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：1190201008
姓名： 周凡

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

1、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。

2、实验环境

Python3.7, Windows10, Spyder, Anaconda 3

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

考虑一个二分类问题 $f: X \to Y$，其中 $X = <X_1, X_2, ..., X_n>, Y \in \{0,1\}$，且假设在给定 Y 前提下所有 X_i 条件独立，满足 $P(X_i|Y=y_k) \sim N(\mu_{ik}, \sigma_i)$, $P(Y) \sim B(\pi)$，则：

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

利用全概率公式对 P(X) 展开得：

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

分子分母同时除以 $P(Y=0)\,P(X|Y=0)$，并代入 $P(Y=0)=1-\pi$，$P(Y=1)=\pi$ 得：

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

根据朴素贝叶斯的假设，所有 X_i 条件独立，可以将向量各维展开得到：

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

又由于 X 向量各维度均服从高斯分布，有 $P(X_{\mathrm{i}}|\mathrm{Y=y}_k)=\frac{1}{\sigma_i\sqrt{2\pi}}\exp(\frac{-(x-\mu_{ik})^2}{2\sigma_i^2})$，

代入上式得：

$$P(Y=0|\mathrm{X})=\frac{1}{1+\exp(lin(\frac{\pi}{1-\pi})+\sum_i(\frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}X_i+\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}))}$$

可将该式改写为向量形式：

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

其中 $\mathrm{w}_0=lin(\frac{\pi}{1-\pi})+\sum_i\frac{\mu_{i0}^2-\mu_{i1}^2}{2\sigma_i^2}, w_i=\frac{\mu_{i1}-\mu_{i0}}{\sigma_i^2}, i>0, \mathbf{X}=\begin{bmatrix}1\\X_1\\X_2\\...\\X_n\end{bmatrix}$

根据上式可得：

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

在给定数据集 $\{<X^1,Y^1>,...,<X^l,Y^l>\}$ 情况下，对 W 进行最大似然估计：

$$\mathbf{w}_{MCLE}=\arg\max_{\mathbf{w}}\prod_l P(Y^l\mid X^l,\mathbf{w})$$

取对数得：

$$\mathrm{l}(\mathbf{w})=\ln\prod_l P(Y^l\mid X^l,\mathbf{w})=\sum_l P(Y^l\mid X^l,\mathbf{w})=\sum_l(Y^l\mathbf{w}^T X-\ln(1+\exp(\mathbf{w}^T X)))$$

将求最大值转化为求最小值：

$$L(\mathbf{w})=\sum_l(-Y^l\mathbf{w}^T X+\ln(1+\exp(\mathbf{w}^T X)))$$

即最终将估计 w 的问题转化为求上式最大值的解，可以利用梯度下降法、牛顿法等方法进行求解。同时，为避免出现过拟合的情况，同 Lab1 一样加入惩罚项得：

$$L_1(\mathbf{w})=\sum_l(-Y^l\mathbf{w}^T X+\ln(1+\exp(\mathbf{w}^T X)))+\frac{\lambda}{2}\mathbf{w}^T\mathbf{w}$$

2. 算法实现

2.1 梯度下降法

此实验使用的梯度下降法与 Lab1 基本一致，但需要修改一阶导数（此处为 L(w)的一阶导数，不包含惩罚项）和 w 更新（其中考虑了惩罚项带来的影响），对于第 i 次迭代：

$$w^i = (1-\alpha\lambda)w^{i+1} - \alpha\frac{\partial}{\partial w}L(w^{i+1})$$

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{l} X_i\left(\frac{exp(w^T X)}{1+\exp(w^T X)} - Y_i\right)$$

在实验中发现，在计算似然函数时，$\ln(1+\exp(\mathbf{w}^T X))$ 中的指数项可能会发生溢出，于是在计算似然函数时添加了判断，若指数 $\mathbf{w}^T X$ 值较大，则进行如下近似处理：

$$\ln(1+\exp(\mathbf{w}^T X)) \approx \mathbf{w}^T X$$

近似代码如下：

```
p[i] = np.dot(w, self.X[i].T)
# 当p[i]足够大时，进行近似处理防止溢出
if(p[i] >= np.log(sys.float_info.max/2)):
    sum += p[i]
else:
    sum += np.log(1+np.exp(p[i]))
```

