第 5 章

CHAPTER 5

Python 回归算法

回归(Regression)是另一类重要的监督学习算法。与分类问题不同的是,在回归问题中,其目标是通过对训练样本的学习,得到从样本特征到样本标签之间的映射,其中,在回归问题中,样本标签是连续值。典型的回归问题有:

(1) 根据人的身高、性别和体重等信息预测其鞋子的大小;

(2) 根据房屋的面积、卧室的数量预测房屋的价格等。

5.1 线性回归

线性回归(Linear Regression)是一类重要的回归问题。在线性回归中,目标值与特征之间存在线性相关的关系。

5.1.1 基本线性回归

1. 线性回归的模型

对于线性回归算法,我们希望从训练数据中学习到线性回归方程,即

$$y = b + \sum_{i=1}^{n} w_i \cdot x_i$$

其中,b 称为偏置,w_i 为回归系数。对于上式,令 $x_0 = 1$,则上式可以表示为:

$$y = \sum_{i=0}^{n} w_i \cdot x_i$$

2. 线性回归模型的损失函数

在线性回归模型中,其目标是求出线性回归方程,即求出线性回归方程中的回归系数 w_i。线性回归的评价是指如何度量预测值(Prediction)与标签(Label)之间的接近程度,线性回归模型的损失函数可以是绝对损失(Absolute Loss)或者平方损失(Squared Loss)。其中,绝对损失函数为:

$$l = |y - \hat{y}|$$

式中的 $\hat{y}$ 为预测值,且 $\hat{y} = \sum_{i=0}^{n} w_i \cdot x_i$。

平方损失函数为:

$$l = (y - \hat{y})^2$$

由于平方损失处处可导,所以通常使用平方误差作为线性回归模型的损失函数。假设有 m 个训练样本,每个样本中有 $n-1$ 个特征,则平方误差可以表示为

$$l=\frac{1}{2}\sum_{i=1}^{m}\left(y^{(i)}-\sum_{j=0}^{n-1}w_j\cdot x_j^{(i)}\right)^2$$

对于如上的损失函数，线性回归的求解是希望求得平方误差的最小值。

【例 5-1】（二维直线例子）已知线性方程 $y=ax+b$ 表示平面上的一直线。在下面的例子中，根据房屋面积、房屋价格的历史数据，建立线性回归模型。然后，根据给出的房屋面积，来预测房屋价格。房屋的数据为：

```
   square_feet  price
0          150   6450
1          200   7450
2          250   8450
3          300   9450
4          350  11450
5          400  15450
6          600  18450
```

实现的 Python 代码如下：

```
import pandas as pd
from io import StringIO
from sklearn import linear_model
import matplotlib.pyplot as plt

#房屋面积与价格历史数据(CSV 文件)
csv_data = 'square_feet,price\n150,6450\n200,7450\n250,8450\n300,9450\n350,11450\n400,15450\
n600,18450\n'
#读入 DataFrame
df = pd.read_csv(StringIO(csv_data))
print(df)
#建立线性回归模型
regr = linear_model.LinearRegression()
#拟合
regr.fit(df['square_feet'].values.reshape(-1, 1), df['price']) #注意此处.reshape(-1, 1),
                                                               #因为 X 是一维的!
#不难得到直线的斜率、截距
a, b = regr.coef_, regr.intercept_
#给出待预测面积
area = 238.5
#方式 1:根据直线方程计算的价格
print(a * area + b)
#方式 2:根据 predict 方法预测的价格
print(regr.predict(area))
#画图
#1.真实的点
plt.scatter(df['square_feet'], df['price'], color = 'blue')
#2.拟合的直线
plt.plot(df['square_feet'], regr.predict(df['square_feet'].values.reshape(-1,1)), color = 'red',
linewidth = 4)
plt.show()
```

运行程序，得到房屋的预测价格如下，效果如图 5-1 所示。

```
[8635.02659574]
[8635.02659574]
```

【例 5-2】（三维平面的例子）已知线性方程 $z=ax+by+c$ 表示一空间平面，利用一组虚拟的数据绘制其平面。

```
import numpy as np
from sklearn import linear_model
from mpl_toolkits.mplot3d import Axes3D
```

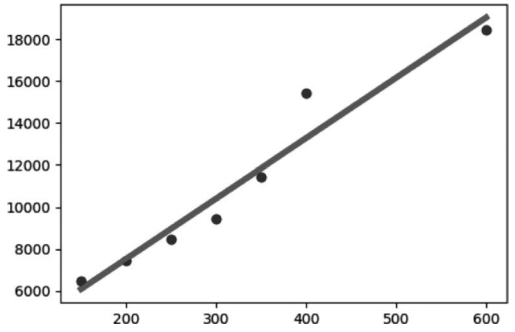

图 5-1　房屋面积与价格之间的关系

```
import matplotlib.pyplot as plt

xx, yy = np.meshgrid(np.linspace(0,10,10), np.linspace(0,100,10))
zz = 1.0 × xx + 3.5 × yy + np.random.randint(0,100,(10,10))
#构建成特征、值的形式
X, Z = np.column_stack((xx.flatten(),yy.flatten())), zz.flatten()
#建立线性回归模型
regr = linear_model.LinearRegression()
#拟合
regr.fit(X, Z)
#不难得到平面的斜率、截距
a, b = regr.coef_, regr.intercept_
#给出待预测的一个特征
x = np.array([[5.8, 78.3]])
#方式 1:根据线性方程计算待预测的特征 x 对应的值 z(注意:np.sum)
print(np.sum(a * x) + b)
#方式 2:根据 predict 方法预测的值 z
print(regr.predict(x))
#画图
fig = plt.figure()
ax = fig.gca(projection = '3d')
#1.画出真实的点
ax.scatter(xx, yy, zz)
#2.画出拟合的平面
ax.plot_wireframe(xx, yy, regr.predict(X).reshape(10,10))
ax.plot_surface(xx, yy, regr.predict(X).reshape(10,10), alpha = 0.3)
```

运行程序,得到输出值如下,效果如图 5-2 所示。

```
335.4328672727273
[335.43286727]
```

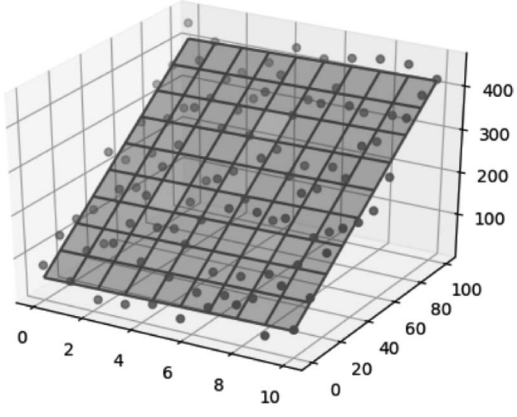

图 5-2　绘制空间平面图

5.1.2　线性回归的最小二乘解法

对于线性回模型，假设训练集中有 m 个训练样本，每个训练样本中有 $n-1$ 个特征，可以使用矩阵的表示方法，预测函数可以表示为：

$$\boldsymbol{Y}=\boldsymbol{XW}$$

其损失函数可以表示为：

$$(\boldsymbol{Y}-\boldsymbol{XW})^{\mathrm{T}}(\boldsymbol{Y}-\boldsymbol{XW})$$

其中，标签 $\boldsymbol{Y}$ 为 $m\times 1$ 的矩阵，训练特征 $\boldsymbol{X}$ 为 $m\times n$ 的矩阵，回归系数 $\boldsymbol{W}$ 为 $n\times 1$ 的矩阵。在最小二乘法中，对 $\boldsymbol{W}$ 求导，即

$$\frac{\mathrm{d}}{\mathrm{d}W}(\boldsymbol{Y}-\boldsymbol{XW})^{\mathrm{T}}(\boldsymbol{Y}-\boldsymbol{XW})=\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{Y}-\boldsymbol{XW})$$

令其为 0，得到

$$\hat{\boldsymbol{W}}=(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

现在利用 Python 实现最小二乘的解法。

【例 5-3】 利用最小二乘法求解。

```
import numpy as np
#导入 Python 的矩阵计算模块
import matplotlib.pyplot as plt

def fun2ploy(x,n):
    '''
    数据转换为[x^0,x^1,x^2,…,x^n]
    首列变 1
    '''
    lens = len(x)
    X = np.ones([1,lens])
    for i in range(1,n):
        X = np.vstack((X,np.power(x,i)))                    #按行堆叠
    return X

def leastseq_byploy(x,y,ploy_dim):
    '''
    最小二乘求解
    '''
    #散点图
    plt.scatter(x,y,color = "r",marker = 'o',s = 50)
    X = fun2ploy(x,ploy_dim);
    #直接求解
    Xt = X.transpose();                                     #转置变成列向量
    XXt = X.dot(Xt);                                        #矩阵乘
    XXtInv = np.linalg.inv(XXt)                             #求逆
    XXtInvX = XXtInv.dot(X)
    coef = XXtInvX.dot(y.T)
    y_est = Xt.dot(coef)
    return y_est,coef

def fit_fun(x):
    '''
    要拟合的函数
    '''
    #return np.power(x,5)
    return np.sin(x)
```

```
if __name__ == '__main__':
    data_num = 100;
    ploy_dim = 10;                                          #拟合参数个数,即权重数量
    noise_scale = 0.2;
    ##数据准备
    x = np.array(np.linspace( - 2 * np.pi,2 * np.pi,data_num))    #数据
    y = fit_fun(x) + noise_scale * np.random.rand(1,data_num)     #添加噪声
    #最小二乘拟合
    [y_est,coef] = leastseq_byploy(x,y,ploy_dim)
    #显示拟合结果
    org_data = plt.scatter(x,y,color = "r",marker = 'o',s = 50)
    est_data = plt.plot(x,y_est,color = "g",linewidth = 3)
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.title("Fit funtion with leastseq method")
    plt.legend(["Noise data","Fited function"]);
    plt.show()
```

