

第八章 线性回归

8.1 线性回归的理论基础

8.1.1 基本形式

一 定义

给定由 d 个属性描述的示例 $\mathbf{x} = (x_1; x_2; \dots; x_d)$, 其中 x_i 是 \mathbf{x} 在第 i 个属性上的取值, 线性模型 (linear model) 试图学得一个通过属性的线性组合来进行预测的函数, 即

$$f(\mathbf{x}) = w_1 x_1 + w_2 x_2 + \dots + w_d x_d + b \quad (3.1)$$

向量形式:

$$f(\mathbf{x}) = \mathbf{w}^T \mathbf{x} + b \quad (3.2)$$

其中 $\mathbf{w} = (w_1; w_2; \dots; w_d)$. \mathbf{w} 和 b 学得之后, 模型就能确定下来.

二 线性模型的意义

线性模型是基础, 许多功能更为强大的非线性模型 (nonlinear model) 可在线性模型的基础上 通过引入 **层级结构** 或 **高维映射** 而得.

8.1.2 线性回归

给定数据集 $D = \{(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_m, y_m)\}$, 其中 $\mathbf{x}_i = (x_{i1}; x_{i2}; \dots; x_{id})$, $y_i \in \mathbb{R}$. "线性回归" (linear regression) 试图学得一个线性模型以尽可能准确地预测实值输出标记.

一 简单形式的线性回归

最简单的情形: 输入属性的数目只有一个. 此时忽略关于属性的下标, 即 $D = \{(x_i, y_i)\}_{i=1}^m$, 其中 $x_i \in \mathbb{R}$.

线性回归试图学得

$$f(x_i) = wx_i + b, \text{ 使得 } f(x_i) \simeq y_i \quad (3.3)$$

1 属性的转化

对离散属性, 若属性值间**存在"序" (order)关系**, 可通过连续化将其转化为**连续值**. "身高"的取值"高" "矮"可转化为 $\{1.0, 0.0\}$. "高" "中" "低"可转化为 $\{1.0, 0.5, 0.0\}$; 若属性值间**不存在序关系**, 假定有 k 个属性值, 则通常转化为 **k 维向量**, 如属性"瓜类"的取值"西瓜" "南瓜" "黄瓜"可转化为 $(0, 0, 1), (0, 1, 0), (1, 0, 0)$

2 均方误差最小化

试图让均方误差最小化, 即

$$\begin{aligned}
(w^*, b^*) &= \arg \min_{(w, b)} \sum_{i=1}^m (f(x_i) - y_i)^2 \\
&= \arg \min_{(w, b)} \sum_{i=1}^m (y_i - wx_i - b)^2
\end{aligned} \tag{3.4}$$

对3.4分别对 w 和 b 求导,得到

$$\frac{\partial E_{(w, b)}}{\partial w} = 2 \left(w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m (y_i - b) x_i \right) \tag{3.5}$$

$$\frac{\partial E_{(w, b)}}{\partial b} = 2 \left(mb - \sum_{i=1}^m (y_i - wx_i) \right) \tag{3.6}$$

最后求得最优的闭式(closed-form)解

$$w = \frac{\sum_{i=1}^m y_i (x_i - \bar{x})}{\sum_{i=1}^m x_i^2 - \frac{1}{m} \left(\sum_{i=1}^m x_i \right)^2} \tag{3.7}$$

$$b = \frac{1}{m} \sum_{i=1}^m (y_i - wx_i) \tag{3.8}$$

其中 $\bar{x} = \frac{1}{m} \sum_{i=1}^m x_i$ 为 x 的均值.