此外，为了提高效率，对 Lab1 梯度下降的学习率的选择进行了优化。当某次迭代后损失函数增大了，则说明学习率偏大，对学习率进行减半处理；若在某学习率下连续迭代了 k 次（k 为一个较大的数，如 10000），损失函数仍未收敛，则说明学习率较小，对其进行翻倍提高学习速率。

学习率调整代码如下：

```
if OldLoss < NewLoss:
    self.step *= 0.5
    j = 0
if j>10000:
    self.step *= 2
    j = 0
```

2.2 牛顿法

牛顿法的基本原理是对目标函数进行二阶泰勒展开，即用一条抛物线对其进行拟合，则相比抛物线与该函数图像的切点，抛物线的顶点应当更加接近该函数的极值点，于是可通过重复求每次拟合得到的二次曲线的极值点来不断逼近目标极值点。公式推导如下：

$$f(x) \approx f(x_0) + f'(x_0)(x-x_0) + \frac{1}{2} f''(x_0)(x-x_0)^2$$

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

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

即在第 i 次迭代中：

$$w^i = w^{i-1} - \left(\frac{\partial^2 L}{\partial w \partial w^T} \right)^{-1} \frac{\partial L}{\partial w}$$

其中：

$$\frac{\partial L}{\partial w} = \sum_{i=1}^{l} X_i \left(\frac{exp(w^T X)}{1+\exp(w^T X)} - Y_i \right)$$

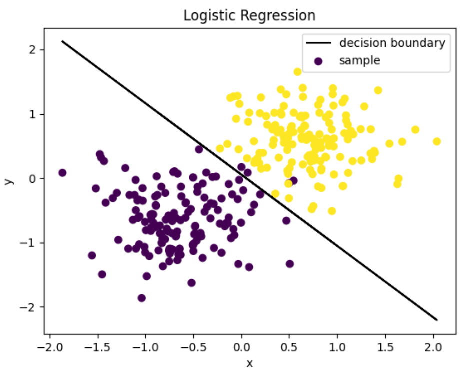
$$\frac{\partial^2 L}{\partial w \partial w^T} = \sum_{i=1}^{l} \left(X_i X_i^T \frac{exp(w^T X)}{\left(1+\exp(w^T X)\right)^2} \right)$$