运行程序,效果如图 5-3 所示。

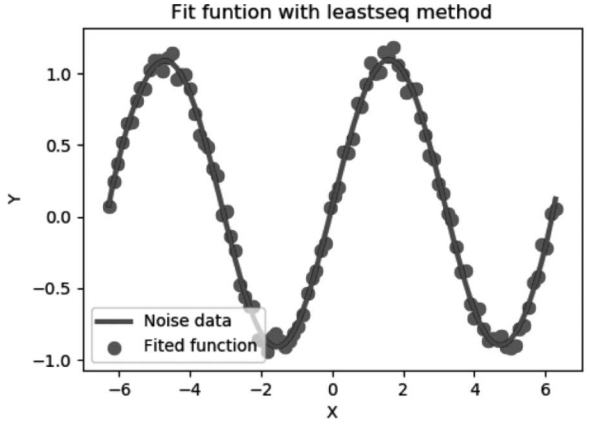

图 5-3　最小二乘拟合效果

我们都知道用最小二乘拟合线性函数没有问题,那么能不能拟合二次函数甚至更高次的函数呢?答案当然是可以的。下面就试试用最小二乘来拟合抛物线形状的图像。

对于二次函数来说,一般形式为 $f(x)=ax^2+bx+c$,其中 a、b、c 为 3 个要求解的参数。为了确定 a、b、c,需要根据给定的样本,然后通过调整这些参数,直到最后找出一组参数 a、b、c,使这些所有的样本点距离 $f(x)$的距离平方和最小。用什么方法来调整这些参数呢?最常见的就是梯度下降法。

scipy 库中有名为 leastsq 的方法,只需要输入一系列样本点,给出待求函数的基本形式,就可以针对上述问题求解了。

【例 5-4】 利用最小二乘拟合二次抛物线函数。

```
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import leastsq
#待拟合的数据
X = np.array([1,2,3,4,5,6])
Y = np.array([9.1,18.3,32,47,69.5,94.8])
#二次函数的标准形式
def func(params, x):
    a, b, c = params
    return a * x * x + b * x + c
```

```
#误差函数,即拟合曲线所求的值与实际值的差
def error(params, x, y):
    return func(params, x) - y

#对参数求解
def slovePara():
    p0 = [10, 10, 10]
    Para = leastsq(error, p0, args=(X, Y))
    return Para

#输出最后的结果
def solution():
    Para = slovePara()
    a, b, c = Para[0]
    print("a=",a," b=",b," c=",c)
    print( "cost:" + str(Para[1]))
    print( "求解的曲线是:")
    print("y=" + str(round(a,2)) + "x*x+" + str(round(b,2)) + "x+" + str(c))
    plt.figure(figsize=(8,6))
    plt.scatter(X, Y, color="green", label="sample data", linewidth=2)

    #    画拟合直线
    x=np.linspace(0,12,100)                                    #在0～12直接画100个连续点
    y=a*x*x+b*x+c                                              #函数式
    plt.plot(x,y,color="red",label="solution line",linewidth=2)
    plt.legend()                                               #绘制图例
    plt.show()
solution()
```

运行程序,输出如下,效果如图 5-4 所示。

```
a= 2.066071414252538  b= 2.597500103604725  c= 4.689999854964827
cost:1
求解的曲线是:
y=2.07x*x+2.6x+4.689999854964827
```

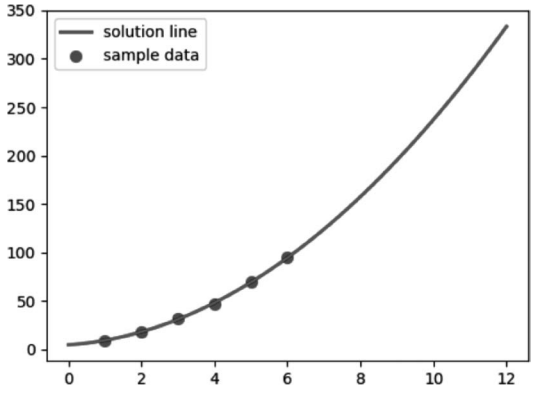

图 5-4　最小二乘拟合二次抛物线效果

在上面的代码中,需注意的是如下几点。

(1) func 是待拟合的曲线的形式,本例中为二次函数的标准形式。

(2) error 为误差函数。很多人会问不应该是最小平方和吗? 为什么不是 func(params, x) − y × func(params, x)−y? 原因是 leastsq 已进行最小平方和计算了。sklearn 中源码为:

```
Minimize the sum of squares of a set of equations.
```

```
x = arg min(sum(func(y) ** 2,axis = 0))
        y
```

leastsq 函数除了可以模拟线性函数二次函数等多项式，还适用于任何波形的模拟。比如方波：

```
def square_wave(x,p):
    a, b, c, T = p
    y = np.where(np.mod(x - b,T)< T/2, 1 + c/a, 0)
    y = np.where(np.mod(x - b,T)> T/2, - 1 + c/a, y)
    return a * y
```

比如高斯分布：

```
def gaussian_wave(x,p):
    a, b, c, d = p
    return a * np.exp( - (x - b) ** 2/(2 * c ** 2)) + d
```

只要将上面代码中的 func 换成对应的函数即可。

5.1.3 牛顿法

除了前面介绍的梯度下降法，牛顿法也是机器学习中用得比较多的一种优化算法。牛顿法的基本思想是利用迭代点 x_k 处的一阶导数（梯度）和二阶导数（海森矩阵）对目标函数进行二次函数近似，然后把二次函数的极小点作为新的迭代点，并不断重复这一过程，直至求得满足精度的近似值。牛顿法下降的速度比梯度下降得快，而且能高度逼近最优值。

1. 基本牛顿法的原理

基本牛顿法是一种基于导数的算法，它每一步的迭代方向都沿着当前点函数值下降的方向。对于一维的情形，对于一个需要求解的优化函数 $f(x)$，求函数极值的问题可以转换为求导函数 $f'(x)=0$。对函数 $f(x)$ 进行泰勒展开到二阶，得到：

$$f(x)=f(x_k)+f'(x_k)(x-x_k)+\frac{1}{2}f''(x_k)(x-x_k)^2$$

对上式求导并令其为 0，则有

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

即得到

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

这就是牛顿法的更新公式。

2. 基本牛顿法的流程

基本牛顿法的流程主要表现在：

（1）给定终止误差值 $0\leqslant\varepsilon\ll1$，初始点 $x_0\in\mathbf{R}^n$，令 $k=0$；

（2）计算 $\boldsymbol{g}_k=\nabla f(x_k)$，如果 $\|\boldsymbol{g}_k\|\leqslant\varepsilon$，则停止，输出 $x^*\approx x_k$；

（3）计算 $G_k=\nabla^2 f(x_k)$，并求解线性方程组 $G_k d=-g^k$ 得到解 d_k；

（4）令 $x_{k+1}=x_k+d_k k=k+1$，并转到步骤（2）。

3. 全局牛顿法

牛顿法最突出的优点是收敛速度快，具有局部二阶收敛性，但是，基本牛顿法初始点需要足够“靠近”极小点；否则，有可能导致算法不收敛，此时就引入了全局牛顿法。全局牛顿法的流程主要表现为：

（1）给定终止误差值 $0\leqslant\varepsilon\ll1$，$\delta\in(0,1)$，$\sigma\in(0,0.5)$，初始点 $x_0\in\mathbf{R}^n$，令 $k=0$；

(2) 计算 $\boldsymbol{g}_k=\nabla f(x_k)$,如果 $\|\boldsymbol{g}_k\|\leqslant\varepsilon$,则停止,输出 $x^*\approx x_k$;

(3) 计算 $G_k=\nabla^2 f(x_k)$,并求解线性方程组 $G_k d=-\boldsymbol{g}^k$ 得到解 d_k;

(4) 设 m_k 是不满足下列不等式的最小非负整数 m:

$$f(x_k+\delta^m d_k)\leqslant f(x_k)+\sigma\delta^m \boldsymbol{g}_k^{\mathrm{T}} d_k$$

(5) 令 $\alpha_k=\delta^{m_k}$,$x_{k+1}=x_k+\alpha_k d_k$,$k=k+1$,并转到步骤(2)。

【例 5-5】 以 Rosenbrock 函数为例,即有

$$f(\boldsymbol{x})=100(x_2-x_1^2)^2+(1-x_1)^2$$

于是可得函数的梯度

$$g(\boldsymbol{x})=\nabla f(\boldsymbol{x})=(-400(x_2-x_1^2)x_1-2(1-x_1)200(x_2-x_1^2))^{\mathrm{T}}$$

函数 $f(\boldsymbol{x})$的海森矩阵为

$$\begin{bmatrix}-400(x_2-3x_1^2)+2 & -400x_1\\ -400x_1 & 200\end{bmatrix}$$

利用牛顿法求解的 Python 代码为:

```
"""
牛顿法
Rosenbrock 函数
函数 f(x) = 100 * (x(2) - x(1).^2).^2 + (1 - x(1)).^2
梯度 g(x) = ( - 400 * (x(2) - x(1)^2) * x(1) - 2 * (1 - x(1)),200 * (x(2) - x(1)^2))^(T)
"""
import numpy as np
import matplotlib.pyplot as plt

def jacobian(x):
    return np.array([ - 400 * x[0] * (x[1] - x[0] ** 2) - 2 * (1 - x[0]),200 * (x[1] - x[0] ** 2)])
def hessian(x):
    return np.array([[ - 400 * (x[1] - 3 * x[0] ** 2) + 2, - 400 * x[0]],[ - 400 * x[0],200]])

X1 = np.arange( - 1.5,1.5 + 0.05,0.05)
X2 = np.arange( - 3.5,2 + 0.05,0.05)
[x1,x2] = np.meshgrid(X1,X2)
f = 100 * (x2 - x1 ** 2) ** 2 + (1 - x1) ** 2;                    #给定的函数
plt.contour(x1,x2,f,20)                                           #画出函数的 20 条轮廓线

def newton(x0):
    print('初始点为:')
    print(x0,'\n')
    W = np.zeros((2,10 ** 3))
    i = 1
    imax = 1000
    W[:,0] = x0
    x = x0
    delta = 1
    alpha = 1
    while i < imax and delta > 10 ** ( - 5):
        p = - np.dot(np.linalg.inv(hessian(x)),jacobian(x))
        x0 = x
        x = x + alpha * p
        W[:,i] = x
        delta = sum((x - x0) ** 2)
        print('第',i,'次迭代结果:')
        print(x,'\n')
        i = i + 1
```

```
    W = W[:,0:i]                                    #记录迭代点
    return W

x0 = np.array([-1.2,1])
W = newton(x0)
plt.plot(W[0,:],W[1,:],'g*',W[0,:],W[1,:])          #画出迭代点收敛的轨迹
plt.show()
```

运行程序,输出如下,效果如图 5-5 所示。

```
初始点为:
[-1.2  1. ]
第 1 次迭代结果:
[-1.1752809   1.38067416]
第 2 次迭代结果:
[ 0.76311487 -3.17503385]
第 3 次迭代结果:
[0.76342968 0.58282478]
第 4 次迭代结果:
[0.99999531 0.94402732]
第 5 次迭代结果:
[0.9999957  0.99999139]
第 6 次迭代结果:
[1. 1.]
```

