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：必修
实验题目：逻辑回归

学号：1181000420
姓名：韦昆杰

一、实验目的

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：

- Windows10
- PyCharm
- VSCode

三、设计思想(本程序中的用到的主要算法及数据结构)

3.1 不带正则项的Logistic回归模型

3.1.1 算法原理

对于线性回归的预测函数，我们定义如下图

$$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \ldots + \theta_j x_j \quad = \theta^T x \quad = \begin{bmatrix} \theta_0 & \theta_1 & \ldots & \theta_j \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \ldots \\ x_j \end{bmatrix}$$

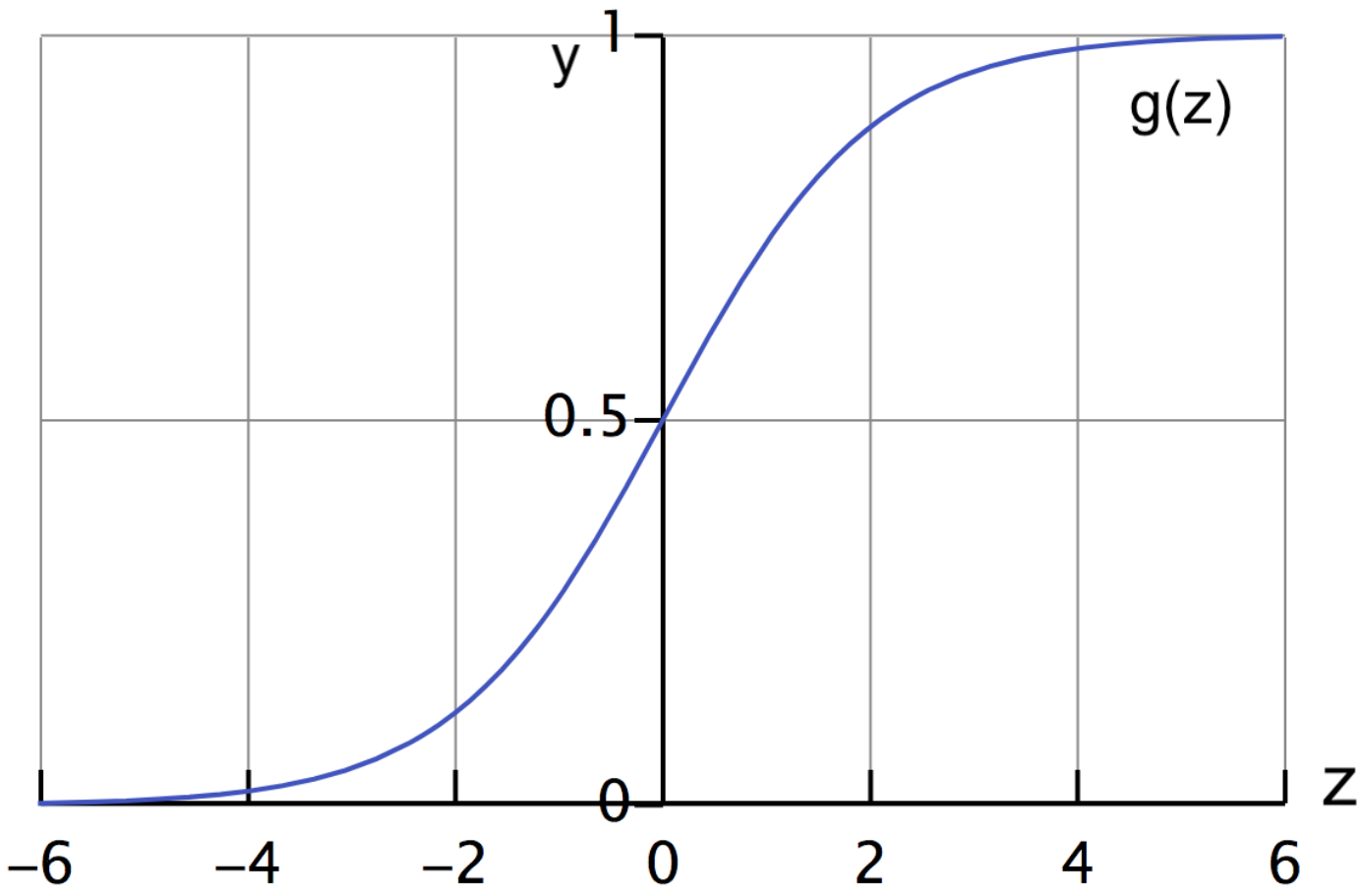
$$Where\ x_0 = 1$$

上面即为线性回归模型，Logistic回归只是在此基础上用线性回归模型的预测结果来逼近真实标记的对数几率。对于Logistic回归模型，我们关注根据特征预测结果为0和1的分类问题。为了将预测与label进行比较，我们希望将预测限制在0到1之间的某些值。因此我们采用sigmoid函数模型定义Logistic回归的预测函数。

$$Sigmoid\ Function: \quad g(z) = \frac{1}{1 + e^{(-z)}}$$

$$Hypothesis: \quad h_\theta(x) = \frac{1}{1 + e^{(-\theta^T x)}}$$

如下图，sigmoid函数在0~1之间并在0.5处分界，若x>0.5该函数迅速接近1，否则迅速接近0。在x取极大或极小值时，y值的变化不明显，避免了极端值对分类的影响。因此我们可以将这个函数作为我们的预测函数：



因此，我们可以将预测概率定义如下

$$P(Y=1|x_i;w)=h(x_i) \qquad P(Y=0|x_i;w)=1-h(x_i) \tag{1}$$

综合起来就是：

$$P(Y|x_i;w)=h(x_i)^y(1-h(x_i))^y \tag{2}$$

对于全部的m组数据，假设它们相互独立，我们可以写成下面的形式

$$P(Y|X;w)=\prod_{i\in(1,m)}P(Y|x_i;w)=\prod_{i\in(1,m)}h(x_i)_i^y(1-h(x_i))^{1-y_i} \tag{3}$$

对(3)式左右两边取对数可得下面的结果

$$J(w)=log(P(Y|X;w))=\sum_{i=1}^{m}logh(x_i)+(1-y_ilog(1-h(x_i)) \tag{4}$$

(5)式即为似然函数，我们实验要求出最大似然，我们可以定义对应的cost函数：

$$Cost(h_\theta(x),y)=\begin{cases}-log(h_\theta(x)) & \text{if } y=1\\ -log(1-h_\theta(x)) & \text{if } y=0\end{cases}$$

为了计算的方便性，我们将上面两个式子写在一起

$$Cost(h_\theta(x),y)=-ylog(h_\theta(x))-(1-y)log(h_\theta(x))$$

最后所有训练数据的代价的和就是不加正则项的的代价函数，代价函数最小，似然函数最大，分类的效果最好

$$J(\theta)=\frac{1}{m}\sum_{i=1}^{m}Cost(h_\theta(x^{(i)}),y^{(i)})$$

$$J(\theta)=\frac{1}{m}[\sum_{i=1}^{m}-y^{(i)}log(h_\theta(x^{(i)}))+(1-y^{(i)})log(1-h_\theta(x^{(i)}))]$$

$$m=number\ of\ samples$$

3.1.2 算法实现

本实验中，sigmoid函数的python实现如下：

```
def sigmoid(z):
    return 1.0 / (1 + np.exp(-z))
```

hypothesis函数的python实现如下：

```
# hypothesis = sigmoid(X @ W)
def hypothesis(X, W):
    return sigmoid(X @ W)
```

代价函数的python实现如下：

```
# Vectorized cost function     cost = (-y'log(h)-(1-y)'log(1-h))/m
def cost_function(X, W, Y):
    cost = 0
    matrix_one = np.ones((Y.shape[0], Y.shape[1]))
    h = hypothesis(X, W)
    m = len(Y)
    cost += (-Y.T @ np.log(h) - (matrix_one - Y).T @ np.log(matrix_one - h)) / m
    cost = float(cost)
    return cost
```

3.2 带正则项的Logistic回归模型

3.2.1 算法原理

下面我们对代价函数进行正则化来避免过拟合，加入惩罚项的代价函数为下图所示

$$L1: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad ||\theta||_1 <= C$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{m}\sum_{j=1}^{n} |\theta_j|$$

$$L2: \quad J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) \quad s.t. \quad ||\theta||_2^2 <= C^2$$

$$J(\theta) = \frac{1}{m}\sum_{i=1}^{m} Cost(h_\theta(x^{(i)}), y^{(i)}) + \frac{\lambda}{2m}\sum_{j=1}^{n} \theta_j^2$$

$$m = number\ of\ samples, \quad n = number\ of\ features$$

3.2.2 算法实现

本实验采用L2正则化，(梯度下降法)对应的W更新的python代码如下：

```
W = W - (learning_rate / m) * (X.T @ (h - Y) + Lambda / m * W)
```

