03376147 1.09516495 1.28867664]
[0.03098    0.99871607 1.17525404]
```

N = 1000

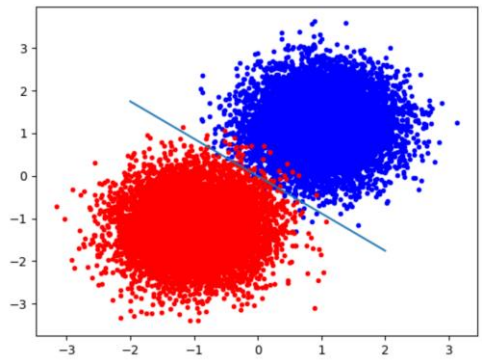
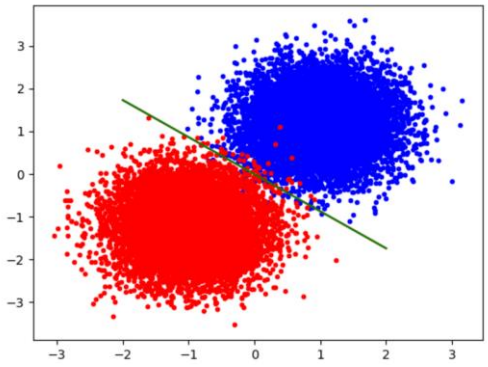




求解的 w，上面为不带正则项，下面为带正则项：

```
[-0.00930638  1.12217089  1.29182219]
[-0.00853071  1.02348309  1.17783671]
```

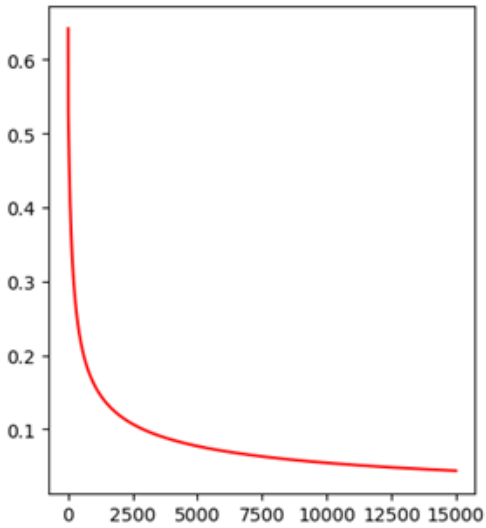
N = 10000

求解的 w，上面为不带正则项，下面为带正则项：

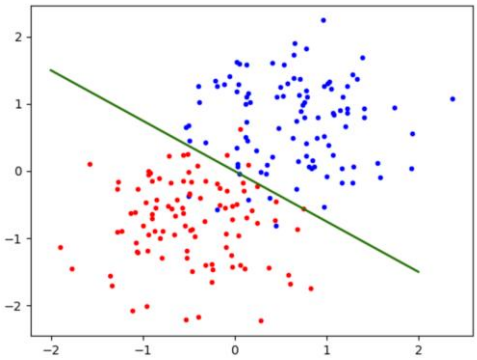
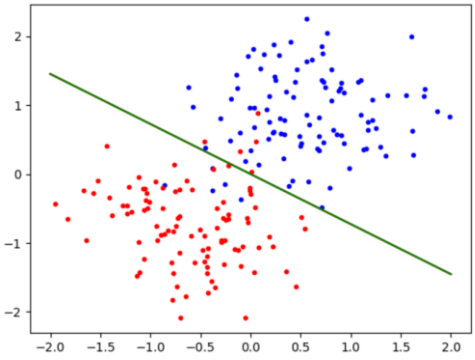
```
[0.00451091 1.14703702 1.43696266]
[0.00412615 1.04575199 1.309383  ]
```

梯度下降法的下降速率（以 N=1000 为例）：



4.1.2 共轭梯度法（左边为不带正则项，右边为带正则项）

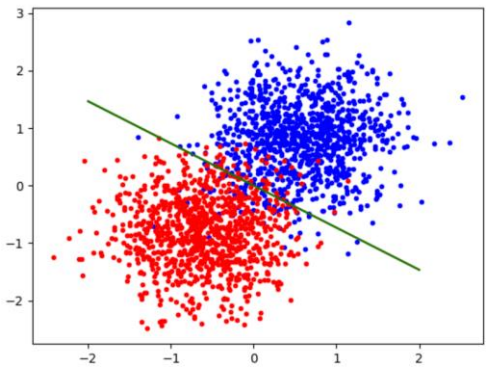
N = 100

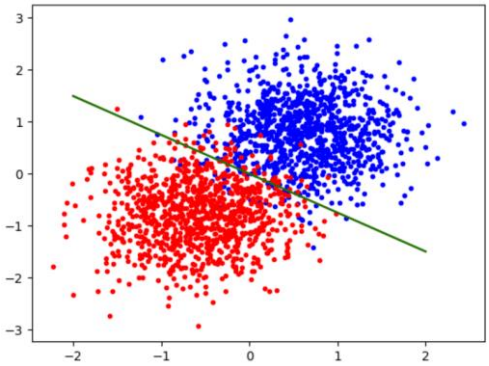



求解的 w，上面为不带正则项，下面为带正则项：