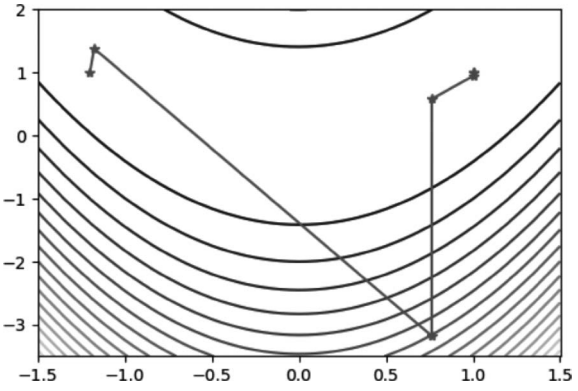

图 5-5　牛顿迭代法求解

【例 5-6】 Python 实现牛顿迭代法求三角函数极值。

```
#coding = utf-8
import math
a = 0.001                                           #定义收敛步长
xd = 1                                              #定义寻找步长
x = 0                                               #定义一个种子 x0
i = 0                                               #循环迭代次数
y = 0
dic = {}
import math
def f(x):
    y = math.sin(x)                                 #定义函数 f(x) = sinx
    return y
def fd(x):
    y = math.cos(x)                                 #函数 f(x)的导数 fd(X) = cosx
    return y
while y >= 0 and y < 3.14 * 4:
    y = y + xd
    x = y
    while abs(fd(x))> 0.001:                        #定义精度为 0.001
```

```
            x = x + fd(x)/f(x)
        if x >= 0 and x < 3.14 * 4:
            #print(x,f(x))
            dic[y] = x
#print(dic)
ls = []
for i in dic.keys():
    cor = 0
    if ls is None:
        ls.append(dic[i])
    else:
        for j in ls:
            if dic[i] - j < 0.1:
                cor = 1
                break
        if cor == 0:
            ls.append(dic[i])
print(ls)
```

运行程序，输出如下：

```
[1.5706752771612507, 4.712388980912051, 7.8539818558789225, 10.995653476776056]
```

5.1.4　局部加权线性回归

在线性回归中会出现欠拟合的情况，有些方法可以用来解决这样的问题。局部加权线性回归(Locally Weighted Linear Regression，LWLR)就是这样的一种方法。局部加权线性回归采用的是给预测点附近的每个点赋予一定的权重，此时的回归系数可以表示为：

$$\hat{\boldsymbol{W}}=(\boldsymbol{X}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{X})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{M}\boldsymbol{Y}$$

其中，$\boldsymbol{M}$ 为每个点的权重。

LWLR 使用核函数来对附近的点赋予更高的权重，常用的有高斯核，对应的权重为：

$$M(i,j)=\exp\left(\frac{\|\boldsymbol{X}^{i}-\boldsymbol{X}\|^{2}}{-2k^{2}}\right)$$

这样的权重矩阵只含对角元素。

【例 5-7】 给定不同的 k 值，绘制对应的局部线性拟合图像。

```
#k = 1.0 出现了欠拟合,k = 0.1 时效果最佳,k = 0.003 时出现了过拟合
from numpy import *
def loadDataSet(filename):
    numFeat = len(open(filename).readline().split('\t')) - 1
    dataMat = []
    labelMat = []
    fr = open(filename)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat,labelMat
def standRegress(xArr,yArr):
    xMat = mat(xArr)
    yMat = mat(yArr).T
    xTx = xMat.T * xMat
    if linalg.det(xTx) == 0.0:
```

```
            print('error')
            return
        ws = xTx.I * (xMat.T * yMat)
        return ws
    def lwlr(testPoint,xArr,yArr,k = 1.0):
        xMat = mat(xArr)
        yMat = mat(yArr).T
        m = shape(xMat)[0]
        weights = mat(eye((m)))
        for j in range(m):
            diffMat = testPoint - xMat[j,:]
            weights[j,j] = exp(diffMat * diffMat.T/( - 2.0 * k ** 2))
        xTx = xMat.T * (weights * xMat)
        if linalg.det(xTx) == 0.0:
            print("error")
            return
        ws = xTx.I * (xMat.T * (weights * yMat))
        return testPoint * ws
    def lwlrTest(testArr,xArr,yArr,k = 1.0):
        m = shape(testArr)[0]
        yHat = zeros(m)
        for i in range(m):
            yHat[i] = lwlr(testArr[i],xArr,yArr,k)
        return yHat
    def rssError(yArr,yHatArr):
        return ((yArr - yHatArr) ** 2).sum()

    xArr,yArr = loadDataSet('ex0.txt')
    xMat = mat(xArr)
    yMat = mat(yArr)
    k = [1.0,0.01,0.003]
    for i in range(3):
        yHat = lwlrTest(xArr, xArr, yArr, k[i])
        srtInd = xMat[:, 1].argsort(0)
        xSort = xMat[srtInd][:, 0, :]
        import matplotlib.pyplot as plt
        fig = plt.figure(i + 1)
        ax = fig.add_subplot(111)
        ax.plot(xSort[:, 1], yHat[srtInd])
        ax.scatter(xMat[:, 1].flatten().A[0], yMat.T[:, 0].flatten().A[0], s = 2, c = 'red')
        plt.title('k = %g' % k[i])
    plt.show()
```

当 k＝1 时，最终的结果如图 5-6 所示；当 k＝0.003 时，最终的结果如图 5-7 所示；当 k＝0.1 时，最终的结果如图 5-8 所示。

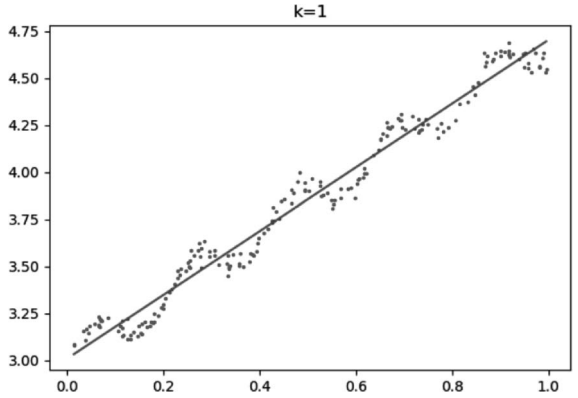

图 5-6　k＝1 时的加权线性拟合效果

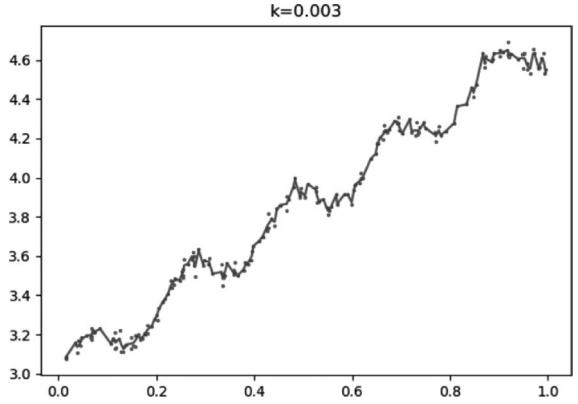

图 5-7　k＝0.003 时的加权线性拟合效果

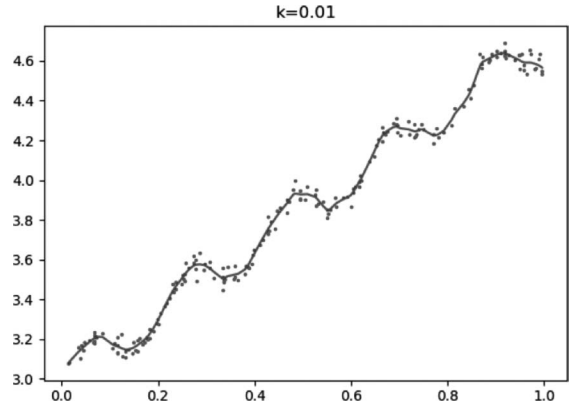

图 5-8　k＝0.01 时的加权线性拟合效果

当 k 的值逐渐变小，其拟合数据的能力也在变强。当 k 取较大值时，如图 5-6 所示，出现了欠拟合，不能很好地反映数据的真实情况；当 k 值取较小值时，如图 5-7 所示，出现了过拟合。

5.2　非线性回归

如果回归模型的因变量是自变量的一次以上函数形式，回归规律在图形上表现为形态各异的各种曲线，称之为非线性回归，这类模型被称为非线性回归模型。在许多实际问题中，回归函数往往是较复杂的非线性函数。

多项式回归的一般表达形式为

$$y=\beta_0+\beta_1 x_1+\beta_4 x_2+\beta_2 x_1 x_2+\beta_3 x_3^2+\varepsilon$$

其中，x_1、x_2 以及 x_3 和因变量都不是线性关系，但是如果把 x_1x_2 替换为 x_4，把 x_3^2 替换为 x_5，则原式可以改写为

$$y=\beta_0+\beta_1 x_1+\beta_4 x_2+\beta_2 x_4+\beta_3 x_5+\varepsilon$$

下面通过一个实例来演示非线性回归应用。

本实例将要拟合我国从 1960 年到 2014 年的 GDP 数据。下载的数据(china_gdp.csv)有两列，第一列是年份(Year，1960—2014)，第二列是对应年份的国内生产总值(Value)。

具体的实现步骤如下。

(1) 导入相应库,加载数据,并显示数据的前 10 行,代码如下。

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
#显示中文
plt.rcParams['font.sans-serif'] = ['SimHei']
plt.rcParams['axes.unicode_minus'] = False
df = pd.read_csv("china_gdp.csv")
df.head(10)
```

运行程序,输出如下:

```
df.head(10)
    Year           Value
0   1960   5.918412e+10
1   1910   4.955705e+10
2   1962   4.668518e+10
3   1963   5.009730e+10
4   1964   5.906225e+10
5   1965   6.970915e+10
6   1966   7.587943e+10
7   1967   7.205703e+10
8   1968   6.999350e+10
9   1969   7.871882e+10
```

(2) 数据可视化。

数据一开始增长得特别慢,从 2005 年开始,增长速度就非常显著了,在 2010 年略微减速,代码如下,运行效果如图 5-9 所示。