3.3 使用梯度下降法求解不带正则项的Logistic回归模型

3.3.1 算法原理

本实验中，我们使用梯度下降法求出代价函数的最小值，核心思想是：通过梯度下降法沿着梯度方向来一步步的迭代求解，得到最小化的损失函数和模型参数值 本实验梯度下降的更新公式为：

$$W_{new} = W_{old} - \frac{rate}{m}X'(h - Y)$$

3.3.2 算法实现

每一次更新W的python代码如下

```
W = W - (learning_rate / m) * X.T @ (h - Y)
```

3.4 使用梯度下降法求解带正则项的Logistic回归模型

3.4.1 算法原理

使用梯度下降法求解带正则项的Logistic回归模型和求解不带正则项所用的算法原理基本一样，只是在更新W时加入了正则项，更新公式为：

$$W_{new} = W_{old} - \frac{rate}{m}(X'(h - Y) + \frac{\lambda}{m})$$

3.4.2 算法实现

本实验采用L2正则化，对应的W更新的python代码如下：

```
W = W - (learning_rate / m) * (X.T @ (h - Y) + Lambda / m * W)
```

3.5 使用牛顿法求解带正则项的Logistic回归模型

3.5.1 算法原理

牛顿法是一种在实数域和复数域上近似求解方程的方法,通过使用函数$f(x)$的泰勒级数的前面几项来寻找方程$f(x) = 0$的根。 经过某一点$(x_i, f(x_i))$的切线方程为$f(x) = f(x_i) + f'(x_i)(x - x_i)$,令该切线方程等于0，得到下式：

$$x_{i+1} = xi - \frac{f(x_i)}{f'(x_i)}$$

在本实验中，我们使用函数$f(x)$的二阶泰勒级数来求方程$f(x) = 0$的根，对应的迭代公式为：

$$x_{i+1} = xi - \frac{f'(x_i)}{f''(x_i)}$$

本实验中我们要使损失函数最小，其实就是解决$\nabla f(\theta) = 0$ 我们对$f(\theta)$进行二阶泰勒展开：

$$f(\theta) = f(\theta^{(k)}) + \nabla f(\theta^{(k)})(\theta - \theta^k) + \frac{1}{2}\nabla^2 f(\theta^{(k)})(\theta - \theta^k)^2$$

令$\nabla f(\theta) = 0$，我们可以求出θ的更新公式：

$$\theta^{(k+1)} = \theta^{(k)} - \nabla^2 f(\theta^k)^{-1}\nabla f(\theta^{(k)})$$

对于本实验，W更新的迭代公式为

$$W_{new} = W_{old} - H^{-1}\nabla J(\theta^{(t)})$$

其中H为Hassian矩阵

3.5.2 算法实现

本实验W更新的python代码如下：

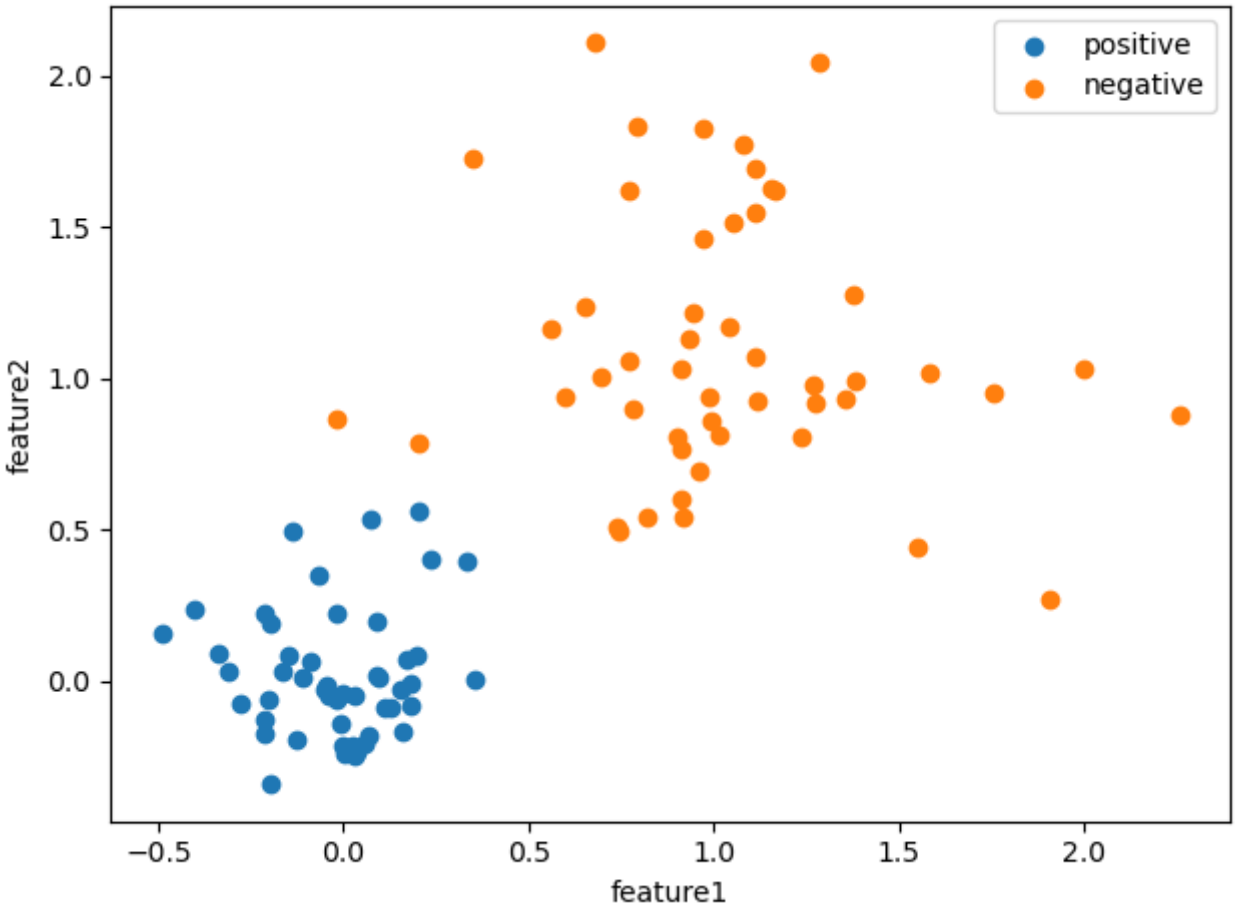
```
h = hypothesis(X, W)
U = 1 / n * X.T @ (h - Y)  # (m,1)
H = 1 / n * np.dot(np.dot(np.dot(X.T, np.diag(h.reshape(n))), np.diag(1 - h.reshape(n))), X)  #
W = W - np.linalg.inv(H) @ U  # (m,1)
```