推导1:

$$w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m (y_i - b) x_i = 0 \tag{1}$$

$$mb = \sum_{i=1}^m (y_i - wx_i) \tag{2}$$

由2可以直接求得 b , 再将2带入到1中,有:

$$\begin{aligned}
&w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m \left(y_i - \frac{1}{m} \sum_{i=1}^m (y_i - wx_i) \right) x_i = 0 \\
&w \sum_{i=1}^m x_i^2 - \sum_{i=1}^m x_i y_i + \frac{1}{m} \sum_{i=1}^m \sum_{i=1}^m x_i (y_i - wx_i) = 0 \\
&w \sum_{i=1}^m x_i^2 - \frac{1}{m} \sum_{i=1}^m x_i \cdot \sum_{i=1}^m x_i \cdot w + \frac{1}{m} \sum_{i=1}^m x_i \cdot \sum_{i=1}^m y_i - \sum_{i=1}^m x_i y_i \\
&w \left(\sum_{i=1}^m x_i^2 - \frac{1}{m} \left(\sum_{i=1}^m x_i \right)^2 \right) = \sum_{i=1}^m x_i y_i - \bar{x} \cdot \sum_{i=1}^m y_i
\end{aligned}$$

则就可以求得:

$$w = \frac{\sum_{i=1}^m y_i (x_i - \bar{x})}{\sum_{i=1}^m x_i^2 - \frac{1}{m} \left(\sum_{i=1}^m x_i \right)^2}$$

二 一般形式

更一般的情形是如本节开头的数据集 D ，样本由 d 个属性描述.此时我们试图学得

$$f(\mathbf{x}_i) = \mathbf{w}^T \mathbf{x}_i + b, \text{使得 } f(\mathbf{x}_i) \simeq y_i,$$

注1: 这里的 $\mathbf{w}^T \mathbf{x}_i$ 和 $\mathbf{x}_i^T \mathbf{w}$ 是等价的.

这也就是"多元线性回归"(multivariate linear regression)

1 形式变换

为了简便起见, 把 \mathbf{w} 和 b 吸收成向量形式 $\hat{\mathbf{w}} = (\mathbf{w}; b)$, 相应的, 把数据集 D 表示为一个 $m \times (d+1)$ 大小的矩阵 \mathbf{X} , 其中每行对应于一个示例, 该行前 d 个元素对应于示例的 d 个属性值, 最后一个元素恒置为 1, 即

$$\mathbf{X} = \begin{pmatrix} x_{11} & x_{12} & \dots & x_{1d} & 1 \\ x_{21} & x_{22} & \dots & x_{2d} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_{m1} & x_{m2} & \dots & x_{md} & 1 \end{pmatrix} = \begin{pmatrix} \mathbf{x}_1^T & 1 \\ \mathbf{x}_2^T & 1 \\ \vdots & \vdots \\ \mathbf{x}_m^T & 1 \end{pmatrix}$$

再把标记也写成向量形式 $\mathbf{y} = (y_1; y_2; \dots; y_m)$, 则类似于式(3.4), 有

$$\hat{\mathbf{w}}^* = \arg \min_{\hat{\mathbf{w}}} (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}}) \quad (3.9)$$

注2: 3.9就是向量的平形式. 同时类似于注1, $\mathbf{X}\hat{\mathbf{w}}$ 展开其中一项, 其实就是 $\mathbf{x}_1^T \mathbf{w} + b$

2 求解过程

令 $E_{\hat{\mathbf{w}}} = (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$, 对 $\hat{\mathbf{w}}$ 求导得到

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 2\mathbf{X}^T (\mathbf{X}\hat{\mathbf{w}} - \mathbf{y}) \quad (3.10)$$

推导2: 将 $E_{\hat{\mathbf{w}}} = (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})^T (\mathbf{y} - \mathbf{X}\hat{\mathbf{w}})$ 展开可以得到:

$E_{\hat{\mathbf{w}}} = \mathbf{y}^T \mathbf{y} - \mathbf{y}^T \mathbf{X}\hat{\mathbf{w}} - \hat{\mathbf{w}}^T \mathbf{X}^T \mathbf{y} + \hat{\mathbf{w}}^T \mathbf{X}^T \mathbf{X}\hat{\mathbf{w}}$, 再对 $\hat{\mathbf{w}}$ 进行求导:

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = \frac{\partial \mathbf{y}^T \mathbf{y}}{\partial \hat{\mathbf{w}}} - \frac{\partial \mathbf{y}^T \mathbf{X}\hat{\mathbf{w}}}{\partial \hat{\mathbf{w}}} - \frac{\partial \hat{\mathbf{w}}^T \mathbf{X}^T \mathbf{y}}{\partial \hat{\mathbf{w}}} + \frac{\partial \hat{\mathbf{w}}^T \mathbf{X}^T \mathbf{X}\hat{\mathbf{w}}}{\partial \hat{\mathbf{w}}}$$

根据向量求导公式可得:

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 0 - \mathbf{X}^T \mathbf{y} - \mathbf{X}^T \mathbf{y} + (\mathbf{X}^T \mathbf{X} + \mathbf{X}^T \mathbf{X}) \hat{\mathbf{w}}$$

进一步得到:

$$\frac{\partial E_{\hat{\mathbf{w}}}}{\partial \hat{\mathbf{w}}} = 2\mathbf{X}^T (\mathbf{X}\hat{\mathbf{w}} - \mathbf{y})$$

注3: 矩阵求导相关参考资料

[矩阵求导-维基百科](#)

进一步, 当 $\mathbf{X}^T \mathbf{X}$ 为满秩矩阵或正定矩阵时, 令(3.10) 为零可以得到:

$$\hat{\mathbf{w}}^* = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (3.11)$$

令 $\hat{\mathbf{x}}_i = (\mathbf{x}_i, 1)$, 则最终的回归模型为:

$$f(\hat{x}_i) = \hat{x}_i^T (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T \mathbf{y} \quad (3.12)$$

而现实任务中 $\mathbf{X}^T \mathbf{X}$ 常常不是满秩矩阵, 常见的作法是引入正则化(regularization)项.

3 广义线性模型和联系函数

一般的, 考虑单调可微函数 $g(\cdot)$, 令

$$y = g^{-1}(w^T x + b) \quad (3.15)$$

这样的模型称为"广义线性模型"(generalized linear model), 其中函数 $g(\cdot)$ 称为"联系函数"(link function). 易见, 对数线性回归是广义线性模型在 $g(\cdot) = \ln(\cdot)$ 时的特例.

8.2 线性回归实战

8.2.1 线性回归的开发流程

- 收集数据: 采用任意方法收集数据
- 准备数据: 回归需要数值型数据, 标称型数据将被转换成二值型数据
- 分析数据: 绘出数据的可视化二维图将有助于对数据做出理解和分析, 在采用缩减法求得新回归系数之后, 可以将新拟合线绘在图上作为对比
- 训练算法: 找到回归系数
- 测试算法: 使用 R^2 或者预测值和数据的拟合度, 来分析模型的效果
- 使用算法: 使用回归, 可以在给定输入的时候预测出一个数值, 这是对分类方法的提升, 因为这样可以预测连续型数据而不仅仅是离散的类别标签

8.2.2 线性回归算法的特点

- 优点: 结果易于理解, 计算上不复杂.
- 缺点: 对非线性的数据拟合不好.
- 适用于数据类型: 数值型和标称型数据.

8.2.3 线性回归实战项目

一 简单线性回归项目

1 简单线性回归的工作原理

- 读入数据, 将数据特征 x 和特征标签 y 存储在矩阵 x 、 y 中
- 验证 $x^T x$ 矩阵是否可逆
- 使用最小二乘法求得回归系数 w 的最佳估计

2 简单线性回归项目实战

先看下数据ex0.txt的数据样式, 如下:

	↑	↗	ex...	用记事本打开	🔍	📁	↶	✕
1	1.000000	0.067732	3.176513					
2	1.000000	0.427810	3.816464					
3	1.000000	0.995731	4.550095					
4	1.000000	0.738336	4.256571					
5	1.000000	0.981083	4.560815					
6	1.000000	0.526171	3.929515					
7	1.000000	0.378887	3.526170					
8	1.000000	0.033859	3.156393					
9	1.000000	0.132791	3.110301					
10	1.000000	0.138306	3.149813					
11	1.000000	0.247809	3.476346					
12	1.000000	0.648270	4.119688					
13	1.000000	0.731209	4.282233					
14	1.000000	0.236833	3.486582					
15	1.000000	0.969788	4.655492					
16	1.000000	0.607492	3.965162					
17	1.000000	0.358622	3.514900					
18	1.000000	0.147846	3.125947					
19	1.000000	0.637820	4.094115					
20	1.000000	0.230372	3.476039					
21	1.000000	0.070237	3.210610					
22	1.000000	0.067154	3.190612					
23	1.000000	0.925577	4.631504					
24	1.000000	0.717733	4.295890					
25	1.000000	0.015371	3.085028					
26	1.000000	0.335070	3.448080					
27	1.000000	0.040486	3.167440					
28	1.000000	0.212575	3.364266					
29	1.000000	0.617218	3.993482					
30	1.000000	0.541196	3.891471					
31	1.000000	0.045353	3.143259					
32	1.000000	0.126762	3.114204					
33	1.000000	0.556486	3.851484					
34	1.000000	0.901144	4.621899					
35	1.000000	0.958476	4.580768					
36	1.000000	0.274561	3.620992					
37	1.000000	0.304306	3.580501					

注：第一列就是 x_0 ，全为1(原因就是我们的公式中的 x 第一列为了计算方便，设为1)。第二列 x_1 也就是 x ，第三列 x_3 也就是 y

编写的具体代码如下：

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

"""
函数说明：加载并解析数据
Parameters:
    filename - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""

def loadDataSet(fileName):
    numFeat = len((open(fileName).readline().split('\t')))
```

```

dataMat = []; labelMat = []
fr = open(fileName)
for line in fr.readlines():
    lineArr = []
    curLine = line.strip().split('\t')
    for i in range(numFeat - 1):
        lineArr.append(float(curLine[i]))
    dataMat.append(lineArr)
    labelMat.append(float(curLine[-1]))
return dataMat, labelMat

"""
函数说明：求解系数矩阵w
Parameters:
    xArr - 输入的样本(x数据集)
    yArr - 输入的对应该标签(y数据集)
Returns:
    ws - 回归系数
"""

def standRegres(xArr, yArr):
    #转换成矩阵
    xMat = np.mat(xArr)
    yMat = np.mat(yArr).T
    #计算 $X^T X$ 
    xTx = xMat.T * xMat
    #判断 $xTx$ 是否可逆，如果不可逆则无法计算
    #linalg.det()计算行列式
    if np.linalg.det(xTx) == 0:
        print('矩阵不可逆,无法求解逆矩阵')
        return
    #利用公式,计算w,  $xTx.I$ 计算 $xTx$ 的逆矩阵
    ws = xTx.I * (xMat.T * yMat)
    return ws

"""
函数说明：绘制回归曲线和数据点
Parameters:
    无
Returns:
    无
"""