```
plt.figure(figsize=(8,5))
x_data, y_data = (df["Year"].values, df["Value"].values)
plt.plot(x_data, y_data, 'ro')
# plt.stem(x_data, y_data)
plt.ylabel('GDP')
plt.xlabel('年份')
plt.show()
```

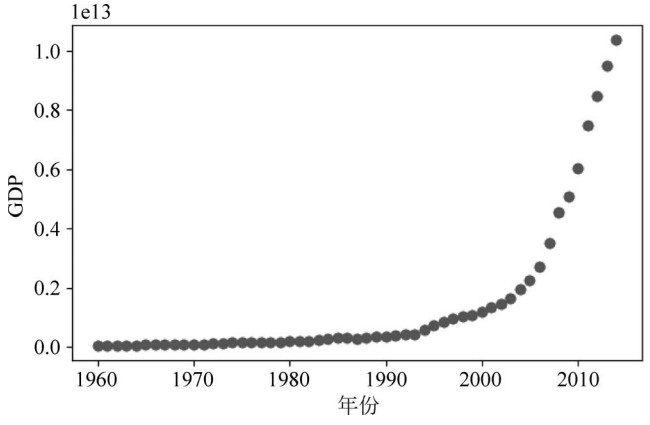

图 5-9 数据可视化

(3) 选择模型。

从图 5-9 可看出,数据一开始增长很慢,中间增长很快,最后又慢了下来,可选用 logistic(逻辑)函数作为模型,代码如下,运行效果如图 5-10 所示。

```
X = np.arange(-5.0, 5.0, 0.1)
```

```
Y = 1.0 / (1.0 + np.exp( - X))
plt.plot(X,Y)
plt.ylabel('因变量')
plt.xlabel('自变量')
plt.show()
```

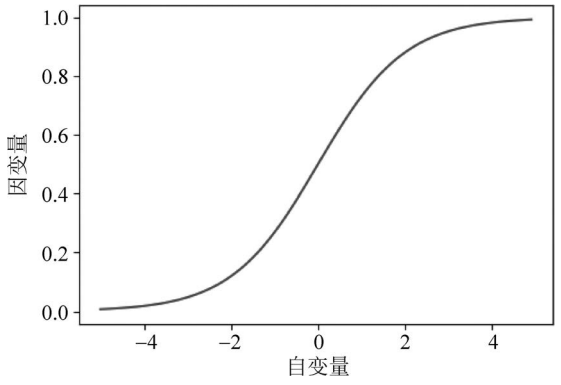

图 5-10　logistic 函数曲线

logistic 函数的方程可表示为

$$\hat{Y}=\frac{1}{1+e^{-\beta_1(X-\beta_2)}}$$

其中，β_1 表示控制曲线的陡度，β_2 为在 x 轴上平移的值。

(4) 构建模型。

接下来，构建回归模型并且初始化参数，代码如下。

```
#构建回归模型
def sigmoid(x, Beta_1, Beta_2):
     y = 1 / (1 + np.exp( - Beta_1 * (x - Beta_2)))
     return y

#初始化参数
beta_1 = 0.10
beta_2 = 1990.0
#逻辑函数
Y_pred = sigmoid(x_data, beta_1 , beta_2)

#根据数据点绘制初始预测图,如图 5 - 11 所示
plt.plot(x_data, Y_pred * 15000000000000.)
plt.plot(x_data, y_data, 'ro')
```

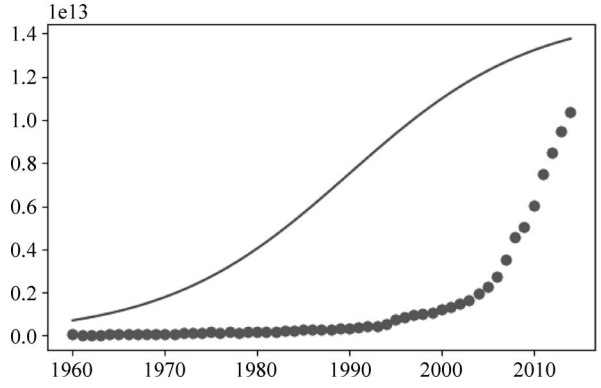

图 5-11　初始预测图

(5) 寻找最优参数。

目标是寻找最优参数。第一步把 x 和 y 都标准化,使用非线性最小二乘拟合(curve_fit) sigmoid 函数。优化参数值,使 sigmoid(xdata, * popt)－ydata 的残差平方和最小化。代码如下。

```
#规范化数据
xdata = x_data/max(x_data)
ydata = y_data/max(y_data)
#优化参数
from scipy.optimize import curve_fit
popt, pcov = curve_fit(sigmoid, xdata, ydata)
#打印最终参数
print(" beta_1 = %f, beta_2 = %f" % (popt[0], popt[1]))
#绘制回归模型
x = np.linspace(1960, 2015, 55)
x = x/max(x)
plt.figure(figsize = (8,5))
y = sigmoid(x, * popt)
plt.plot(xdata, ydata, 'ro', label = 'data')
plt.plot(x,y, linewidth = 3.0, label = 'fit')
plt.legend(loc = 'best')
plt.ylabel('GDP')
plt.xlabel('年份')
plt.show()
```

运行程序,输出如下,效果如图 5-12 所示。

```
beta_1 = 690.451712, beta_2 = 0.997207
```

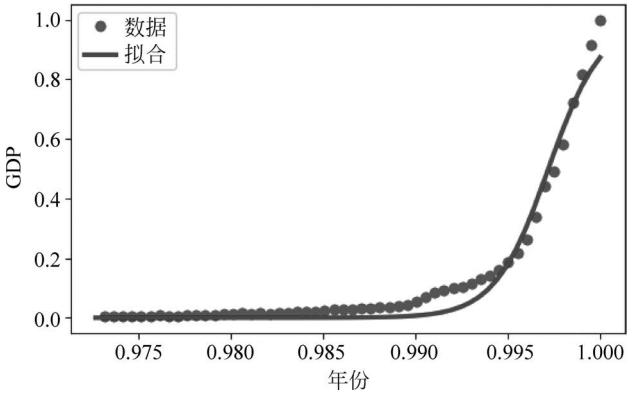

图 5-12　回归模型

(6) 评估模型。

从图 5-12 可看出效果很好,但在运行过程中 R^2 是负的,而且当 R^2 很大时,在测试集上的实际效果不是很好。

```
#把数据分为训练集和测试集
msk = np.random.rand(len(df))< 0.8
train_x = xdata[msk]
test_x = xdata[～msk]
train_y = ydata[msk]
test_y = ydata[～msk]

#用训练集建立一个模型
popt,pcov = curve_fit(sigmoid,train_x,train_y)

yyy = sigmoid(train_x, * popt)
```

```
plt.plot(train_x, train_y, 'ro', label = 'data')
plt.plot(train_x,yyy, linewidth = 3.0, label = 'fit')
plt.plot(test_x, test_y, 'go', label = 'data')
#在测试集上预测
y_hat = sigmoid(test_x, * popt)
print("test_x:",test_x * sum(df["Year"]))
print("test_y:",test_y * sum(df['Value']))
print("y_hat:",y_hat * sum(df['Value']))
#评估
print("平均绝对误差: %.2f" % np.mean(np.absolute(y_hat - test_y)))
print("残差平方和 (MSE): %.2f" % np.mean((y_hat - test_y) ** 2))
from sklearn.metrics import r2_score
print("R2 值: %.2f" % r2_score(test_y,y_hat) )
```

运行程序,输出如下:

```
 beta_1 = 690.451712, beta_2 = 0.997207
test_x: [106409.07894737 107168.75620655 107331.54419067 107385.80685204
 107548.59483615 107819.908143   107874.17080437 107982.69612711
 108145.48411122 108308.27209533 108579.58540218 108742.3733863
 109176.47467726 109230.73733863]
test_y: [3.78263553e + 11 1.23013570e + 12 1.13258445e + 12 1.34992655e + 12
 1.55367208e + 12 2.07118319e + 12 2.37170873e + 12 2.74000347e + 12
 3.38041343e + 12 6.57072900e + 12 1.01688021e + 13 1.48211321e + 13
 6.45866456e + 13 7.24407333e + 13]
y_hat: [1.44217819e + 07 1.38790719e + 09 3.69276039e + 09 5.11694904e + 09
 1.36134364e + 10 6.95023443e + 10 9.62764283e + 10 1.84658389e + 11
 4.89429988e + 11 1.28899275e + 12 6.17191244e + 12 1.45366937e + 13
 5.95876395e + 13 6.39691865e + 13]
平均绝对误差: 0.03
残差平方和 (MSE): 0.01
R2 值: 0.98
```

5.3　岭回归与 Lasso 回归

在处理较为复杂的数据回归问题时,普通的线性回归算法通常会出现预测精度不够,如果模型中的特征之间有相关关系,就会增加模型的复杂程度,并且对整个模型的解释并不高,这时就需要对数据中的特征进行选择。对于回归算法,特征选择的方法有岭回归(Ridge Regression)和 Lasso 回归。

岭回归和 Lasso 回归都属于正则化的特征选择方法,对于处理较为复杂的数据问题通常选用这两种方法。

5.3.1　线性回归存在的问题

如果模型中的特征之间有相关关系,就会增加模型的复杂程度。当数据中的特征之间有较强的线性相关时,即特征之间出现严重共线时,用普通最小二乘法估计模型参数,往往参数估计的方差太大,此时,求解出来的模型就很不稳定。在具体取值上与真值有较大的偏差,有时会出现与实际意义不符的正负号。

假设已知线性回归模型为:

$$y = 10 + 2x_1 + 5x_2$$

其中,$x_1 \in (0,10)$,$x_2 \in (10,25)$。其中部分训练数据如表 5-1 所示。

表 5-1 部分训练数据

x_1	0.444 71	9.788 73	10.463 42	9.022 327	0.931 401 4	10.263 29	10.464 25	9.875 585
x_2	11.2567	10.8953	21.5342	19.5781	22.674 84	24.3317	18.1766	19.1721

利用普通最小二乘法求回归系数的估计得：

$$w_0 = 22.709\ 626\ 465\ 5$$

$$w_1 = 3.066\ 285\ 457\ 42$$

$$w_2 = 4.078\ 313\ 815\ 18$$

这与实际模型中的参数有很大的差别。计算 x_1、x_2 的样本相关系数得 $r_{12}=0.9854$，表明 x_1 与 x_2 之间高度相关。通过这个例子可以看到，解释变量之间高度相关时，普通最小二乘估计明显变坏。

5.3.2 岭回归模型

岭回归是在平方误差的基础上增加正则项：

$$l = \sum_{i=1}^{m}\left(y^{(i)} - \sum_{j=0}^{n} w_j x_j^{(i)}\right)^2 + \lambda \sum_{j=0}^{n} w_j^2$$

其中，$\lambda > 0$。通过确定 λ 的值可以使得在方差和偏差之间达到平衡；随着 λ 的增大，模型方差减小而偏差增大。

1. 岭回归模型的求解

与线性回归一样，在利用最小二乘法求解岭回归模型的参数时，首先对 $\boldsymbol{W}$ 求导，结果为：

$$2\boldsymbol{X}^{\mathrm{T}}(\boldsymbol{Y} - \boldsymbol{X}\boldsymbol{W}) - 2\lambda\boldsymbol{W}$$

令其为 0，可求得 $\boldsymbol{W}$ 的值为：

$$\hat{\boldsymbol{W}} = (\boldsymbol{X}^{\mathrm{T}}\boldsymbol{X} + \lambda\boldsymbol{I})^{-1}\boldsymbol{X}^{\mathrm{T}}\boldsymbol{Y}$$

其中，$\boldsymbol{I}$ 为单位矩阵。

【例 5-8】 下面给出一个岭回归简单的代码示例，这个代码显示了不同的 alpha 对模型参数 weights 的影响程度。alpha 越大，则 weights 的数值越小；alpha 越小，则 weights 的数值越大，注意为了更好地观察所生成的图片，将 x 轴作了反转。