```
[-2.74390077e-27  2.26334608e-01  3.02027132e-01]
[-2.74390077e-27  2.26334608e-01  3.02027132e-01]
```

N = 1000

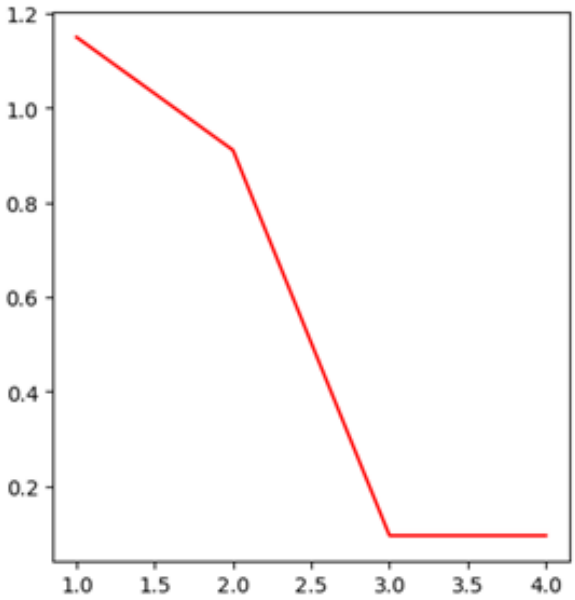




求解的 w，上面为不带正则项，下面为带正则项：

```
[-5.96217938e-27  2.18903075e-01  2.93000753e-01]
[-5.96217938e-27  2.18903075e-01  2.93000753e-01]
```

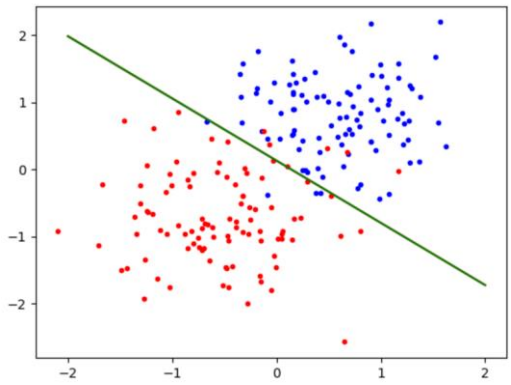
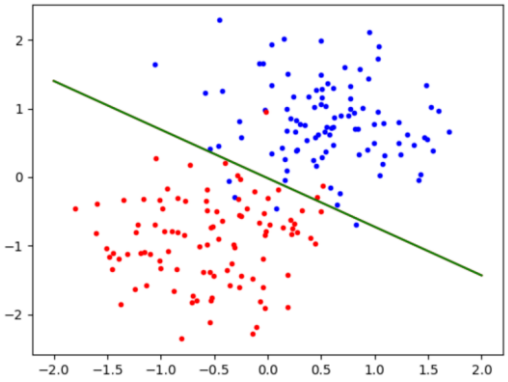
共轭梯度法的下降速率（以 N=1000 为例）：



因为共轭梯度法下降的次数就是维度，所以只会迭代 3 次，下降 3 次

4.1.3 牛顿法（左边为不带正则项，右边为带正则项）

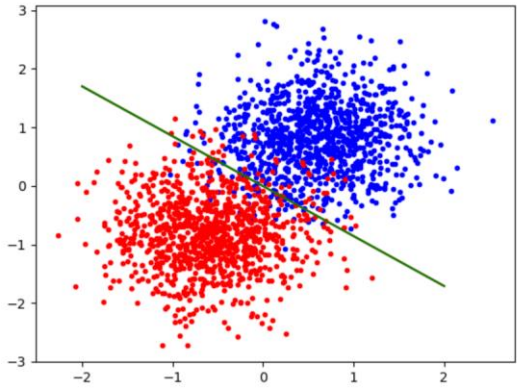
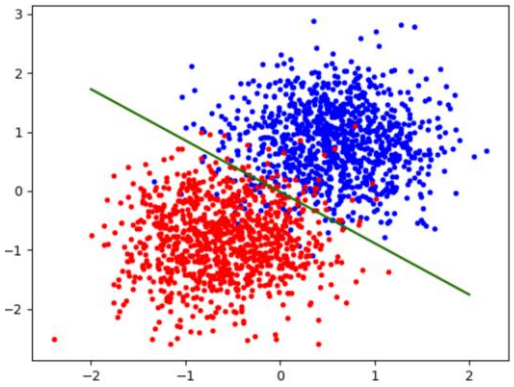
N = 100

求解的 w，上面为不带正则项，下面为带正则项：