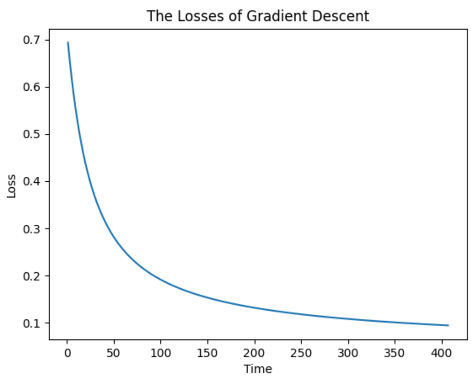
四、 实验结果与分析

1. 自己生成的数据

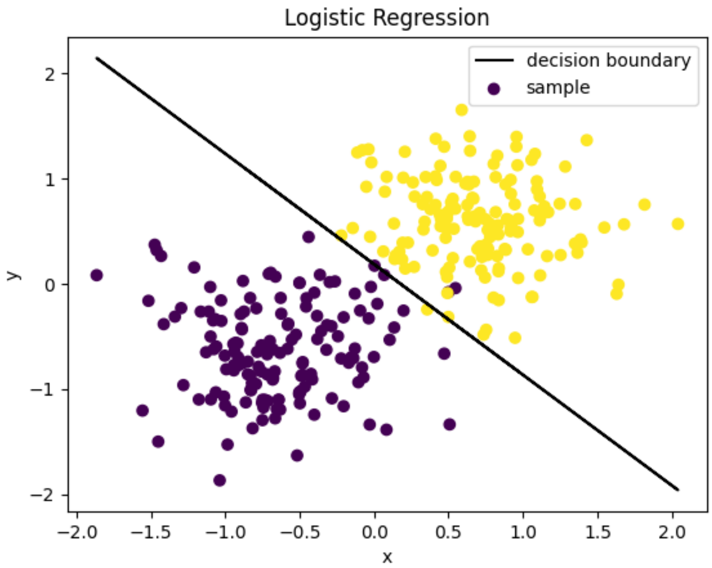
1.1 满足朴素贝叶斯假设，λ=0.0001，样本量 300，各维方差 0.2

1.1.1 梯度下降法



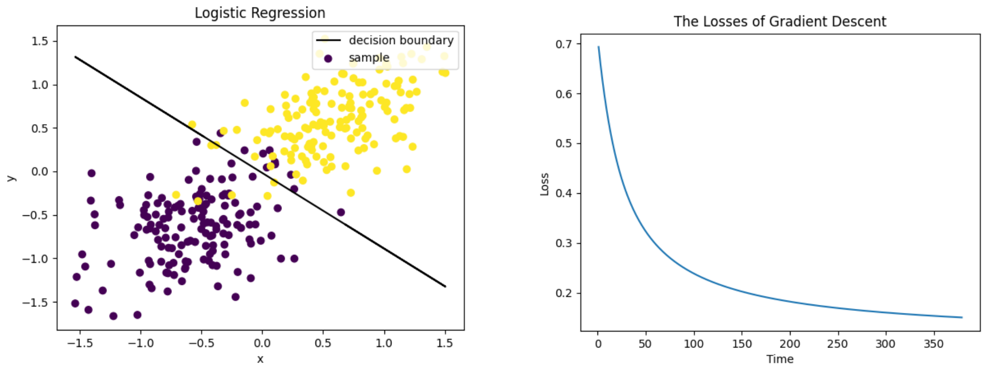


1.1.2 牛顿法

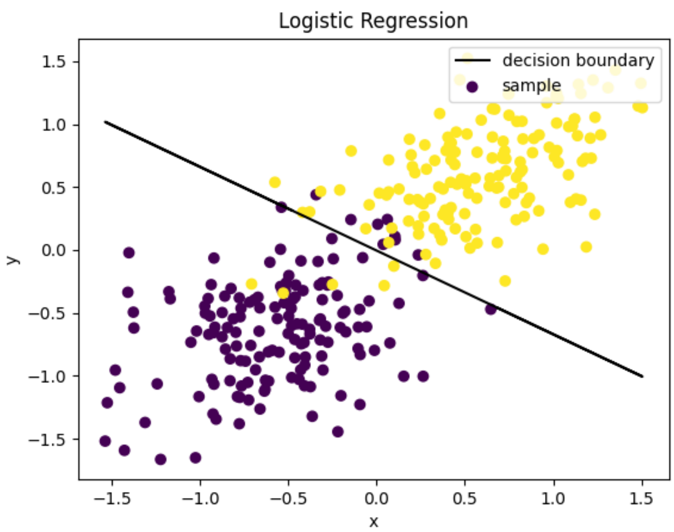


1.2 不满足朴素贝叶斯假设，协方差 cov=0.1，其他参数同上

1.1.1 梯度下降法



1.1.2 牛顿法



1.3 分析

由上面两个实验发现，对于满足朴素贝叶斯假设的数据集，该算法的分类效果很好，但即使不满足朴素贝叶斯，该算法的效果仍然非常不错，仅有极少量的点未被正确分类。