```
"""
岭回归测试代码
这里需要先生成一个线性相关的设计矩阵 X,再使用岭回归对其进行建模
岭回归中最重要的就是参数 alpha 的选择,本例显示了不同的 alpha 下
模型参数 omega 不同的结果
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

#这里设计矩阵 X 是一个希尔伯特矩阵(Hilbert matrix)
#其元素 A(i,j) = 1(i + j - 1),i 和 j 分别为其行标和列标
#希尔伯特矩阵是一种数学变换矩阵,正定且高度病态
#任何一个元素发生一点变动,整个矩阵的行列式的值和逆矩阵都会发生巨大变化
#这里设计矩阵是一个 10×5 的矩阵,即有 10 个样本,5 个变量
X = 1. / (np.arange(1, 6) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)

print('设计矩阵为:')
```

```
print(X)

#alpha 取值为 10^(-10)到 10^(-2)之间的连续的 200 个值
n_alphas = 200
alphas = np.logspace(-10, -2, n_alphas)
print('\n alpha 的值为:')
print(alphas)

#初始化一个 Ridge Regression
clf = linear_model.Ridge(fit_intercept=False)

#参数矩阵,即每一个 alpha 对应的参数所组成的矩阵
coefs = []
#根据不同的 alpha 训练出不同的模型参数
for a in alphas:
    clf.set_params(alpha=a)
    clf.fit(X, y)
    coefs.append(clf.coef_)
#获得绘图句柄
ax = plt.gca()
#参数中每一个维度使用一个颜色表示
ax.set_color_cycle(['b', 'r', 'g', 'c', 'k'])

#绘制 alpha 和对应的参数之间的关系图
ax.plot(alphas, coefs)
ax.set_xscale('log')                                    #x 轴使用对数表示
ax.set_xlim(ax.get_xlim()[::-1])                        #将 x 轴反转,便于显示
plt.grid()
plt.xlabel('alpha')
plt.ylabel('weights')
plt.title('Ridge coefficients as a function of the regularization')
plt.axis('tight')
plt.show()
```

运行程序,输出如下,效果如图 5-13 所示。

```
设计矩阵为:
[[1.          0.5         0.33333333 0.25        0.2        ]
 [0.5         0.33333333 0.25        0.2         0.16666667]
 [0.33333333 0.25        0.2         0.16666667 0.14285714]
 [0.25        0.2         0.16666667 0.14285714 0.125      ]
 [0.2         0.16666667 0.14285714 0.125       0.11111111]
 [0.16666667 0.14285714 0.125       0.11111111 0.1        ]
 [0.14285714 0.125       0.11111111 0.1         0.09090909]
 [0.125       0.11111111 0.1         0.09090909 0.08333333]
 [0.11111111 0.1         0.09090909 0.08333333 0.07692308]
 [0.1         0.09090909 0.08333333 0.07692308 0.07142857]]
alpha 的值为:
[1.00000000e-10 1.09698580e-10 1.20337784e-10 1.32008840e-10
 1.44811823e-10 1.58856513e-10 1.74263339e-10 1.91164408e-10
 2.09704640e-10 2.30043012e-10 2.52353917e-10 2.76828663e-10
 3.03677112e-10 3.33129479e-10 3.65438307e-10 4.00880633e-10
 4.39760361e-10 4.82410870e-10 5.29197874e-10 5.80522552e-10
  ......          ......
  1.72258597e-03 1.88965234e-03 2.07292178e-03 2.27396575e-03
  2.49450814e-03 2.73644000e-03 3.00183581e-03 3.29297126e-03
  3.61234270e-03 3.96268864e-03 4.34701316e-03 4.76861170e-03
  5.23109931e-03 5.73844165e-03 6.29498899e-03 6.90551352e-03
```

```
7.57525026e-03 8.30994195e-03 9.11588830e-03 1.00000000e-02]
```

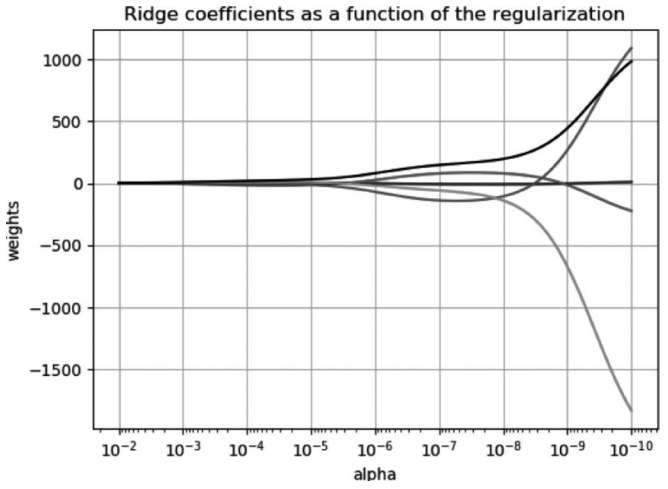

图 5-13　不同的 alpha 值对模型参数 weights 的影响程度

【例 5-9】 随机产生 100 组数据集，每组数据集包含 25 个点，每个点满足 $y=\sin\left(\frac{2}{\pi}x\right)+e$，其中 $x\in\{0.041\times i, i=1,2,\cdots,24\}$，$e$ 是添加的高斯噪声$(0,0.3^2)$。在每组数据集上用具有不同 λ 的七阶多项式进行岭回归拟合。

```
import numpy as np
import matplotlib.pyplot as plt
from tkinter import _flatten

x_arange = 0.041 * np.arange(0, 25, 1)                    #每组数据的 25 个点
y_True = np.sin(2 * np.pi * x_arange)                     #每个数据点对应的值(没有添加噪声)
y_Noise = np.zeros(y_True.shape)                          #添加噪声的值
x_Prec = np.linspace(0, 24*0.041, 100)                    #画图范围

mu = 0                                                    #噪声的 mu 值
sigma = 0.3                                               #噪声的 sigma 值
Num = 100                                                 #100 组数据集
n = 8                                                     #七阶多项式
lamda = [np.exp(1), np.exp(0), np.exp(-5), np.exp(-10)]   #不同的 lambda 值
phi = np.mat(np.zeros((x_arange.size, n)))                #phi 矩阵
x = np.mat(x_arange).T                                    #输入数据矩阵

#phi 矩阵运算
for i_n in range(n):
    for y_n in range(x_arange.size):
        phi[y_n, i_n] = x[y_n, 0] ** i_n

plt.figure(figsize=(15, 10))
index = 221
for i_lamda in lamda:
    plt.subplot(index)
    index += 1
    plt.title("lambda = %f" % i_lamda)
    plt.plot(x_Prec, np.sin(2 * np.pi * x_Prec), color='g')
    for k in range(Num):
        for i in range(x_arange.size):
            y_Noise[i] = y_True[i] + np.random.normal(mu, sigma)
        y = np.mat(y_Noise).T
```

```
        #求解 W 参数
        W = (phi.T * phi + i_lamda * np.eye(n)).I * phi.T * y

        ploy = list(_flatten(W.T.tolist()))
        ploy.reverse()
        p = np.poly1d(ploy)
        if k % 5 == 0:                                    #只画 20 条曲线
            plt.plot(x_Prec, p(x_Prec), color = 'r')
plt.show()
```

运行程序,效果如图 5-14 所示。

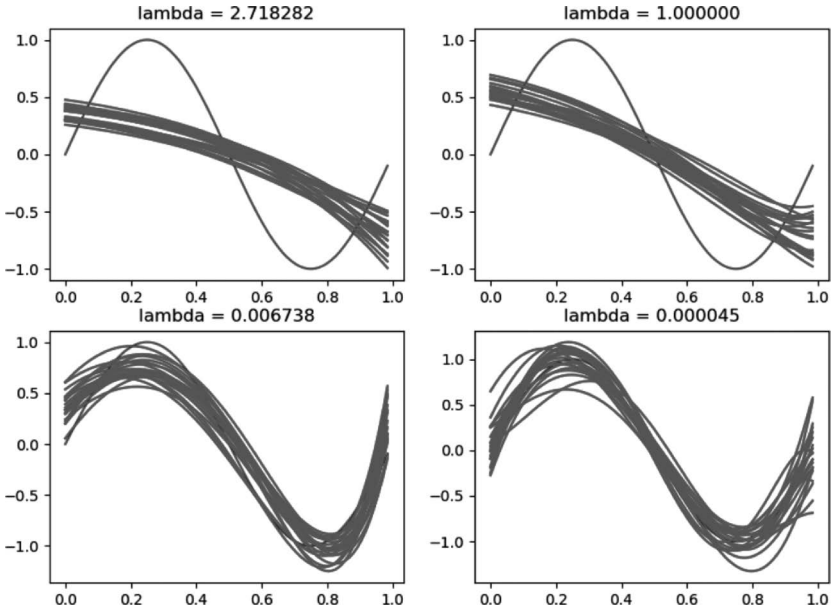

图 5-14　具有不同 λ 的七阶多项式进行岭回归拟合

2. 基于交叉验证的岭回归

在前面提到过,在岭回归中,alpha 的选择是一个比较麻烦的问题。这其实是一个模型选择的问题,在模型选择中,最简单的模型选择方法就是交叉验证(Cross-validation),将交叉验证内置在岭回归中,就免去了 alpha 的人工选择,其具体实现方式如下。

【例 5-10】 基于交叉验证的岭回归实现。