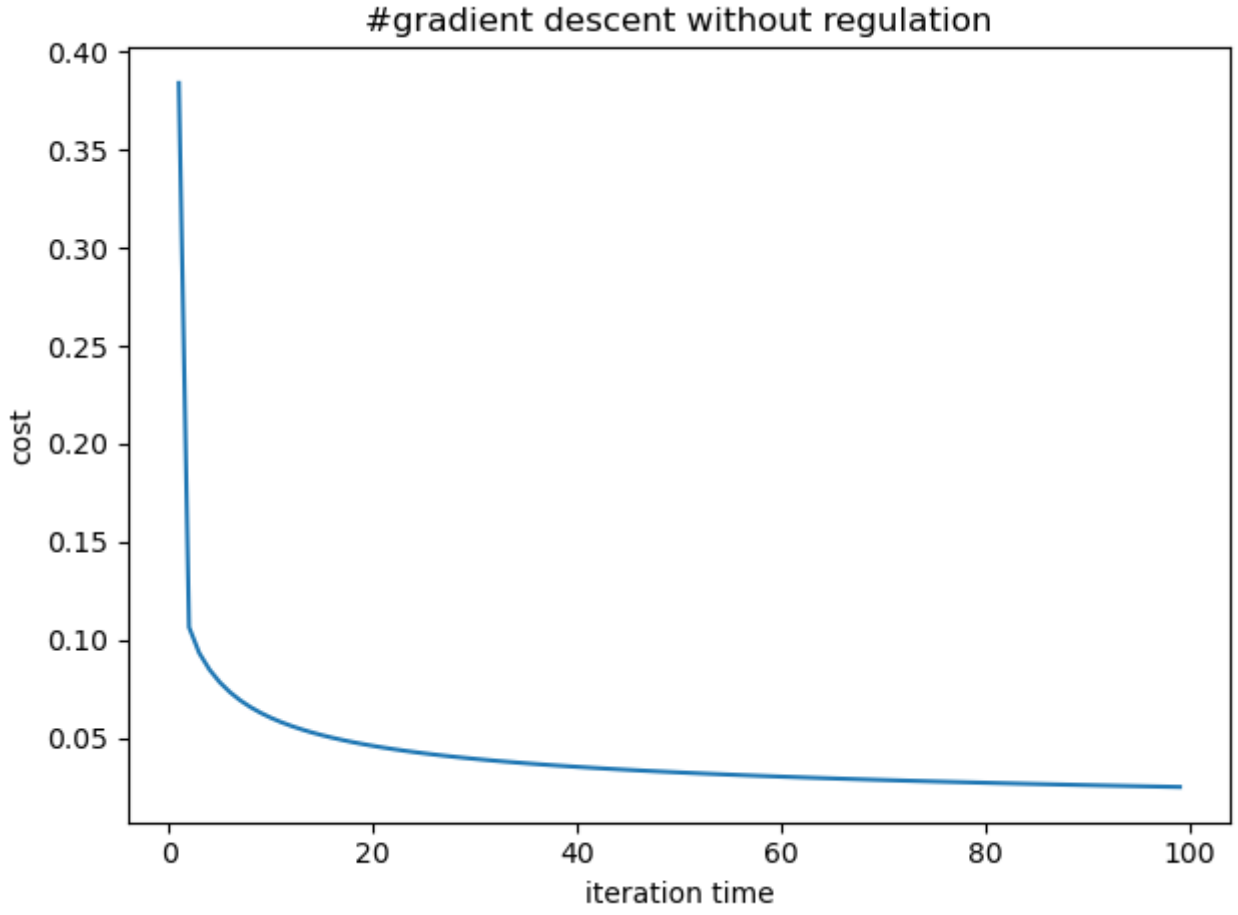
四、实验结果分析

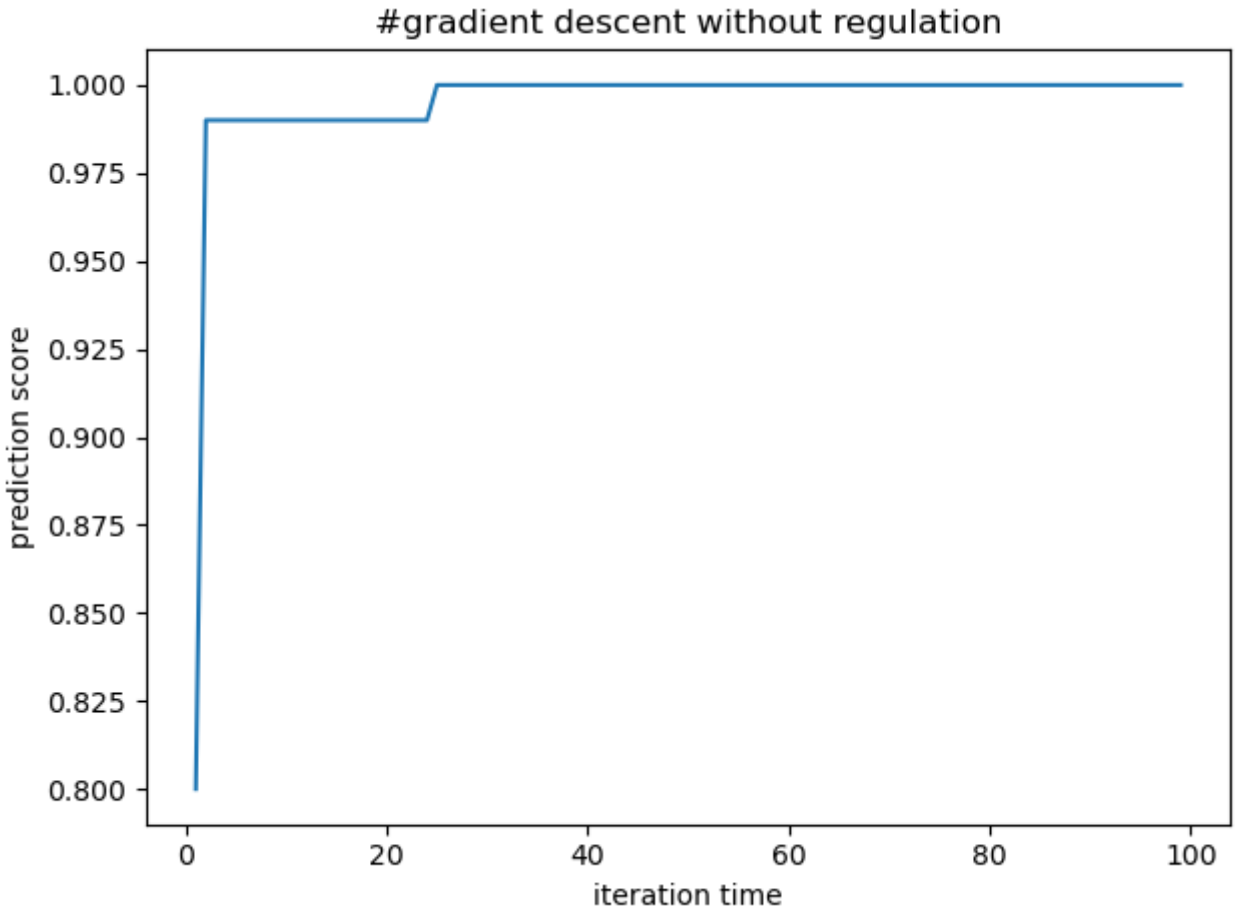
4.1 使用梯度下降法求解Logistic回归

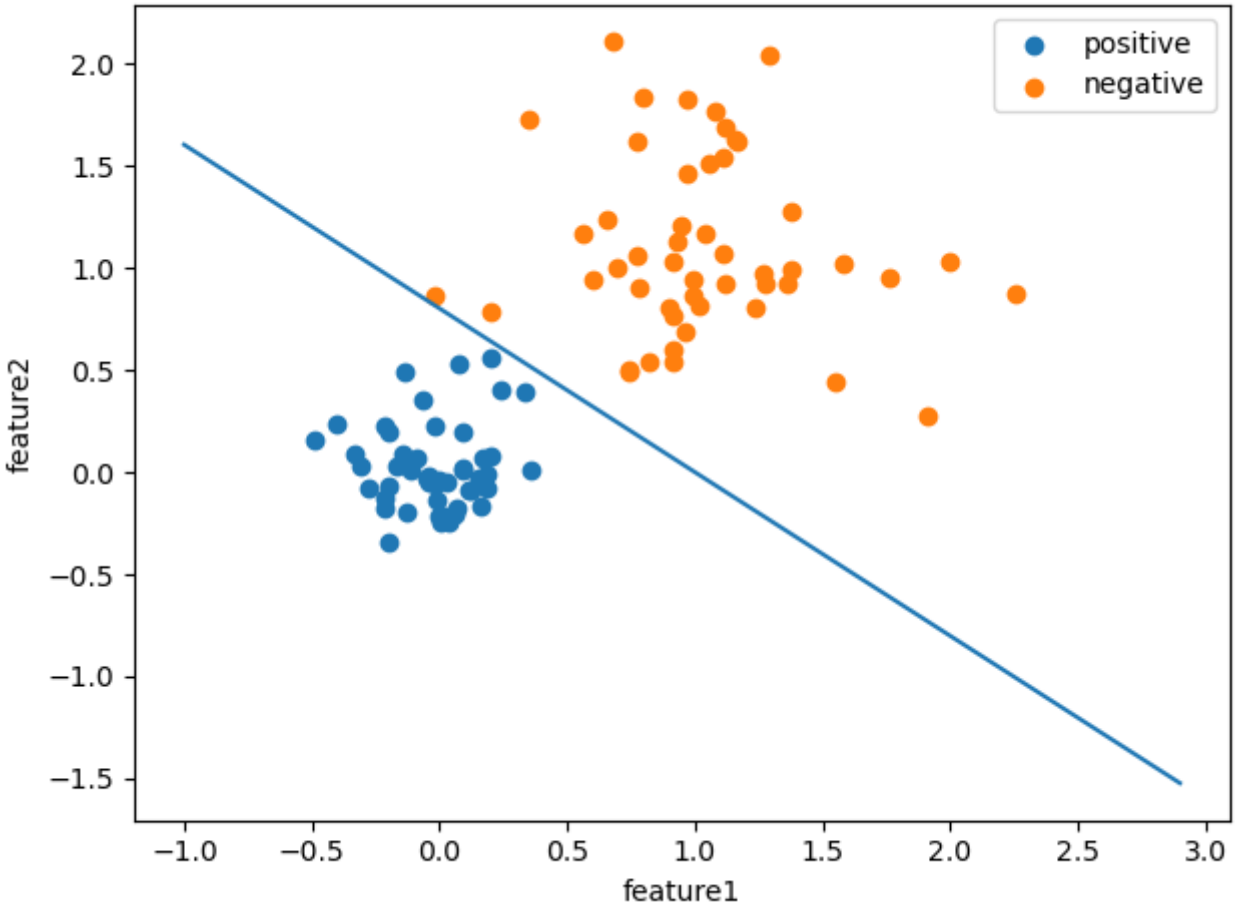
4.1.1不带正则项的梯度回归

给定步长为0.1，迭代次数从1到100，代价函数、预测准确率的相应变化和迭代100次的拟合情况如下：



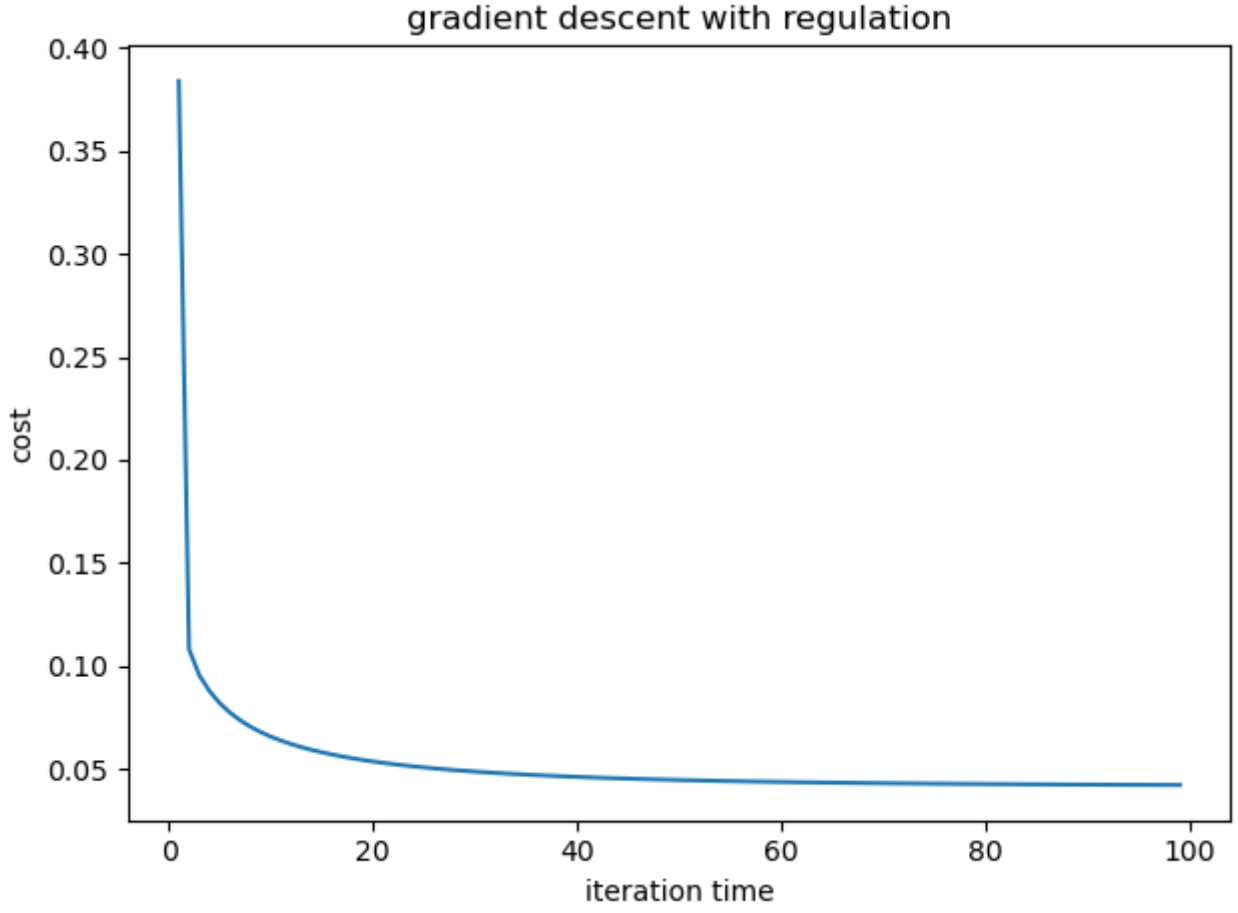