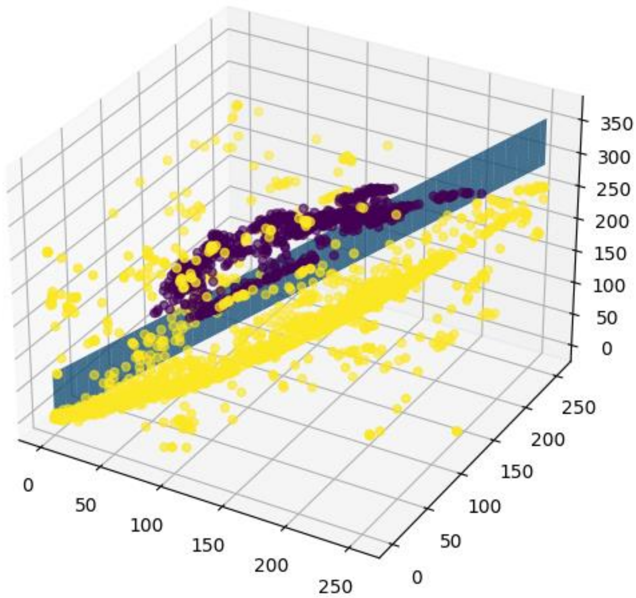
此外，实验还发现牛顿法下降速度明显由于梯度下降法。

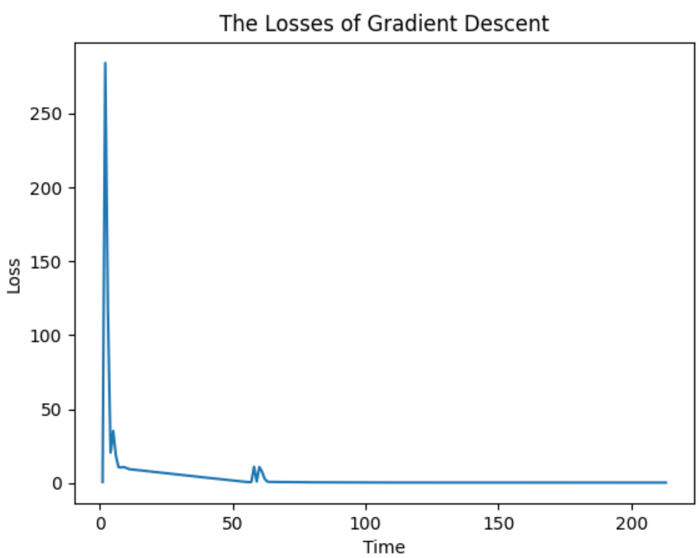
2. UCI Skin_NonSkin dataset

正则项 λ=0.0001，数据维度为 3

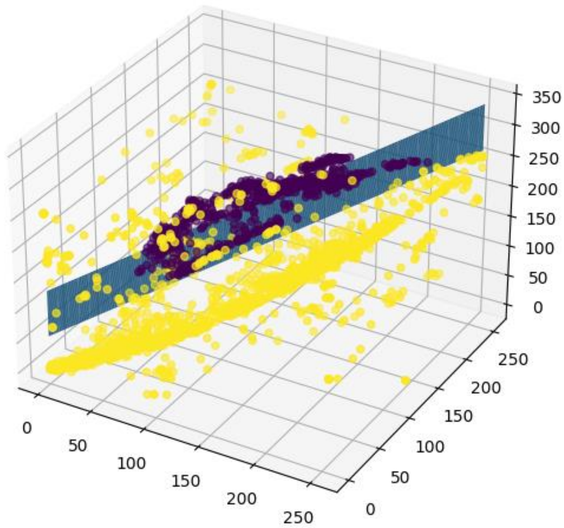
由于该数据集数据量较多，故仅选择了总集的 33%进行实验。

2.1 梯度下降法





2.2 牛顿法



2.3 准确率

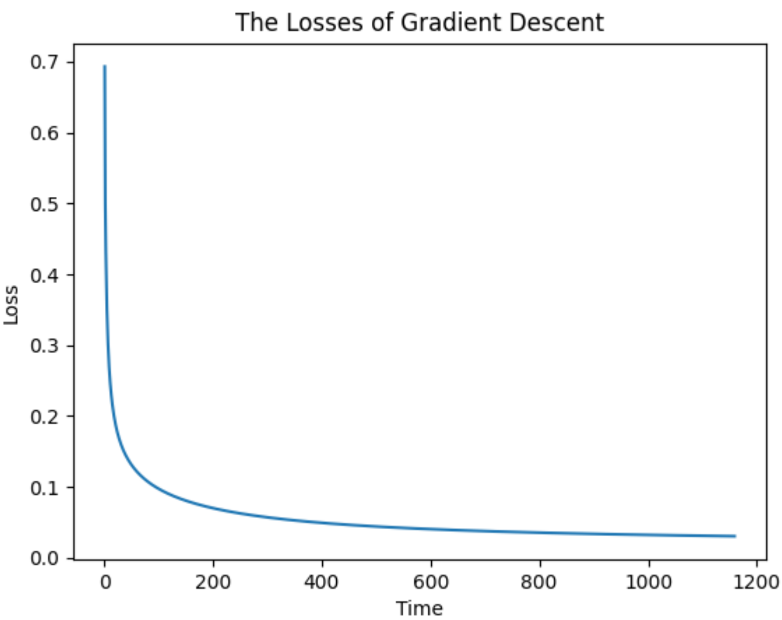
```
skin_gradientdescent:
accuracy:  0.910959854401815
skin_newton:
accuracy:  0.9205983889528193
```