```
[-0.34797829  2.47866719  2.67150714]
[-0.34797829  2.47866719  2.67150714]
```

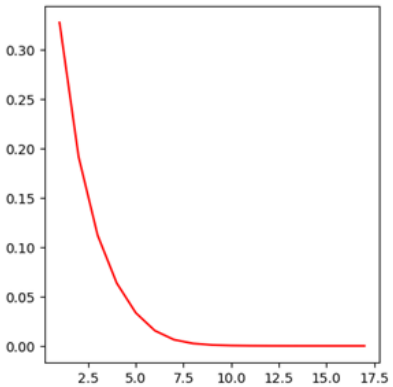
N = 1000

求解的 w，上面为不带正则项，下面为带正则项：

```
[0.02087316 3.33463118 3.91186914]
[0.02087316 3.33463118 3.91186914]
```

牛顿法的下降速率（以 N=1000 为例）：



4.1.4 正确率

关于正确率，做了一个表格：
GD 梯度下降，CG 共轭梯度，NT 牛顿法

	GD 无正则	GD 有正则	CG	NT 无正则	NT 有正则
N=50	0.957	0.957	0.87	0.97	0.97
N=100	0.98	0.98	0.987	0.977	0.977
N=1000	0.997	0.997	0.97		

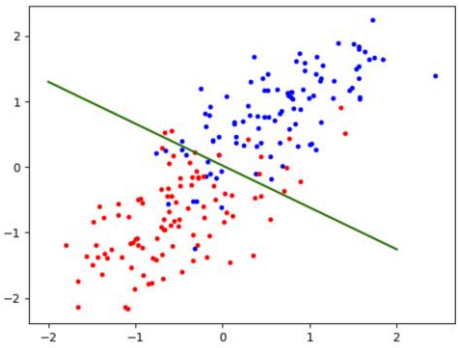
可见：
随着训练样本数的增加，准确率也会越来越高；
加入正则项的 loss 值比不加正则项的 loss 值小，即模型效果越好；

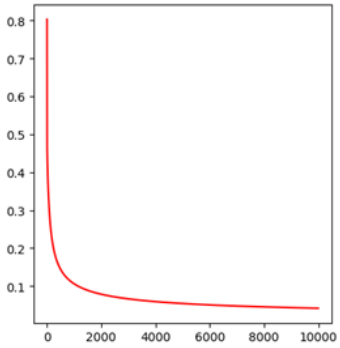
4.2 不满足朴素贝叶斯假设的数据

不满足朴素贝叶斯假设的数据，我们不再对 N 进行讨论，设 N=100，与满足朴素贝叶斯假设且 N=100 的进行对比；

由于加正则项实际上是为了使得结果更优，因此也不再对不加正则项的梯度下降法、共轭梯度法、牛顿法进行讨论，我们只看加正则项的方法下，模型的效果如何；

4.2.1 梯度下降法

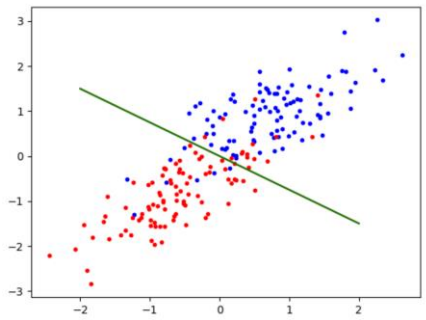


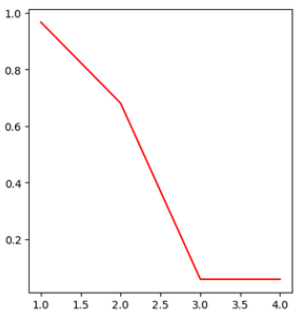


求解的 w:

```
[0.00334268 0.44404517 2.58845706]
```

4.2.2 共轭梯度法

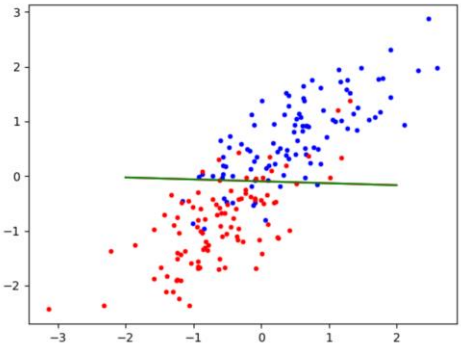


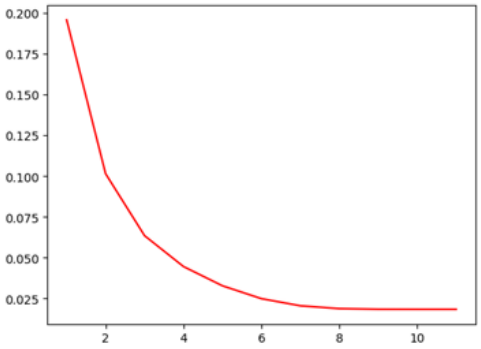


求解的 w:

```
[-1.43554106e-33  1.55395517e-01  2.07651584e-01]
```

4.2.3 牛顿法





求解的 w：

```
[0.23726758 0.08737291 2.47170637]
```

感觉不满足朴素贝叶斯的数据，无论哪种方法学习的正确率都不如之前。

五、结论

1. 梯度下降法主要沿着负梯度方向进行更新，只引入了一阶导数。
2. 看了牛顿法，在这里也写点感想，牛顿法则将一阶导数和二阶导数相结合进行参数的迭代更新，但是其二阶导数矩阵涉及到海瑟矩阵求逆（海瑟矩阵又是一个令人头疼的东西，微积分学的东西很多都忘了， 还是一个个查的慢慢捡起来），复杂度较高，因此基于牛顿法之上，出现了拟牛顿法，其主要思想是构造一个新的矩阵来近似替代海瑟矩阵的逆，这样可以避免牛顿法中的复杂的海瑟矩阵求逆操作。
3. 本质上来说，牛顿法是二阶收敛，而梯度下降是一阶收敛，所以牛顿法收敛更快，但是每次迭代的时间，牛顿法比梯度下降时间长。牛顿法就是用一个二次曲面去拟合你当前所处位置的局部曲面，而梯度下降法使用一个平面去拟合当前的局部曲面，所有牛顿法的下降路径会更简单，更符合最优路径。
4. 共轭梯度法强就强在不只是二阶收敛，而是基于梯度下降中的负梯度方向之外，引入了共轭向量，可以使收敛更快。
5. 牛顿法是二阶优化方法，拟牛顿法和共轭梯度法一般叫做 1.5 阶优化方法，梯度下降法及其变形则是一阶优化方法。
6. 加入惩罚项在本次实验中没有看到太大作用，可能维度也只是 3 维，不大会出现过拟合的情况。
7. 逻辑回归对二分类问题有很大作用，分类效果很好；
8. 逻辑回归的前提是数据满足朴素贝叶斯假设：特征的各维独立、类条件分布满足高斯分布、各类别协方差矩阵相等(与类别无关)；
9. 当数据不满足朴素贝叶斯假设时，分类效果可能会变差；
10. 增大训练样本数可以使得分类效果更好；

六、参考文献

1. 李庆扬. 王能超. 易大义 《数值分析》
2. 逻辑回归
https://blog.csdn.net/weixin_39445556/article/details/83930186
3. 逻辑回归——牛顿法矩阵实现方式
https://www.cnblogs.com/f-young/p/8100127.html

七、附录：源代码（带注释）

```
'''
逻辑回归

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。

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。

验证：1.可以手工生成两个分别类别数据（可以用高斯分布），验证你的算法。考察类条件分布不满足朴素贝叶斯假设，会得到什么样的结果。
2. 逻辑回归有广泛的用处，例如广告预测。可以到 UCI 网站上，找一实际数据加以测试。
'''

import math

import numpy as np
import matplotlib.pyplot as plt



def sigmoid(wx):
    wx = wx.T
    size = wx.shape[0]
    for i in range(size):
        wx[i] = 1 / (1 + np.exp(-wx[i]))
    return wx

def generate_data(train_sample, navie = True):
    pos_mean = [0.6, 0.8] #正例的两维度均值
    neg_mean = [-0.6, -0.8] #反例的两维度均值
    X = np.zeros((2 * train_sample, 2))
    Y = np.zeros((2 * train_sample, 1))
    if navie:
        cov = np.mat([[0.3, 0], [0, 0.4]])
        X[:train_sample, :] = np.random.multivariate_normal(pos_mean, cov, train_sample)
```