#gradient descent without regulation
cost
iteration time
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100

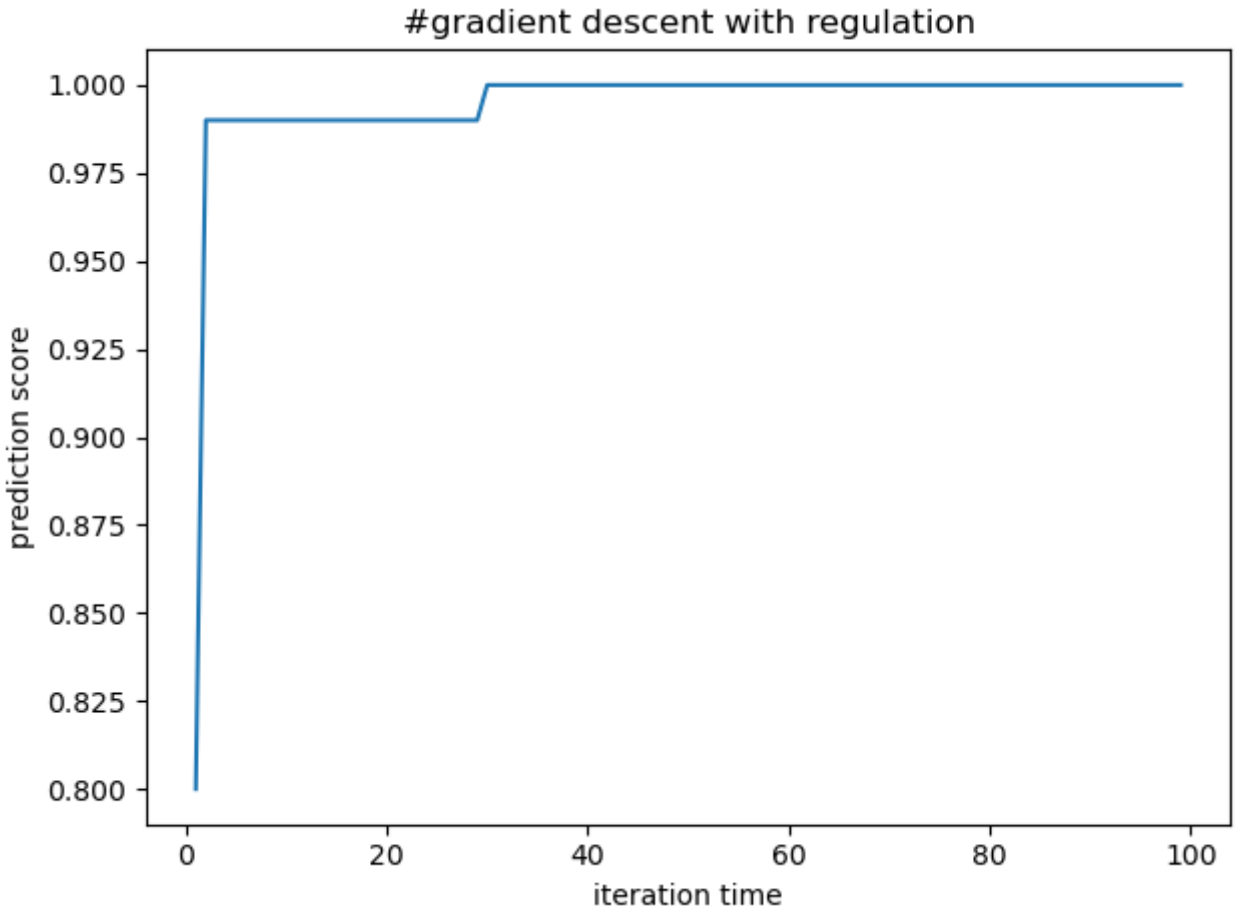

#gradient descent without regulation
prediction score
iteration time
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
20
40
60
80
100