通过图像观察发现该数据集的分布其实并不适合用线性分类来求解，所以最终的准确率并不是特别高，仅达到 0.9。

3. UCI data_banknote_authentication dataset

正则项 λ=0.0001，数据维度为 4

2.1 梯度下降法损失函数



2.2 准确率

```
banknote_gradientdescent:
accuracy:  0.9868804664723032
banknote_newton:
accuracy:  0.9868804664723032
```

相比上一组数据集，该组数据集的分类结果显然更优。

五、 结论

1、 即使所处理的数据集不满足朴素贝叶斯假设，逻辑回归仍然可以很好地实现准确的分类。
2、 梯度下降法和牛顿法都可以得到很好的结果，不过牛顿法的下降速度明显快于梯度下降法。
3、 逻辑回归可以很好地解决线性分类问题，但是当遇到非线性可分的数据则表现较差。

六、 参考文献

- 周志华 著. 机器学习, 北京: 清华大学出版社, 2016.1
- CSDN 牛顿法: https://blog.csdn.net/itplus/article/details/21896453
- Banknote dataset: http://archive.ics.uci.edu/ml/datasets/banknote+authentication
- Skin_Nonskin dataset: https://archive.ics.uci.edu/ml/datasets/skin+segmentation

七、附录：源代码（带注释）

- 主程序见 logistics_regression.py

梯度下降法见 gradient_descent.py

- 牛顿法见 newton.py
- Skin 数据集见 Skin_NonSkin.txt
- Banknote 数据集见 data_banknote_authentication.csv