```
"""
基于交叉验证的岭回归 alpha 选择
可以直接获得一个相对不错的 alpha
"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn import linear_model

#这里设计矩阵 X 是一个希尔伯特矩阵(Hilbert matrix)
#其元素 A(i,j) = 1(i + j - 1),i 和 j 分别为其行标和列标
#希尔伯特矩阵是一种数学变换矩阵,正定且高度病态
#任何一个元素发生一点变动,整个矩阵的行列式的值和逆矩阵都会发生巨大变化
#这里设计矩阵是一个 10×5 的矩阵,即有 10 个样本,5 个变量
X = 1. / (np.arange(1, 6) + np.arange(0, 10)[:, np.newaxis])
y = np.ones(10)
print('设计矩阵为:')
print(X)
```

```
#初始化一个 Ridge Cross - Validation Regression
clf = linear_model.RidgeCV(fit_intercept = False)
#训练模型
clf.fit(X, y)
print
print('alpha 的数值 : ', clf.alpha_)
print('参数的数值:', clf.coef_)
```

运行程序,输出如下:

```
设计矩阵为:
[[1.          0.5         0.33333333 0.25        0.2        ]
 [0.5         0.33333333 0.25        0.2         0.16666667]
 [0.33333333 0.25        0.2         0.16666667 0.14285714]
 [0.25        0.2         0.16666667 0.14285714 0.125      ]
 [0.2         0.16666667 0.14285714 0.125       0.11111111]
 [0.16666667 0.14285714 0.125       0.11111111 0.1        ]
 [0.14285714 0.125       0.11111111 0.1         0.09090909]
 [0.125       0.11111111 0.1         0.09090909 0.08333333]
 [0.11111111 0.1         0.09090909 0.08333333 0.07692308]
 [0.1         0.09090909 0.08333333 0.07692308 0.07142857]]
alpha 的数值 :  0.1
参数的数值: [ - 0.43816548  1.19229228  1.54118834  1.60855632  1.58565451]
```

5.3.3 Lasso 回归模型

Lasso 回归模型采用的是 L1 正则,即 Lasso 是在平方误差的基础上增加 L1 正则:

$$l = \sum_{i=1}^{m} (y^{(i)} - \sum_{j=0}^{n} w_j x_j^{(i)})^2 + \lambda \sum_{j=0}^{n} |w_j|$$

其中,$\lambda > 0$。通过确定 λ 的值可以使得在方差和偏差之间达到平衡:随着 λ 的增大,模型方差减小而偏差增大。与基于 L2 正则的岭回归不同的是,上述的损失函数在 $w_j = 0$ 处是不可导的,因此传统的基于梯度的方法不能直接应用在上述的损失函数的求解上。为了求解这样的问题,一些近似的优化算法被采用,或者可以采用一些简单的方法来近似这样的优化过程。

【例 5-11】 利用 Lasso 回归模型实现数据拟合效果。

```
import numpy as np                                          #快速操作结构数组的工具
import matplotlib.pyplot as plt                             #可视化绘制
from sklearn.linear_model import Lasso,LassoCV,LassoLarsCV  #Lasso 回归,LassoCV 交叉验证实
                                                            #现 alpha 的选取,LassoLarsCV 基
                                                            #于最小角回归交叉验证实现
                                                            #alpha 的选取

#样本数据集,第一列为 x,第二列为 y,在 x 和 y 之间建立回归模型
data = [
    [0.067732,3.176513],[0.427810,3.816464],[0.995731,4.550095],[0.738336,4.256571],
    [0.981083,4.560815],[0.526171,3.929515],[0.378887,3.526170],[0.033859,3.156393],
    [0.132791,3.110301],[0.138306,3.149813],[0.247809,3.476346],[0.648270,4.119688],
    [0.731209,4.282233],[0.236833,3.486582],[0.969788,4.655492],[0.607492,3.965162],
    [0.358622,3.514900],[0.147846,3.125947],[0.637820,4.094115],[0.230372,3.476039],
    [0.070237,3.210610],[0.067154,3.190612],[0.925577,4.631504],[0.717733,4.295890],
    [0.015371,3.085028],[0.335070,3.448080],[0.040486,3.167440],[0.212575,3.364266],
    [0.617218,3.993482],[0.541196,3.891471]
    ]
#生成 x 和 y 矩阵
dataMat = np.array(data)
x = dataMat[:,0:1]
```

```
y = dataMat[:,1]

# ---------- Lasso 回归 -------
model = Lasso(alpha = 0.01)                    # 调节 alpha 可以实现对拟合的程度
#model = LassoCV()                             # LassoCV 自动调节 alpha 可以实现选择
                                               # 最佳的 alpha
#model = LassoLarsCV()                         # LassoLarsCV 自动调节 alpha 可以实现
                                               # 选择最佳的 alpha
model.fit(x, y)                                # 线性回归建模
print('系数矩阵:\n',model.coef_)
print('线性回归模型:\n',model)
#print('最佳的 alpha:',model.alpha_)           # 只有在使用 LassoCV、LassoLarsCV 时
                                               # 才有效
#使用模型预测
predicted = model.predict(x)
#绘制散点图 参数:x 横轴 y 纵轴
plt.scatter(x, y, marker = 'x')
plt.plot(X, predicted,c = 'r')
#绘制 x 轴和 y 轴坐标
plt.xlabel("x")
plt.ylabel("y")
#显示图形
plt.show()
```

运行程序,输出如下,效果如图 5-15 所示。

```
系数矩阵:
 [1.52826579]
线性回归模型:
 Lasso(alpha = 0.01, copy_X = True, fit_intercept = True, max_iter = 1000,
   normalize = False, positive = False, precompute = False, random_state = None,
   selection = 'cyclic', tol = 0.0001, warm_start = False)
```

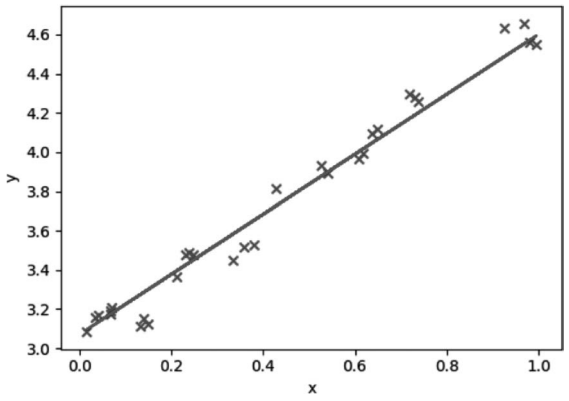

图 5-15　Lasso 回归效果

5.3.4　拟牛顿法

BFGS算法是使用较多的一种拟牛顿法,是由 Broyden、Fletcher、Goldfarb 和 Shanno 4 人分别提出的,故称为 BFGS 校正。

拟牛顿方程为

$$\nabla f(\boldsymbol{x}_k) = \nabla f(\boldsymbol{x}_{k+1}) + G_{k+1}(\boldsymbol{x}_k - \boldsymbol{x}_{k+1})$$

可化简为

$$G_{k+1}(\boldsymbol{x}_{k+1} - \boldsymbol{x}_k) = \nabla f(\boldsymbol{x}_{k+1}) - \nabla f(\boldsymbol{x}_k)$$

令 $\boldsymbol{B}_{k+1} \triangleq \boldsymbol{G}_{k+1}$，则可得

$$B_{k+1}(\boldsymbol{x}_{k+1}-\boldsymbol{x}_k)=\nabla f(\boldsymbol{x}_{k+1})-\nabla f(\boldsymbol{x}_k)$$

在 BFGS 校正方法中，假设

$$\boldsymbol{B}_{k+1}=\boldsymbol{B}_k+\boldsymbol{E}_k$$

1. BFGS 校正公式的推导

令 $\boldsymbol{E}_k=\alpha\boldsymbol{u}_k\boldsymbol{u}_k^{\mathrm{T}}+\beta\boldsymbol{v}_k\boldsymbol{v}_k^{\mathrm{T}}$，其中 $\boldsymbol{u}_k$、$\boldsymbol{v}_k$ 均为 $n\times 1$ 的向量。$\boldsymbol{y}_k=\nabla f(\boldsymbol{x}_{k+1})-\nabla f(\boldsymbol{x}_k)$，$\boldsymbol{s}_k=\boldsymbol{x}_{k+1}-\boldsymbol{x}_k$，则拟牛顿方程 $B_{k+1}(\boldsymbol{x}_{k+1}-\boldsymbol{x}_k)=\nabla f(\boldsymbol{x}_{k+1})-\nabla f(\boldsymbol{x}_k)$ 可以化简为

$$\boldsymbol{B}_{k+1}\boldsymbol{s}_k=\boldsymbol{y}_k$$

将 $\boldsymbol{B}_{k+1}=\boldsymbol{B}_k+\boldsymbol{E}_k$ 代入上式，即有

$$(\boldsymbol{B}_k+\boldsymbol{E}_k)\boldsymbol{s}_k=\boldsymbol{y}_k$$

将 $\boldsymbol{E}_k=\alpha\boldsymbol{u}_k\boldsymbol{u}_k^{\mathrm{T}}+\beta\boldsymbol{v}_k\boldsymbol{v}_k^{\mathrm{T}}$ 代入上式，则有

$$(\boldsymbol{B}_k+\alpha\boldsymbol{u}_k\boldsymbol{u}_k^{\mathrm{T}}+\beta\boldsymbol{v}_k\boldsymbol{v}_k^{\mathrm{T}})\boldsymbol{s}_k=\boldsymbol{y}_k$$

$$\Rightarrow\alpha(\boldsymbol{u}_k^{\mathrm{T}}\boldsymbol{s}_k)\boldsymbol{u}_k+\beta(\boldsymbol{v}_k^{\mathrm{T}}\boldsymbol{s}_k)\boldsymbol{v}_k=\boldsymbol{y}_k-\boldsymbol{B}_k\boldsymbol{s}_k$$

已知 $\boldsymbol{u}_k^{\mathrm{T}}\boldsymbol{s}_k$、$\boldsymbol{v}_k^{\mathrm{T}}\boldsymbol{s}_k$ 均为实数，$\boldsymbol{y}_k-\boldsymbol{B}_k\boldsymbol{s}_k$ 为 $n\times 1$ 的向量。上式中，参数 α 和 β 解的可能性有很多，取特殊的情况，假设 $\boldsymbol{u}_k=r\boldsymbol{B}_k\boldsymbol{s}_k$，$\boldsymbol{v}_k=\theta\boldsymbol{y}_k$，则

$$\boldsymbol{E}_k=\alpha r^2\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k+\beta\theta^2\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}$$

可以推出

$$\alpha[(r\boldsymbol{B}_k\boldsymbol{s}_k)](r\boldsymbol{B}_k\boldsymbol{s}_k)+\beta[(\theta\boldsymbol{y}_k)^{\mathrm{T}}\boldsymbol{s}_k](\theta\boldsymbol{y}_k)=\boldsymbol{y}_k-\boldsymbol{B}_k\boldsymbol{s}_k$$

即

$$[\alpha r^2(\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k\boldsymbol{s}_k)+1](\boldsymbol{B}_k\boldsymbol{s}_k)+[\beta\theta^2(\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k)-1](\boldsymbol{y}_k)=0$$

令 $\alpha r^2(\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k\boldsymbol{s}_k)+1=0$，即 $\beta\theta^2(\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k)-1=0$，则

$$\alpha r^2=-\frac{1}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k\boldsymbol{s}_k}$$

$$\beta\theta^2=\frac{1}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$$

最终的 BFGS 校正公式为

$$\boldsymbol{B}_{k+1}=\boldsymbol{B}_k-\frac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k\boldsymbol{s}_k}+\frac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$$

2. BFGS 校正的算法流程

设 $\boldsymbol{B}_k$ 对称正定，$\boldsymbol{B}_{k+1}$ 由上述的 BFGS 校正公式确定，那么 $\boldsymbol{B}_{k+1}$ 对称正定的充要条件是 $\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k>0$。

在利用 Armijo 搜索准则时，并不是都满足上述的充要条件，此时可以对 BFGS 校正公式做些改变：

$$\boldsymbol{B}_{k+1}=\begin{cases}\boldsymbol{B}_k, & \boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k\leqslant 0\\ \dfrac{\boldsymbol{B}_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k}{\boldsymbol{s}_k^{\mathrm{T}}\boldsymbol{B}_k\boldsymbol{s}_k}+\dfrac{\boldsymbol{y}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}, & \boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k>0\end{cases}$$

BFGS 拟牛顿的算法流程为：

(1) 初始化参数 $\delta\in(0,1)$，$\sigma\in(0,0.5)$，初始化点 x_0，终止误差 $0\leqslant\varepsilon\ll 1$，初始化对称正定

矩阵 $\boldsymbol{B}_0$。令 $k:=0$。

（2）重复以下过程。

① 计算 $\boldsymbol{g}_k-\nabla f(\boldsymbol{x}_k)$。如果 $\|\boldsymbol{g}_k\|\leqslant\varepsilon$，退出。输出 $\boldsymbol{x}_k$ 作为近似极小值点。

② 解线性方程组得解 $\boldsymbol{d}_k$：$\boldsymbol{B}_k\boldsymbol{d}=-\boldsymbol{g}_k$。

③ 设 m_k 是满足如下不等式的最小非负整数 m：

$$f(\boldsymbol{x}_k+\delta^m\boldsymbol{d}_k)\leqslant f(\boldsymbol{x}_k)+\sigma\delta^m\boldsymbol{g}_k^{\mathrm{T}}\boldsymbol{d}_k$$

令 $\alpha_k=\delta^{m_k}$，$\boldsymbol{x}_{k+1}=\boldsymbol{x}_k+\alpha_k\boldsymbol{d}_k$

④ 由上述公式确定 $\boldsymbol{B}_{k+1}$。

（3）令 $k:=k+1$。

利用 Sherman-Morrison 公式可对上式进行变换，得到：

$$\boldsymbol{B}_{k+1}^{-1}=\left(\boldsymbol{I}-\frac{\boldsymbol{s}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}\right)^{\mathrm{T}}\boldsymbol{B}_k^{-1}\left(I-\frac{\boldsymbol{y}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}\right)+\frac{\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$$

令 $\boldsymbol{H}_{k+1}=\boldsymbol{B}_{k+1}^{-1}$，则得到：

$$\boldsymbol{H}_{k+1}=\left(\boldsymbol{I}-\frac{\boldsymbol{s}_k\boldsymbol{y}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}\right)^{\mathrm{T}}\boldsymbol{H}_k\left(\boldsymbol{I}-\frac{\boldsymbol{y}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}\right)+\frac{\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$$

【例 5-12】 利用拟牛顿法求解无约束优化问题

$$\min_{\boldsymbol{x}=(t_1,t_2)\in\mathbf{R}^2}\ t_1^2+2t_2^2$$

$$\text{s.t.}\quad \boldsymbol{x}_0=(1,1)^{\mathrm{T}}$$

实现的 Python 代码为：