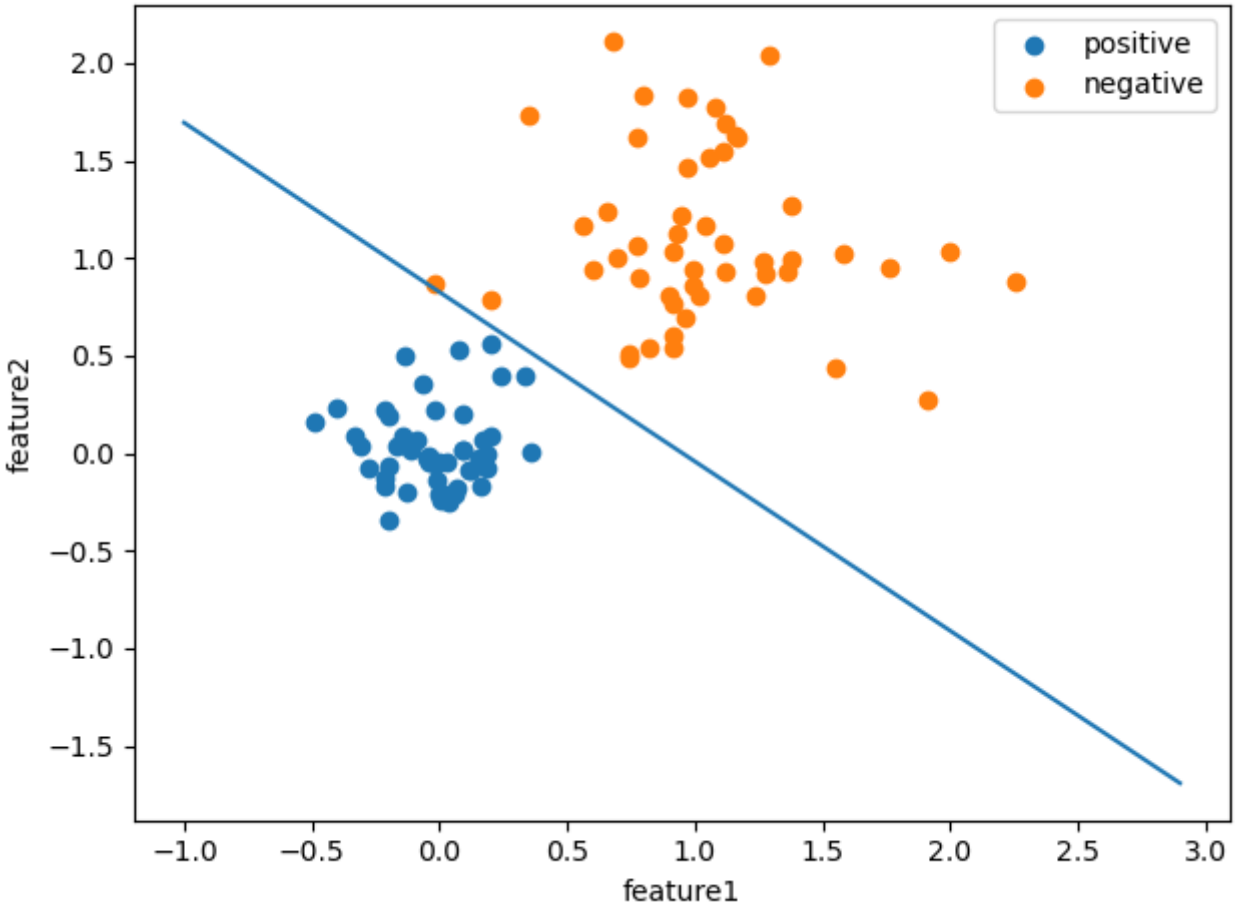


4.1.1带正则项的梯度回归

给定步长为0.1，迭代次数从1到100，代价函数、预测准确率的相应变化和迭代100次的拟合情况如下：


gradient descent with regulation
cost
iteration time
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0
20
40
60
80
100

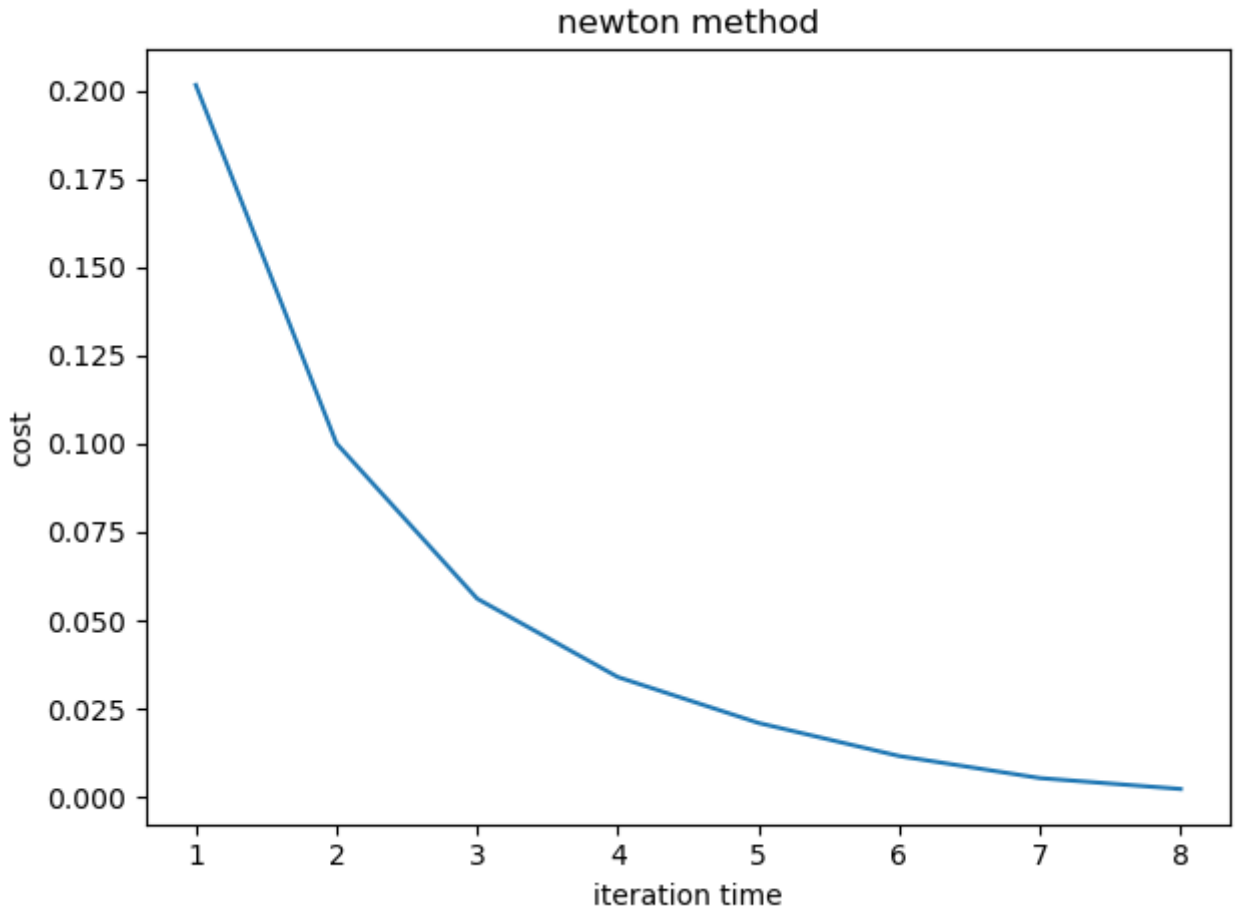

#gradient descent with regulation
prediction score
iteration time
1.000
0.975
0.950
0.925
0.900
0.875
0.850
0.825
0.800
0
20
40
60
80
100