logistics_regression.py：

```
import numpy as np

import matplotlib.pyplot as plt
```

```
import pandas as pd

import gradient_descent

import newton

from mpl_toolkits.mplot3d import Axes3D



def get_data(SampleAmount, naive):

    boundary = np.ceil(SampleAmount/2).astype(np.int32)

    lam = 0.2

    cov = 0.1

    X_mean0 = [-0.6, -0.6]

    X_mean1 = [0.6, 0.6]

    X = np.zeros((SampleAmount, 2))

    train_X = np.ones((SampleAmount, 3))

    Y = np.zeros(SampleAmount)

    # 满足朴素贝叶斯

    if naive:

        X[:boundary, :] = np.random.multivariate_normal(

            X_mean0, [[lam, 0], [0, lam]], size=boundary)

        X[boundary:, :] = np.random.multivariate_normal(

            X_mean1, [[lam, 0], [0, lam]],

size=SampleAmount-boundary)
```

```
        Y[:boundary] = 0
        Y[boundary:] = 1
    # 不满足朴素贝叶斯
    else:
        X[:boundary, :] = np.random.multivariate_normal(
            X_mean0, [[lam, cov], [cov, lam]],
size=boundary)
        X[boundary:, :] = np.random.multivariate_normal(
            X_mean1, [[lam, cov], [cov, lam]],
size=SampleAmount-boundary)
        Y[:boundary] = 0
        Y[boundary:] = 1
    train_X[:, 1] = X[:, 0]
    train_X[:, 2] = X[:, 1]
    return X, Y, train_X


# 画二维图
def graph(X, Y, w):
    plt.scatter(X[:, 0], X[:, 1], c=Y, label="sample")
    dimension = np.size(w, axis=1)
    w = w.reshape(dimension)
    coeff = -(w/w[dimension-1])[0:dimension-1]
```

```python
    decisionboundary = np.poly1d(coeff[::-1])
    result_Y = decisionboundary(X[:, 0])
    plt.plot(X[:, 0], result_Y, linestyle='-', color='k',
             marker='', label="decision boundary")
    plt.xlabel("x")
    plt.ylabel("y")
    plt.title("Logistic Regression")
    plt.legend(loc="upper right")
    plt.show()
    return 0


# 画三维图
def graph_3D(X, Y, w):
    fig = plt.figure()
    ax = Axes3D(fig)
    ax.scatter(X[:, 0], X[:, 1], X[:, 2], c=Y)
    dimension = np.size(w, axis=1)
    w = w.reshape(dimension)
    coeff = -(w/w[dimension-1])[0:dimension-1]
    x = np.arange(np.min(X[:, 0]), np.max(X[:, 0])+1, 1)
    y = np.arange(np.min(X[:, 1]), np.max(X[:, 1])+1, 1)
    xp, yp = np.meshgrid(x, y)
```

```
    z = coeff[0] + coeff[1]*xp + coeff[2]*yp

    ax.plot_surface(x, y, z)

    plt.show()

    return 0


# 画损失函数图像

def graph_loss(loss, counter):

    plt.plot(counter, loss)

    plt.xlabel("Time")

    plt.ylabel("Loss")

    plt.title("The Losses of Gradient Descent")

    plt.show()

    return 0


# 计算准确性

def cal_accuracy(test_x, test_y, test_size, dimension, w):

    label = np.ones(test_size)

    correct_count = 0

    xt = np.ones((test_size, dimension+1))

    for i in range(dimension):

        xt[:, i+1] = test_x[:, i]

    for i in range(test_size):
```

```
        if np.dot(w, xt[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("accuracy: ", correct_rate)
    return correct_rate


# 模拟实验
def exp(SampleAmount, w, lamda, step, epsilon, naive):
    X, Y, train_X = get_data(SampleAmount, naive)
    gd = gradient_descent.GradientDescent(
        train_X, Y, w, SampleAmount, np.size(train_X,
axis=1), lamda, step, epsilon)
    w1, loss, counter = gd.gradient_descent()
    nt = newton.Newton(train_X, Y, w, SampleAmount,
np.size(train_X, axis=1), lamda, step, epsilon)
    w2 = nt.newton()
    graph(X, Y, w1)
    graph_loss(loss, counter)
```

```
    graph(X, Y, w2)
    return 0


# 处理 UCI 数据
def generate_UCI_data(train_rate, step, load_data):
    np.random.shuffle(load_data)  # 打乱数据集以便选出训练集
    load_data_size = np.size(load_data, axis=0)
    train_data = load_data[:int(load_data_size*train_rate), :]
    test_data = load_data[int(load_data_size*train_rate):, :]
    dimension = np.size(load_data, axis=1) - 1
    # 训练集
    train_x = train_data[:, 0:dimension]
    train_x = train_x[::step]
    train_size = np.size(train_x, axis=0)
    train_y = train_data[:, dimension:dimension+1]
    train_y = train_y[::step]
    train_y = train_y.reshape(train_size)
    # 测试集
    test_size = np.size(test_data, axis=0)
```

```
    test_x = test_data[:, 0:dimension]
    test_y = test_data[:,
dimension:dimension+1].reshape(test_size)
    return train_x, train_y, train_size, test_x, test_y,
test_size, dimension


# skin_nonskin experiment
def skin_exp(w, lamda, step, epsilon):
    load_data = np.loadtxt("./Skin_NonSkin.txt",
dtype=np.int32)
    load_data[:, 3] = load_data[:, 3] - 1
    x, train_y, train_size, test_x, test_y, test_size,
dimension = generate_UCI_data(
        0.5, 30, load_data)
    train_x = np.ones((train_size, dimension+1))
    for i in range(dimension):
        train_x[:, i+1] = x[:, i]
    gd = gradient_descent.GradientDescent(
        train_x, train_y, w, train_size, dimension+1, lamda,
step, epsilon)
    w1, loss, counter = gd.gradient_descent()
    graph_3D(x, train_y, w1)
```

```
    graph_loss(loss, counter)

    nt = newton.Newton(

        train_x, train_y, w, train_size, dimension+1, lamda,
step, epsilon)

    w2 = nt.newton()

    graph_3D(x, train_y, w2)

    # 计算准确率

    print("skin_gradientdescent: ")

    cal_accuracy(test_x, test_y, test_size, dimension, w1)

    print("skin_newton: ")

    cal_accuracy(test_x, test_y, test_size, dimension, w2)

    return 0


# banknote experiment

def banknote_exp(w, lamda, step, epsilon):

    load_data =
pd.read_csv("./data_banknote_authentication.csv")

    data = np.array(load_data)

    x, train_y, train_size, test_x, test_y, test_size,
dimension = generate_UCI_data(

        0.5, 1, data)

    train_x = np.ones((train_size, dimension+1))
```

```
    for i in range(dimension):
        train_x[:, i+1] = x[:, i]
    gd = gradient_descent.GradientDescent(
        train_x, train_y, w, train_size, dimension+1, lamda,
step, epsilon)
    w1, loss, counter = gd.gradient_descent()
    graph_loss(loss, counter)
    nt = newton.Newton(
        train_x, train_y, w, train_size, dimension+1, lamda,
step, epsilon)
    w2 = nt.newton()
    # 计算准确率
    print("banknote_gradientdescent: ")
    cal_accuracy(test_x, test_y, test_size, dimension, w1)
    print("banknote_newton: ")
    cal_accuracy(test_x, test_y, test_size, dimension, w2)
    return 0


SampleAmount = 300
lamda = 0.0001
step = 0.1
naive = False
```

```
w1 = np.zeros((1, 3))
w2 = np.zeros((1, 4))
w3 = np.ones((1, 5))*0
exp(SampleAmount, w1, lamda, step, 0.0001, True)
exp(SampleAmount, w1, lamda, step, 0.0001, False)
skin_exp(w2, lamda, step, 0.00001)
banknote_exp(w3, lamda, step, 0.00001)
```