```
#基于 DFP 的拟牛顿法
import numpy as np
from numpy import linalg
import matplotlib.pyplot as plt

def compute_original_fun(x):
    """ 1. 计算原函数的值
    input:  x, 一个向量
    output: value, 一个值
    """
    value = x[0] ** 2 + 2 * x[1] ** 2
    return value

def compute_gradient(x):
    """ 2. 计算梯度
    input:  x, 一个向量
    output: value, 一个向量
    """
    value = np.mat([[0],[0]], np.double)
    value[0] = 2 * x[0]
    value[1] = 4 * x[1]
    return value

def draw_result(result):
    """ 3. 将收敛过程(即最小值的变化情况)画图 """
    plt.figure("min value")
    plt.plot(range(len(result)), result, "y", label = "min value")
    plt.title("min value's change")
    plt.legend()
```

```
    return plt

def main(x0, H, epsilon = 1e-6, max_iter = 1000):
    """
    x0: 初始迭代点
    H: 校正的对角正定矩阵
    eplison: 最小值上限
    max_iter: 最大迭代次数
    result: 最小值
    alpha ** m: 步长
    d: 方向
    """
    result = [compute_original_fun(x0)[0,0]]
    for k in range(max_iter):
        #计算梯度
        g = compute_gradient(x0)
        #终止条件
        if linalg.norm(g) < epsilon:
            break
        #计算搜索方向
        d = -H * g
        #简单线搜索求步长
        alpha = 1/2
        for m in range(max_iter):
            if compute_original_fun(x0 + alpha ** m * d) <= (compute_original_fun(x0) + (1/2) *
alpha ** m * g.T * d):
                break
        x = x0 + alpha ** m * d
        #DFP 校正迭代矩阵
        s = x - x0
        y = compute_gradient(x) - g
        if s.T * y > 0:
            H = H - (H * y * y.T * H)/(y.T * H * y) + (s * s.T)/(s.T * y)
        x0 = x
        result.append(compute_original_fun(x0)[0,0])
    return result

if __name__ == "__main__":
    x0 = np.asmatrix(np.ones((2,1)))
    H = np.asmatrix(np.eye(x0.size))
    result = main(x0, H)
    draw_result(result).show()
```

运行程序，效果如图 5-16 所示。

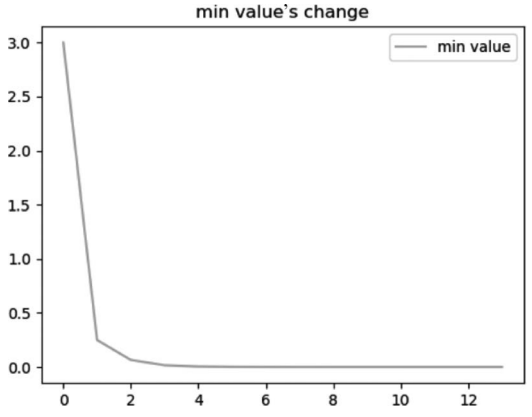

图 5-16　拟牛顿求解效果

5.3.5　L-BFGS 求解岭回归模型

1. BFGS 算法存在的问题

在 BFGS 算法中，每次都要存储近似海森矩阵 $\boldsymbol{B}_k^{-1}$，在高维数据时，存储 $\boldsymbol{B}_k^{-1}$ 将浪费很多存储空间，而在实际的运算过程中，需要的是搜索方向，因此出现了 L-BFGS 算法，这是 BFGS 算法的一种改进算法。在 L-BFGS 算法中，只保存最近的 m 次迭代信息，以减少数据的存储空间。

2. L-BFGS 算法思路

令 $\rho_k=\dfrac{1}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$，$\boldsymbol{V}_k=\boldsymbol{I}-\dfrac{\boldsymbol{y}_k\boldsymbol{s}_k^{\mathrm{T}}}{\boldsymbol{y}_k^{\mathrm{T}}\boldsymbol{s}_k}$，则 BFGS 算法中的 $\boldsymbol{H}_{k+1}$ 可以表示为

$$\boldsymbol{H}_{k+1}=\boldsymbol{V}_k^{\mathrm{T}}\boldsymbol{H}_k\boldsymbol{V}_k+\rho_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}$$

如果假定初始的矩阵 $\boldsymbol{H}_0=\boldsymbol{I}$，则可以得到：

$$\begin{aligned}\boldsymbol{H}_2&=\boldsymbol{V}_1^{\mathrm{T}}\boldsymbol{H}_1\boldsymbol{V}_1+\rho_1\boldsymbol{s}_1\boldsymbol{s}_1^{\mathrm{T}}\\&=\boldsymbol{V}_1^{\mathrm{T}}(\boldsymbol{V}_0^{\mathrm{T}}\boldsymbol{H}_0\boldsymbol{V}_0+\rho_0\boldsymbol{s}_0\boldsymbol{s}_0^{\mathrm{T}})\boldsymbol{V}_1+\rho_1\boldsymbol{s}_1\boldsymbol{s}_1^{\mathrm{T}}\\&=\boldsymbol{V}_1^{\mathrm{T}}\boldsymbol{V}_0^{\mathrm{T}}\boldsymbol{H}_0\boldsymbol{V}_0\boldsymbol{V}_1+\boldsymbol{V}_1^{\mathrm{T}}\rho_0\boldsymbol{s}_0\boldsymbol{s}_0^{\mathrm{T}}\boldsymbol{V}_1+\rho_1\boldsymbol{s}_1\boldsymbol{s}_1^{\mathrm{T}}\end{aligned}$$

则 $\boldsymbol{H}_{k+1}$ 为

$$\begin{aligned}\boldsymbol{H}_{k+1}=&(\boldsymbol{V}_k^{\mathrm{T}}\boldsymbol{V}_{k-1}^{\mathrm{T}}\cdots\boldsymbol{V}_1^{\mathrm{T}}\boldsymbol{V}_0^{\mathrm{T}})\boldsymbol{H}_0(\boldsymbol{V}_0\boldsymbol{V}_1\cdots\boldsymbol{V}_{k-1}\boldsymbol{V}_k)+(\boldsymbol{V}_k^{\mathrm{T}}\boldsymbol{V}_{k-1}^{\mathrm{T}}\cdots\boldsymbol{V}_1^{\mathrm{T}})\rho_1\boldsymbol{s}_1\boldsymbol{s}_1^{\mathrm{T}}(\boldsymbol{V}_1\cdots\boldsymbol{V}_{k-1}\boldsymbol{V}_k)+\cdots+\\&\boldsymbol{V}_k^{\mathrm{T}}\rho_{k-1}\boldsymbol{s}_{k-1}\boldsymbol{s}_{k-1}^{\mathrm{T}}\boldsymbol{V}_k+\rho_k\boldsymbol{s}_k\boldsymbol{s}_k^{\mathrm{T}}\end{aligned}$$

这样，在 L-BFGS 算法中，不再保存完整的 $\boldsymbol{H}_k$，而是存储向量序列 $\{\boldsymbol{s}_k\}$ 和 $\{\boldsymbol{y}_k\}$，需要矩阵 $\boldsymbol{H}_k$ 时，使用向量序列 $\{\boldsymbol{s}_k\}$ 和 $\{\boldsymbol{y}_k\}$ 计算就可以得到，而向量序列 $\{\boldsymbol{s}_k\}$ 和 $\{\boldsymbol{y}_k\}$ 也不是都要保存，只要保存最新的 m 步向量即可。L-BFGS 算法中确定新的下降方向的具体过程为：

- $d=-\nabla f(\boldsymbol{x}_k)$
- 令 $i=k-1:k-m$
  - ◎ $\alpha_i=\dfrac{\boldsymbol{s}_i\cdot p}{\boldsymbol{s}_i\cdot\boldsymbol{y}_i}$
  - ◎ $p=p-\alpha_i\cdot\boldsymbol{y}_i$
- $p=\left(\dfrac{\boldsymbol{s}_{k-1}\cdot\boldsymbol{y}_{k-1}}{\boldsymbol{y}_{k-1}\cdot\boldsymbol{y}_{k-1}}\right)$
- 令 $i=k-m:k-1$
  - ◎ $\beta=\dfrac{\boldsymbol{y}_i\cdot p}{\boldsymbol{y}_i\cdot\boldsymbol{y}_i}$
  - ◎ $p=p+(\alpha_i-\beta)\cdot\boldsymbol{s}_i$

【例 5-13】 利用岭回归对数据进行预测。