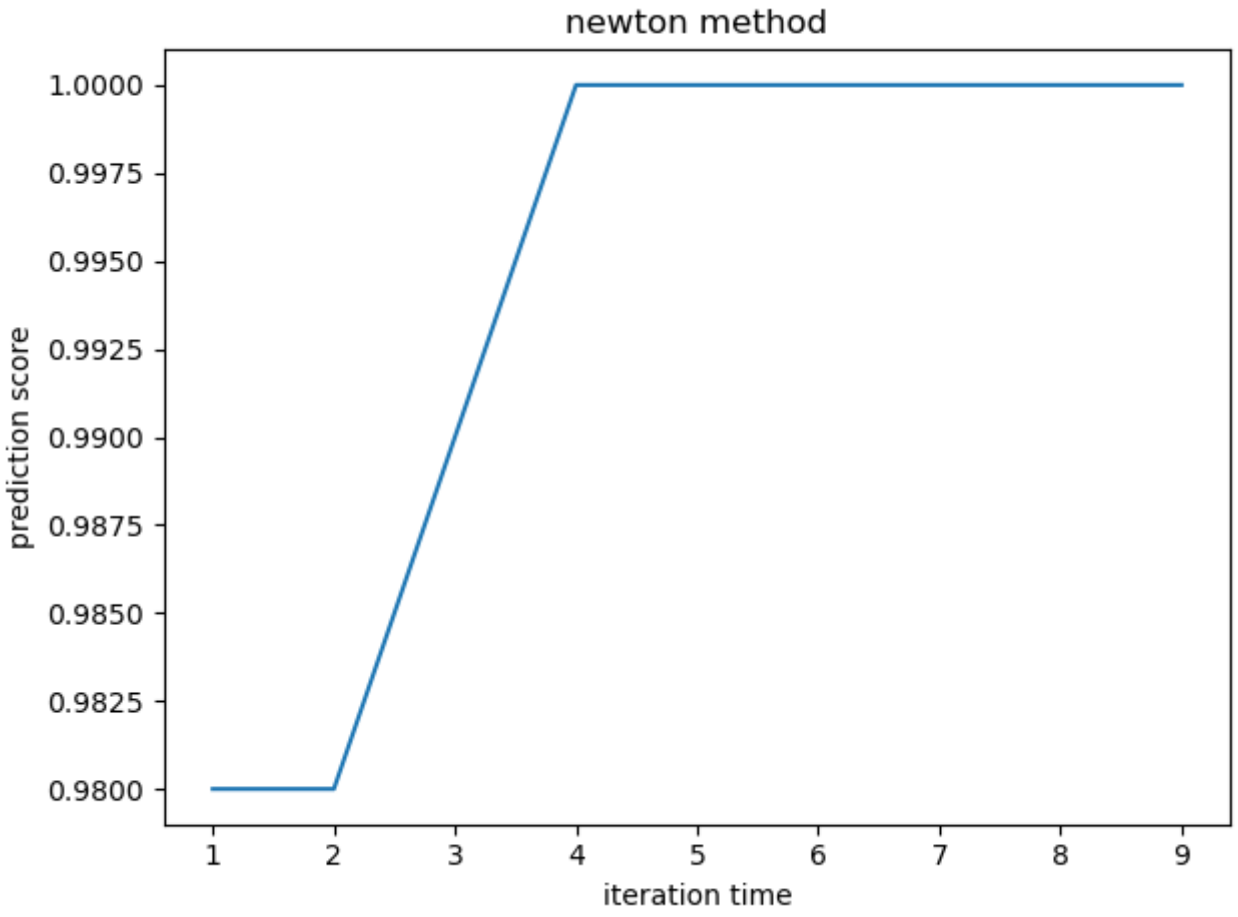


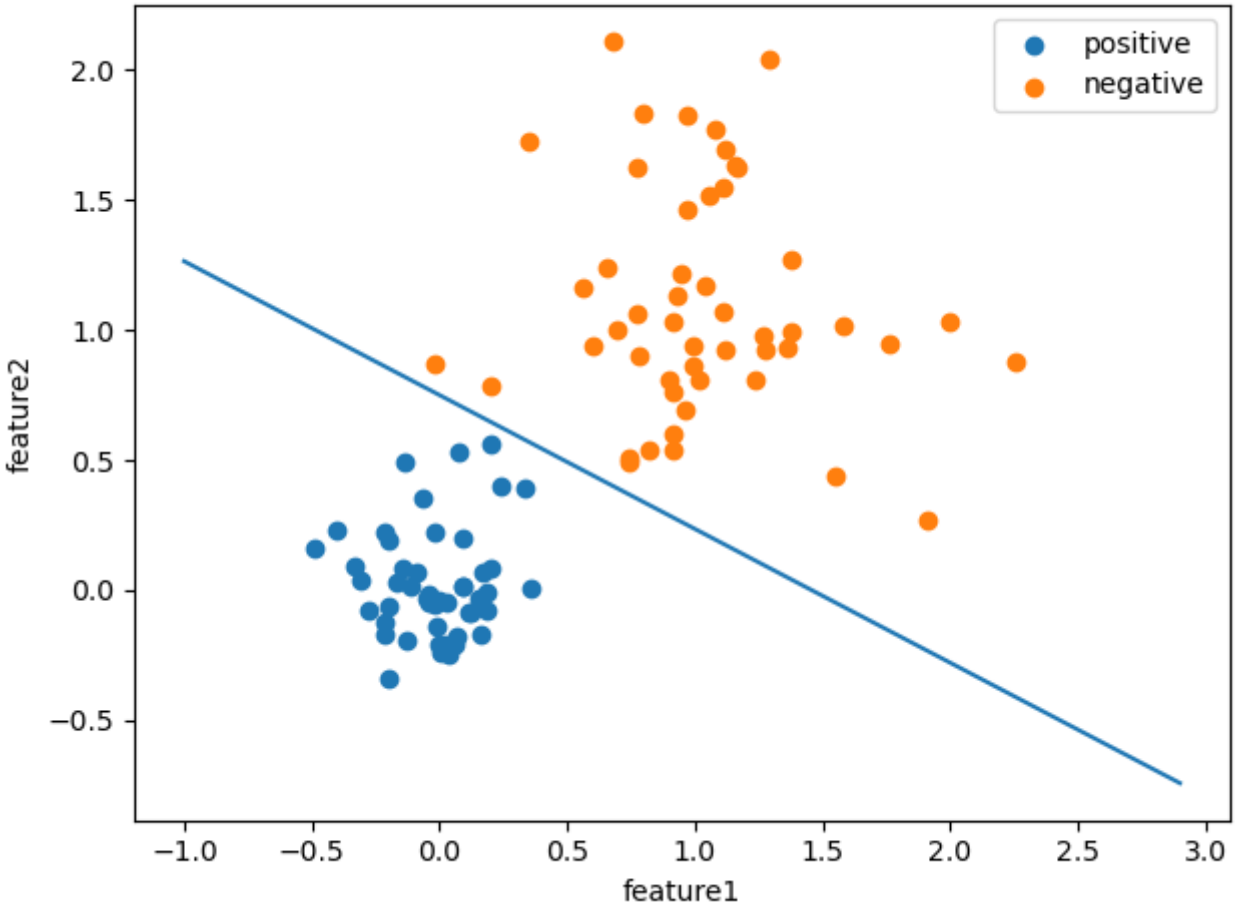
4.2 使用牛顿法求解Logistic回归

使用牛顿法对相同一组数据求解的显著优势是达到同梯度下降相同训练效果所需的迭代次数大大降低，我们发现五次左右的迭代就能实现很好的分类效果

迭代次数从1到10，代价函数、预测准确率的相应变化和迭代10次的拟合情况如下：

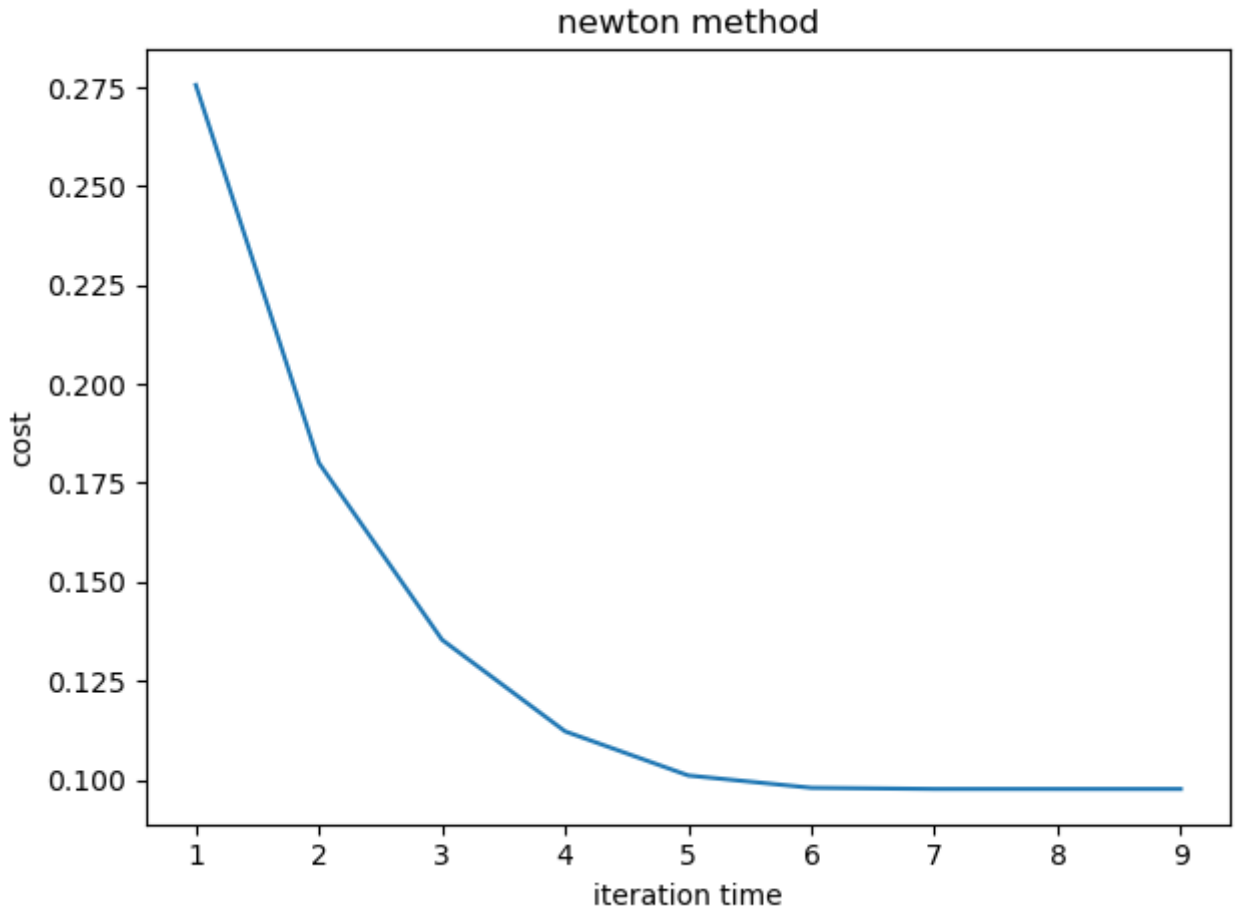

newton method
cost
iteration time
0.200
0.175
0.150
0.125
0.100
0.075
0.050
0.025
0.000
1
2
3
4
5
6
7
8

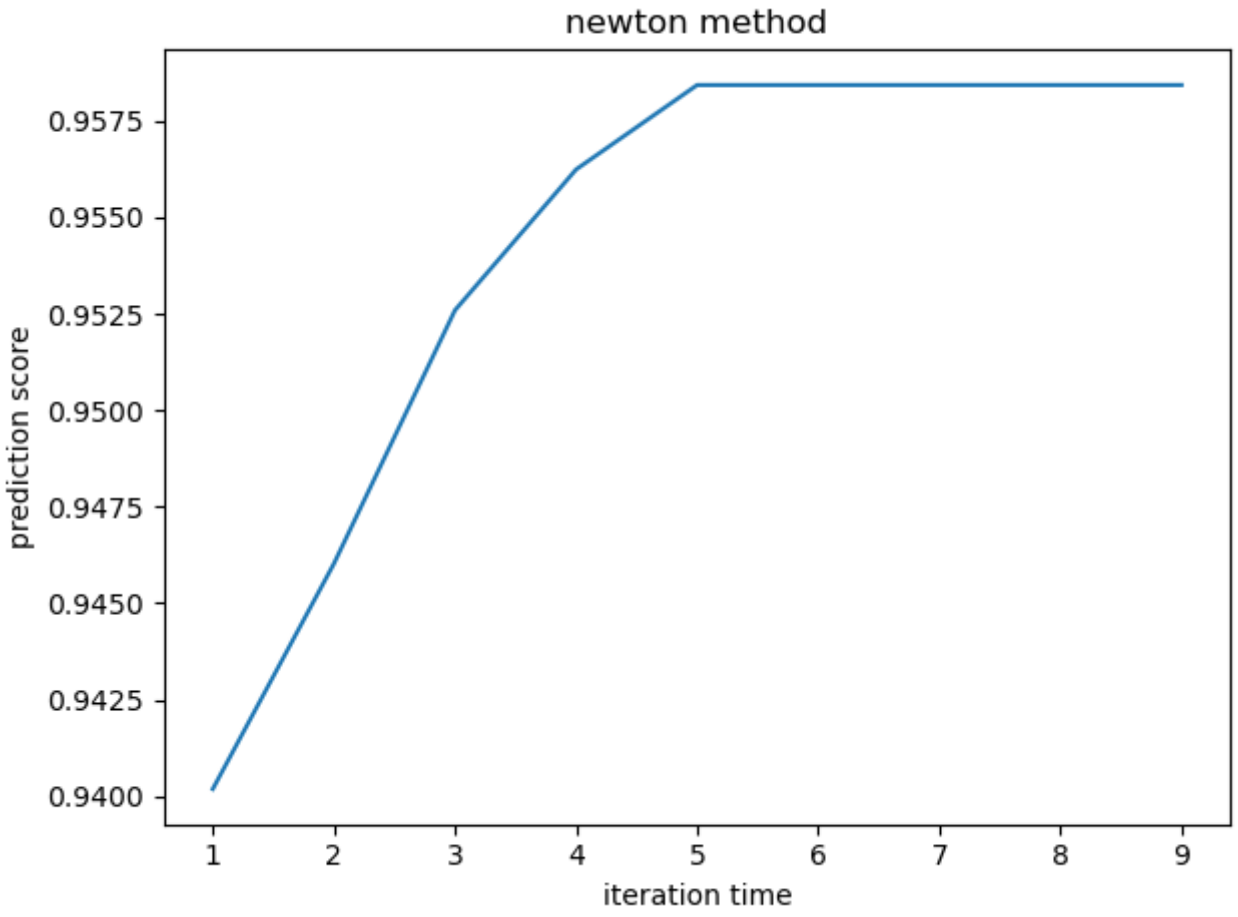

newton method
prediction score
iteration time
1.0000
0.9975
0.9950
0.9925
0.9900
0.9875
0.9850
0.9825
0.9800
1
2
3
4
5
6
7
8
9



4.3 uci网站上banknote authentication数据求解Logistic回归

使用牛顿法，迭代次数从1到10，代价函数、预测准确率的相应变化如下：


newton method
cost
iteration time
0.275
0.250
0.225
0.200
0.175
0.150
0.125
0.100
1
2
3
4
5
6
7
8
9


newton method
prediction score
iteration time
0.9575
0.9550
0.9525
0.9500
0.9475
0.9450
0.9425
0.9400
1
2
3
4
5
6
7
8
9

我们将banknote authentication数据分成训练集和测试集，其中训练集占0.8，测试集占0.2，对训练集使用牛顿法求解，然后分别对训练数据和测试数据的分类准确率如下：

```
对train data的分类准确率是0.9571558796718322
对test data训练的分类准确率是0.9087591240875912
```

可以发现,对训练集的分类准确率更好，说明存在一定过拟合

五、结论

- 相同条件下，牛顿法相比梯度下降法有明显优势，可以在达到相同效果时显著降低迭代次数。但牛顿法每轮迭代都涉及海森矩阵的求逆，计算复杂度高，尤其在高维空间几乎不可行
- 加入正则项可以防止过拟合

六、参考文献

- [1] gradient descent wikipedia
- [2] coursera吴恩达机器学习课程
- [3] Logistic回归的Loss函数

七、附录:源代码(带注释)