```
            X[train_sample:,  :]  =  np.random.multivariate_normal(neg_mean,  cov,
train_sample)
            Y[:train_sample] = 1
            Y[train_sample:] = 0
        else:
            cov = np.mat([[0.3, 0.5], [0.5, 0.4]])
            X[:train_sample,  :]  =  np.random.multivariate_normal(pos_mean,  cov,
train_sample)
            X[train_sample:,  :]  =  np.random.multivariate_normal(neg_mean,  cov,
train_sample)
            Y[:train_sample] = 1
            Y[train_sample:] = 0
        # print(X)
        # print(Y)
        # plt.plot(X[:train_sample, :], 'go')
        # plt.plot(X[train_sample:, :], 'ro')
        plt.scatter(X[:train_sample, 0], X[:train_sample, 1], c = 'b', marker = '.')
        plt.scatter(X[train_sample:, 0], X[train_sample:, 1], c = 'r', marker = '.')
        return X.T, Y

#损失函数，(不)带正则项(极大似然）,并对 loss 做归一化处理
def loss(X, Y, w, lamuda=0):
        size = X.shape[1]
        wX = np.zeros((size, 1))
        wX = np.dot(w, X).T
        part1 = np.dot(Y.T, wX)
        part2 = 0
        for i in range(size):
            part2 += np.log(1+np.exp(wX[i]))
        L_w = part1 - part2 - lamuda * np.dot(w, w.T) / 2
        return -L_w /size

#梯度下降法
def gradient_descent(X, Y, w, alpha=0.1, epsilon=0.1, lamuda = 0):
        cnt = 0 #记录迭代次数
        size = X.shape[1] #size 为数据量
        new_loss = loss(X, Y, w, lamuda) #计算损失函数的值，通过比较损失函数的差结束迭代
        # print(new_loss)
        while True:
            cnt += 1
            old_loss = new_loss
            wX = np.zeros((size, 1))
            wX = np.dot(w, X) #初始化 wX，方便后续计算 sigmoid 函数
            gradient_w = - np.dot(X, (Y - sigmoid(wX))) /size #归一化处理  3x1
```

```
            old_w = w
            w = w - alpha * lamuda * w - alpha * gradient_w.T #迭代 w 的值
            new_loss = loss(X, Y, w, lamuda)
            if old_loss < new_loss: #步长过大，下降不了
                w = old_w
                alpha /= 2
                continue
            if old_loss - new_loss < epsilon: #结束迭代
            #if np.linalg.norm(gradient_w) <= epsilon:
                break
        print(cnt)
        return w

# 共轭梯度法
def conjugate_gradient(X, Y, w, epsilon, lamuda=0):
        cnt = 0
        Q = np.dot(X, X.T) + lamuda * np.eye(X.shape[0])
        w = np.zeros((1, X.shape[0]))
        gradient_w = derivative(w, X, Y, lamuda)
        # gradient_w = np.dot(w, X).dot(X.T) - np.dot(X, Y).T + lamuda * w # 3x2n,2nx3,1x3
        r = -gradient_w
        d = r
        # for i in range(X.shape[0]):
        # while np.linalg.norm(gradient_w) >= 0.1 :
        while cnt < X.shape[0]:
            cnt += 1
            alpha = np.dot(r, r.T) / np.dot(d, Q).dot(d.T)
            r_old = r
            w = w + alpha * d
            r = r - alpha * np.dot(d, Q)
            beta = np.dot(r, r.T) / np.dot(r_old, r.T)
            d = r + beta * d
        print(cnt)
        # cnt = 0 #限制迭代次数
        # size = X.shape[1]
        # wX = np.zeros((size, 1))
        # wX = np.dot(w, X)
        # A = np.dot(X, X.T) #方便计算，为原式的正定矩阵  3x3
        # gradient_w = derivative(w, X, Y, lamuda) #计算第一次梯度方向  3x1
        # d = - gradient_w #第一次迭代方向为负梯度方向  3x1
        # alpha = - np.dot(d.T, gradient_w) / np.dot(d.T, A).dot(d) #初始化步长
        # while cnt<=2*size: #控制迭代次数为 w 的维数
        #         alpha = - np.dot(d.T, gradient_w) / np.dot(d.T, A).dot(d) #更新步长，使损失函数
达到最小的步长
```