```
import numpy as np

def load_data(file_path):
    '''导入训练数据
    input:  file_path(string):训练数据
    output: feature(mat):特征
            label(mat):标签
    '''
    f = open(file_path)
    feature = []
```

```
    label = []
    for line in f.readlines():
        feature_tmp = []
        lines = line.strip().split("\t")
        feature_tmp.append(1)  #x0
        for i in range(len(lines) - 1):
            feature_tmp.append(float(lines[i]))
        feature.append(feature_tmp)
        label.append(float(lines[-1]))
    f.close()
    return np.mat(feature), np.mat(label).T

def ridge_regression(feature, label, lam):
    '''最小二乘的求解方法
    input:  feature(mat):特征
            label(mat):标签
    output: w(mat):回归系数
    '''
    n = np.shape(feature)[1]
    w = (feature.T * feature + lam * np.mat(np.eye(n))).I * feature.T * label
    return w

def get_gradient(feature, label, w, lam):
    '''
    input:  feature(mat):特征
            label(mat):标签
    output: w(mat):回归系数
    '''
    err = (label - feature * w).T
    left = err * (-1) * feature
    return left.T + lam * w

def get_result(feature, label, w, lam):
    '''
    input:  feature(mat):特征
            label(mat):标签
    output: w(mat):回归系数
    '''
    left = (label - feature * w).T * (label - feature * w)
    right = lam * w.T * w
    return (left + right) / 2

def get_error(feature, label, w):
    '''
    input:  feature(mat):特征
            label(mat):标签
    output: w(mat):回归系数
    '''
    m = np.shape(feature)[0]
    left = (label - feature * w).T * (label - feature * w)
    return (left / (2 * m))[0, 0]

def bfgs(feature, label, lam, maxCycle):
    '''利用 BFGS 训练岭回归模型
    input:  feature(mat):特征
            label(mat):标签
            lam(float):正则化参数
            maxCycle(int):最大迭代次数
    output: w(mat):回归系数
    '''
    n = np.shape(feature)[1]
```

```
    #1. 初始化
    w0 = np.mat(np.zeros((n, 1)))
    rho = 0.55
    sigma = 0.4
    Bk = np.eye(n)
    k = 1
    while (k < maxCycle):
        print("\titer: ", k, "\terror: ", get_error(feature, label, w0))
        gk = get_gradient(feature, label, w0, lam)  #计算梯度
        dk = np.mat(-np.linalg.solve(Bk, gk))
        m = 0
        mk = 0
        while (m < 20):
            newf = get_result(feature, label, (w0 + rho ** m * dk), lam)
            oldf = get_result(feature, label, w0, lam)
            if (newf < oldf + sigma * (rho ** m) * (gk.T * dk)[0, 0]):
                mk = m
                break
            m = m + 1

        #BFGS 校正
        w = w0 + rho ** mk * dk
        sk = w - w0
        yk = get_gradient(feature, label, w, lam) - gk
        if (yk.T * sk > 0):
            Bk = Bk - (Bk * sk * sk.T * Bk) / (sk.T * Bk * sk) + (yk * yk.T) / (yk.T * sk)

        k = k + 1
        w0 = w
    return w0

def lbfgs(feature, label, lam, maxCycle, m = 10):
    '''利用 L-BFGS 训练岭回归模型
    input:  feature(mat):特征
            label(mat):标签
            lam(float):正则化参数
            maxCycle(int):最大迭代次数
            m(int):L-BFGS 中选择保留的个数
    output: w(mat):回归系数
    '''
    n = np.shape(feature)[1]
    #1. 初始化
    w0 = np.mat(np.zeros((n, 1)))
    rho = 0.55
    sigma = 0.4
    H0 = np.eye(n)
    s = []
    y = []
    k = 1
    gk = get_gradient(feature, label, w0, lam)  #3X1
    print(gk)
    dk = -H0 * gk
    #2. 迭代
    while (k < maxCycle):
        print("iter: ", k, "\terror: ", get_error(feature, label, w0))
        m = 0
        mk = 0
        gk = get_gradient(feature, label, w0, lam)
        #2.1 Armijo 线搜索
        while (m < 20):
            newf = get_result(feature, label, (w0 + rho ** m * dk), lam)
```

```
            oldf = get_result(feature, label, w0, lam)
            if newf < oldf + sigma * (rho ** m) * (gk.T * dk)[0, 0]:
                mk = m
                break
            m = m + 1
        #2.2 L-BFGS 校正
        w = w0 + rho ** mk * dk
        #保留 m 个
        if k > m:
            s.pop(0)
            y.pop(0)
        #保留最新的
        sk = w - w0
        qk = get_gradient(feature, label, w, lam)  #3X1
        yk = qk - gk
        s.append(sk)
        y.append(yk)
        #two-loop
        t = len(s)
        a = []
        for i in range(t):
            alpha = (s[t - i - 1].T * qk) / (y[t - i - 1].T * s[t - i - 1])
            qk = qk - alpha[0, 0] * y[t - i - 1]
            a.append(alpha[0, 0])
        r = H0 * qk

        for i in range(t):
            beta = (y[i].T * r) / (y[i].T * s[i])
            r = r + s[i] * (a[t - i - 1] - beta[0, 0])
        if yk.T * sk > 0:
            print("update OK!!!!")
            dk = -r
        k = k + 1
        w0 = w
    return w0

def save_weights(file_name, w0):
    '''保存最终的结果
    input:  file_name(string):需要保存的文件
            w0(mat):权重
    '''
    f_result = open("weights", "w")
    m, n = np.shape(w0)
    for i in range(m):
        w_tmp = []
        for j in range(n):
            w_tmp.append(str(w0[i, j]))
        f_result.write("\t".join(w_tmp) + "\n")
    f_result.close()

if __name__ == "__main__":
    #1. 导入数据
    print("----------1.load data ----------")
    feature, label = load_data("data.txt")
    #2. 训练模型
    print("----------2.training ridge_regression ----------")
    method = "lbfgs"                                    #选择的方法
    if method == "bfgs":                                #选择 BFGS 训练模型
        w0 = bfgs(feature, label, 0.5, 1000)
    elif method == "lbfgs":                             #选择 L-BFGS 训练模型
        w0 = lbfgs(feature, label, 0.5, 1000, m=10)
```

```
    else:                                              #使用最小二乘的方法
        w0 = ridge_regression(feature, label, 0.5)
    #3. 保存最终的模型
    print("----------3.save model -------------")
    save_weights("weights", w0)
```

运行程序,输出如下:

```
----------1.load data -------------
----------2.training ridge_regression -------------
[[ -19745.73360913]
 [ -110149.14499117]
 [ -306327.80345624]]
iter:  1        error:  5165.1938312549155
update OK!!!!
iter:  2        error:  140.3163820505484
update OK!!!!
iter:  3        error:  137.40907856263757
……
iter:  997      error:  76.68074448016529
iter:  998      error:  76.68074448016529
iter:  999      error:  76.68074448016529
----------3.save model -------------
```

5.4　小结

回归通常是机器学习中使用的第一个算法。通过学习因变量和自变量之间的关系实现对数据的预测。本章通过线性回归、岭回归、Lasso 回归这几个方面介绍回归问题,每节通过理论与 Python 实践相结合,让读者快速掌握以 Python 解决回归问题的方法。

5.5　习题

1. BFGS 算法是使用较多的一种_____,是由_____、_____、_____和_____4 人分别提出的,故称为 BFGS 校正。

2. 牛顿法最突出的优点是_____,具有局部_____,但是,基本牛顿法初始点需要足够“靠近”_____,否则,有可能导致算法不收敛,此时就引入了全局牛顿法。

3. 岭回归和 Lasso 回归都属于_____的特征选择方法,对于处理较为复杂的_____通常选用这两种方法。

4. 典型的回归问题有哪些?

5. 利用 Python 编写代码实现经典线性回归模型。