gradient_descent.py：

```
import numpy as np
import sys


class GradientDescent(object):
    def __init__(self, X, Y, w, m, n, lamda, step, epsilon):
        self.X = X
        self.Y = Y
        self.w = w
        self.m = m
        self.n = n
        self.lamda = lamda
        self.step = step
        self.epsilon = epsilon

```

```
    def sigmoid_func(self, x):

        return 1/(1+np.exp(-x))


    def likelihood_func(self, w):

        amount = np.size(self.X, axis=0)

        p = np.zeros((amount, 1))

        sum = 0

        for i in range(amount):

            p[i] = np.dot(w, self.X[i].T)

            # 当p[i]足够大时，进行近似处理防止溢出

            if(p[i] >= np.log(sys.float_info.max/2)):

                sum += p[i]

            else:

                sum += np.log(1+np.exp(p[i]))

        return np.dot(self.Y, p) - sum


    def partial_derivative(self, w):

        return np.dot(self.sigmoid_func(np.dot(w,
self.X.T))-self.Y, self.X)


    # 梯度下降法,m 为样本数,n 为参数个数，lamda 为惩罚项系数

    # step 为步长，epsilon 为迭代误差，dimension 为X 样本维度
```

```
    def gradient_descent(self):
        w = self.w
        # 记录损失函数值的变化情况
        losslist = []
        counterlist = []
        i = 1
        j = 0
        while 1:
            OldLoss = -self.likelihood_func(w)/self.m
            gradient = self.partial_derivative(w)/self.m
            losslist.append(OldLoss)
            counterlist.append(i)
            w = w - self.step*self.lamda*w -
self.step*gradient
            NewLoss = -self.likelihood_func(w)/self.m
            i = i+1
            j = j+1
            # gnorm = np.dot(gradient, gradient.T)
            # print(OldLoss-NewLoss)
            # print(OldLoss)
            # 若损失函数收敛则结束循环
            if abs(OldLoss-NewLoss) < self.epsilon:
```

```
                losslist.append(NewLoss)

                counterlist.append(i)

                break

            else:

                if OldLoss < NewLoss:

                    self.step *= 0.5

                    j = 0

                if j>10000:

                    self.step *= 2

                    j = 0

        return w, losslist, counterlist
```

newton.py:

```
import numpy as np



class Newton(object):

    def __init__(self, X, Y, w, m, n, lamda, step, epsilon):

        self.X = X

        self.Y = Y

        self.w = w

        self.m = m

        self.n = n

        self.lamda = lamda
```

```
        self.step = step

        self.epsilon = epsilon


    def sigmoid_func(self, x):

        return 1/(1+np.exp(-x))


    def partial_derivative(self, w):

        return np.dot(self.sigmoid_func(np.dot(w,
self.X.T))-self.Y, self.X) + self.lamda*w


    def second_derivative(self, w):

        ans = np.eye(self.n) * self.lamda

        for i in range(self.m):

            temp = self.sigmoid_func(np.dot(w,
self.X[i].T))

            ans += self.X[i] * np.transpose([self.X[i]]) *
temp * (1 - temp)

        return ans


    def newton(self):

        w = self.w

        while 1:
```

```
            gradient = self.partial_derivative(w)

            gnorm = np.linalg.norm(gradient)

            print(gnorm)

            if gnorm < self.epsilon:

                break

            w = w - np.dot(gradient,

np.linalg.pinv(self.second_derivative(w)))

        return w
```