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

def generate_data(mu1=0, sigma1=0.2, mu2=1, sigma2=0.4, size=100, proportion=0.5):
    X = np.zeros((size, 3))
    Y = np.zeros((size, 1))
    pos_size = int(size * proportion)
    neg_size = size - pos_size
    x1 = np.random.normal(mu1, sigma1, pos_size)
    x2 = np.random.normal(mu1, sigma1, pos_size)
    x3 = np.random.normal(mu2, sigma2, neg_size)
    x4 = np.random.normal(mu2, sigma2, neg_size)
    X[:, 0] = [1 for i in range(size)]
    X[:, 1] = np.hstack((x1, x3))
    X[:, 2] = np.hstack((x2, x4))
    for i in range(pos_size):
        Y[i] = 1
    for j in range(neg_size):
        Y[pos_size + j] = 0
    return X, Y


def plot_data(X, W, Y, fit=False):
    x1 = []
    x2 = []
    x3 = []
    x4 = []
    for i in range(len(Y)):
        if Y[i][0] == 1:
            x1.append(X[i][1])
            x2.append(X[i][2])
        else:
            x3.append(X[i][1])
            x4.append(X[i][2])
    plt.scatter(x1, x2, label="positive")
    plt.scatter(x3, x4, label="negative")
    if fit:
        x = [i / 10 for i in range(-10, 30)]
        y = [- (W[0][0] + W[1][0] * i) / W[2][0] for i in x]
        plt.plot(x, y)

    plt.xlabel('feature1')
    plt.ylabel('feature2')
    plt.legend()


def sigmoid(z):
    return 1.0 / (1 + np.exp(-z))
```

```
# hypothesis = sigmoid(X @ W)
def hypothesis(X, W):
    return sigmoid(X @ W)


# Vectorized cost function     cost = (-y'log(h)-(1-y)'log(1-h))/m
def cost_function(X, W, Y):
    cost = 0
    matrix_one = np.ones((Y.shape[0], Y.shape[1]))
    h = hypothesis(X, W)
    m = len(Y)
    cost += (-Y.T @ np.log(h) - (matrix_one - Y).T @ np.log(matrix_one - h)) / m
    cost = float(cost)
    return cost


def score(X, W, Y):
    score = 0
    size = Y.shape[0]
    h = X @ W
    for i in range(size):
        prediction = 1 if h[i][0] > 0 else 0
        if prediction == Y[i][0]:
            score += 1
    return score / size

def train_test_split(X, Y, test_size=0.2):
    test_num = int(test_size * Y.shape[0])
    train_num = Y.shape[0] - test_num
    train_X = X[:train_num, :-1]
    train_Y = Y[:train_num, -1:]
    test_X = X[train_num:, :-1]
    test_Y = Y[train_num:, -1:]
    return train_X, train_Y, test_X, test_Y


def gradient_descent(X, Y, learning_rate=0.1, iteration_time=0):
    W = np.zeros((X.shape[1], 1))
    m = Y[0]
    cost = cost_function(X, W, Y)
    if iteration_time > 0:
        for i in range(iteration_time):
            z = X @ W
            h = sigmoid(z)
            W = W - (learning_rate / m) * X.T @ (h - Y)
            if cost < cost_function(X, W, Y):
                learning_rate = 1 / 2 * learning_rate
        return W
    while True:
        if cost < 1e-1:
            break
```

```
        z = X @ W
        h = sigmoid(z)
        W = W - (learning_rate / m) * X.T @ (h - Y)
        if cost < cost_function(X, W, Y):
            learning_rate = 1 / 2 * learning_rate
    return W


def gradient_descent_with_regulation(X, Y, learning_rate=0.1, Lambda=0.1, iteration_time=0):
    W = np.zeros((X.shape[1], 1))
    m = Y[0]
    cost = cost_function(X, W, Y)
    if iteration_time > 0:
        for i in range(iteration_time):
            z = X @ W
            h = sigmoid(z)
            W = W - (learning_rate / m) * (X.T @ (h - Y) + Lambda / m * W)
            if cost < cost_function(X, W, Y):
                learning_rate = 1 / 2 * learning_rate
        return W
    while True:
        if cost < 1e-1:
            break
        z = X @ W
        h = sigmoid(z)
        W = W - (learning_rate / m) * (X.T @ (h - Y) + Lambda / m * W)
        if cost < cost_function(X, W, Y):
            learning_rate = 1 / 2 * learning_rate
    return W


def newton(X, Y, iteration_time):
    W = np.zeros((X.shape[1], 1))
    n, m = X.shape
    for i in range(iteration_time):
        h = hypothesis(X, W)
        U = 1 / n * X.T @ (h - Y)  # (m,1)
        H = 1 / n * np.dot(np.dot(np.dot(X.T, np.diag(h.reshape(n))), np.diag(1 - h.reshape(n)))
        W = W - np.linalg.inv(H) @ U  # (m,1)
    return W


def newton_with_regulation(X, Y, iteration_time, Lambda):
    W = np.zeros((X.shape[1], 1))
    n, m = X.shape
    for i in range(iteration_time):
        h = hypothesis(X, W)
        U = 1 / n * (X.T @ (h - Y) + Lambda * W)  # (m,1)
        H = 1 / n * np.dot(np.dot(np.dot(X.T, np.diag(h.reshape(n))), np.diag(1 - h.reshape(n)))
        W = W - np.linalg.inv(H) @ U - Lambda / n * W # (m,1)
    return W
```

```
X, Y = generate_data()
plot_W = np.ones((X.shape[1], 1))
plot_data(X, plot_W, Y)
plt.show()

# gradient descent without regulation
time = [i for i in range(1, 100)]
cost = []
prediction = []
for iteration_time in range(1, 100):
    W = gradient_descent(X, Y, iteration_time=iteration_time)
    cost.append(cost_function(X, W, Y))
    prediction.append(score(X, W, Y))
# plot cost with iteration time
plt.plot(time, cost)
plt.title('#gradient descent without regulation')
plt.xlabel('iteration time')
plt.ylabel('cost')
plt.show()
# plot score with iteration time
plt.plot(time, prediction)
plt.title('#gradient descent without regulation')
plt.xlabel('iteration time')
plt.ylabel('prediction score')
plt.show()
# fit plot
plot_data(X, W, Y, fit=True)
plt.show()

# gradient descent with regulation
time = [i for i in range(1, 100)]
cost = []
prediction = []
for iteration_time in range(1, 100):
    W = gradient_descent_with_regulation(X, Y, iteration_time=iteration_time)
    cost.append(cost_function(X, W, Y))
    prediction.append(score(X, W, Y))
plt.plot(time, cost)
plt.title('gradient descent with regulation')
plt.xlabel('iteration time')
plt.ylabel('cost')
plt.show()
# plot score with iteration time
plt.plot(time, prediction)
plt.title('#gradient descent with regulation')
plt.xlabel('iteration time')
plt.ylabel('prediction score')
plt.show()
# fit plot
```

```
plot_data(X, W, Y, fit=True)
plt.show()


# newton method
time = [i for i in range(1, 10)]
cost = []
prediction = []
for iteration_time in range(1, 10):
    W = newton(X, Y, iteration_time=iteration_time)
    cost.append(cost_function(X, W, Y))
    prediction.append(score(X, W, Y))
plt.plot(time, cost)
plt.title('newton method')
plt.xlabel('iteration time')
plt.ylabel('cost')
plt.show()
# plot score with iteration time
plt.plot(time, prediction)
plt.title('newton method')
plt.xlabel('iteration time')
plt.ylabel('prediction score')
plt.show()
# fit plot
plot_data(X, W, Y, fit=True)
plt.show()

#read data from uci
data = pd.read_csv('data_banknote_authentication.txt')
x = data.iloc[:, :-1]
y = data.iloc[:, -1:]
X = np.array(x)
Y = np.array(y)


# newton method
time = [i for i in range(1, 10)]
cost = []
prediction = []
for iteration_time in range(1, 10):
    W = newton(X, Y, iteration_time=iteration_time)
    cost.append(cost_function(X, W, Y))
    prediction.append(score(X, W, Y))
plt.plot(time, cost)
plt.title('newton method')
plt.xlabel('iteration time')
plt.ylabel('cost')
plt.show()
# plot score with iteration time
plt.plot(time, prediction)
plt.title('newton method')
```

```
plt.xlabel('iteration time')
plt.ylabel('prediction score')
plt.show()


train_X, train_Y, test_X, test_Y = train_test_split(X, Y, test_size=0.2)
train_W = newton(train_X, train_Y, iteration_time=10)
print(f'对train data的分类准确率是{score(train_X, train_W, train_Y)}')
print(f'对test data训练的分类准确率是{score(test_X, train_W, test_Y)}')

train_W = newton_with_regulation(train_X, train_Y, iteration_time=10,Lambda=0.1)
print(f'对train data的分类准确率是{score(train_X, train_W, train_Y)}')
print(f'对test data训练的分类准确率是{score(test_X, train_W, test_Y)}')
```