```
        #         w = w + alpha * d.T #更新 w 矩阵，沿着共轭方向下降
        #         gradient_w = np.dot(X, (Y - sigmoid(wX))) #更新梯度，计算共轭方向和步长需
要
        #         beta = np.dot(d.T, A).dot(gradient_w) / np.dot(d.T, A).dot(d) #计算共轭方向需要
的线性关系系数
        #         d = -gradient_w + beta * d #得到共轭方向
        #         cnt = cnt + 1
        # print(cnt)
        return w


            # 1x2n,2nx3,3x2n,2nx1    3x2n,2nx3,1x2n,2nx1      1x3 3x3
            # 3x2n, 2nx1

# 牛顿法
def newton(X, Y, w, epsilon, lamuda=0):
        cnt = 0
        size = X.shape[1]
        I = np.eye(X.shape[0])
        wX = np.zeros((size, 1))
        wX = np.dot(w, X)
        while cnt < 1000:
                cnt += 1
                gradient_w = np.dot(X, (Y - sigmoid(wX))) #1x3
                # print(np.linalg.norm(gradient_w))
                gradient2_w = np.dot(X, X.T) * np.dot(sigmoid(wX).T,sigmoid(-wX)) + lamuda * I
                w = w - np.dot(gradient_w.T, np.linalg.inv(gradient2_w))
                if np.linalg.norm(gradient_w) <= epsilon:
                        break
        print(cnt)
        return w

#一阶导
def derivative(w, X, Y, lamuda=0):
        result = np.zeros((1, X.shape[0]))
        for i in range(X.shape[1]):
                multi = np.dot(w, X[:, i])
                result += (Y[i] - math.exp(multi) / (1 + math.exp(multi))) * X[:, i].T
        return -1 * result + lamuda * w

#二阶导（海瑟阵）
def second_derivative(w, X, lamuda=0):
        result = np.eye(X.shape[0]) * lamuda
        for i in range(X.shape[1]):
```

```
            matrix = X[:, i].T.reshape(1, X.shape[0])
            multi = np.dot(w, X[:, i])
            r = math.exp(multi) / (1+math.exp(multi))
            result += np.dot(matrix.T, matrix) * r * (1-r)
        return np.linalg.pinv(result)

#牛顿法
def newton2(X, Y, w, epsilon, lamuda=0):
        cnt = 0
        while True:
            cnt += 1
            gradient = derivative(w, X, Y, lamuda)
            if np.linalg.norm(gradient) < epsilon:
                break
            w -= np.dot(gradient, second_derivative(w, X, lamuda))
        print(cnt)
        return w


lamuda = 1
epsilon = 1e-3
alpha = 0.1
train_sample = 100
accept_gradient = 0.1 #梯度下降法阈值
loss_1 = 0
X, Y = generate_data(train_sample, False)
train_X = np.ones((X.shape[0] + 1, 2 * train_sample)) #初始化训练样本
train_X[1:X.shape[0]+1, :] = X #训练样本变成增广矩阵
w = np.zeros((1, X.shape[0] + 1)) #初始化 w
w1 = np.zeros((1, X.shape[0] + 1)) #初始化 w1(带正则项)

#  得到 w
# w = gradient_descent(train_X, Y, w, alpha, epsilon)
# w1 = gradient_descent(train_X, Y, w, alpha, epsilon, lamuda)

# w = newton(train_X, Y, w, epsilon, lamuda)
# w1 = newton(train_X, Y, w, epsilon, lamuda)

# w = conjugate_gradient(train_X, Y, w, epsilon, lamuda)
# w1 = conjugate_gradient(train_X, Y, w, epsilon, lamuda)

w = newton2(train_X, Y, w, epsilon)
w1 = newton2(train_X, Y, w, epsilon, lamuda)
```

```
w = w[0]
w1 = w1[0]


# print(loss)
# print(train_X)

#  得到回归面
print(w)
print(w1)
test_x = np.linspace(-2, 2)
test_y = - (w[0] + w[1] * test_x) / w[2]
test_y_1 = - (w1[0] + w1[1] * test_x) / w1[2]
plt.plot(test_x, test_y, 'r')
plt.plot(test_x, test_y_1, 'g')
plt.show()
```