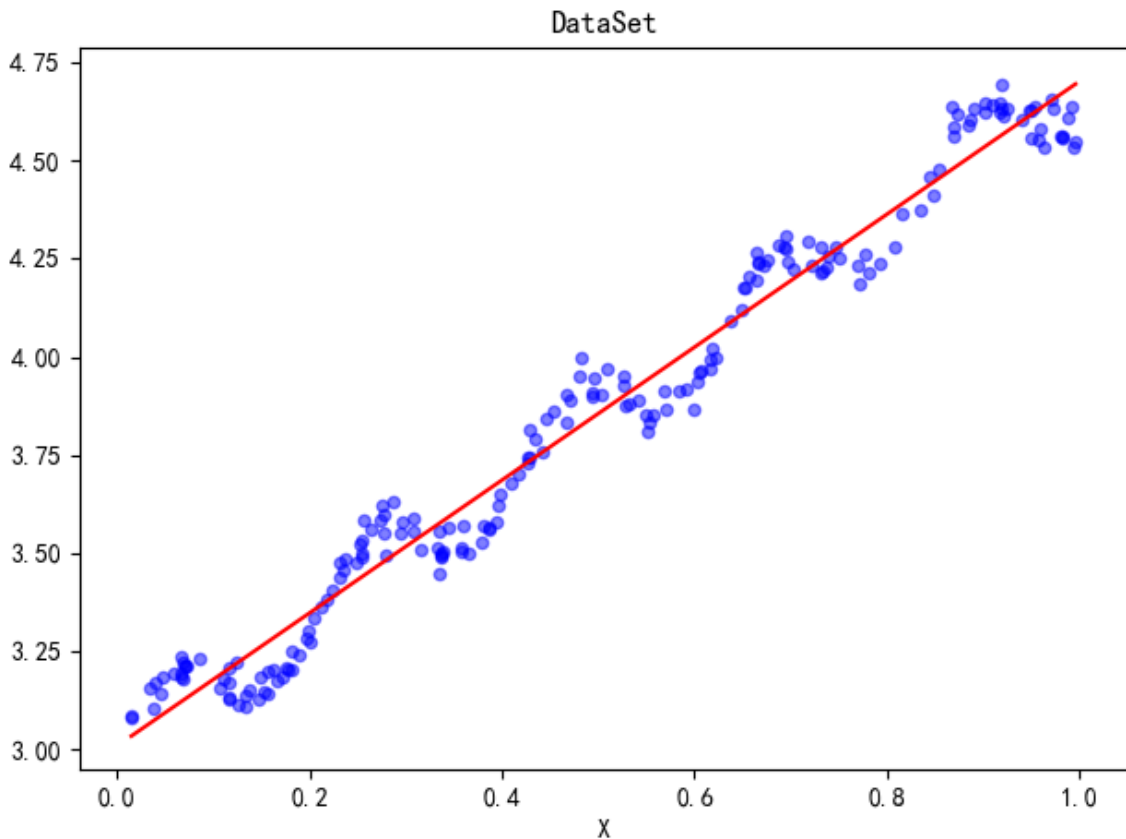
def plotRegression(xArr, yArr):
    ws = standRegres(xArr, yArr)
    xMat = np.mat(xArr)
    yMat = np.mat(yArr)
    xCopy = xMat.copy()
    xCopy.sort(0)
    yHat = xCopy * ws
    #关于matplotlib的绘图基础，参见<利用python进行数据分析>(第二版)
    #里面讲的非常好
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(xCopy[:, 1], yHat, c = 'red')
    #flatten是numpy.ndarray.flatten的一个函数，即返回一个折叠成一维的数组。
    #但是该函数只能适用于numpy对象，即array或者mat，普通的list列表是不行的
    ax.scatter(xMat[:, 1].flatten().A[0], yMat.flatten().A[0], s = 20, c =
'blue', alpha = 0.5)
    plt.title('DataSet')

```

```
plt.xlabel('x')
plt.show()
print(ws)

if __name__ == '__main__':
    xArr, yArr = loadDataSet('ch08\ex0.txt')
    plotRegression(xArr, yArr)
```

看下运行结果



求解的ws,如下:

```
Linear_Regression_Hsu01 x
H:\Anaconda3\python.exe D:/机器学习实战/Linear_Regression_Hsu01.py
[[3.00774324]
 [1.69532264]]

Process finished with exit code 0
```

几乎任一数据集都可以用上述方法建立模型,那么,如何判断这些模型的好坏呢?比较一下图8-3的两个子图,如果在两个数据集上分别作线性回归,将得到完全一样的模型(拟合直线)。显然两个数据是不一样的,那么模型分别在二者上的效果如何?我们当如何比较这些效果的好坏呢?有种方法可以计算预测值 \hat{y} 序列和真实值 y 序列的匹配程度,那就是计算这两个序列的相关系数。

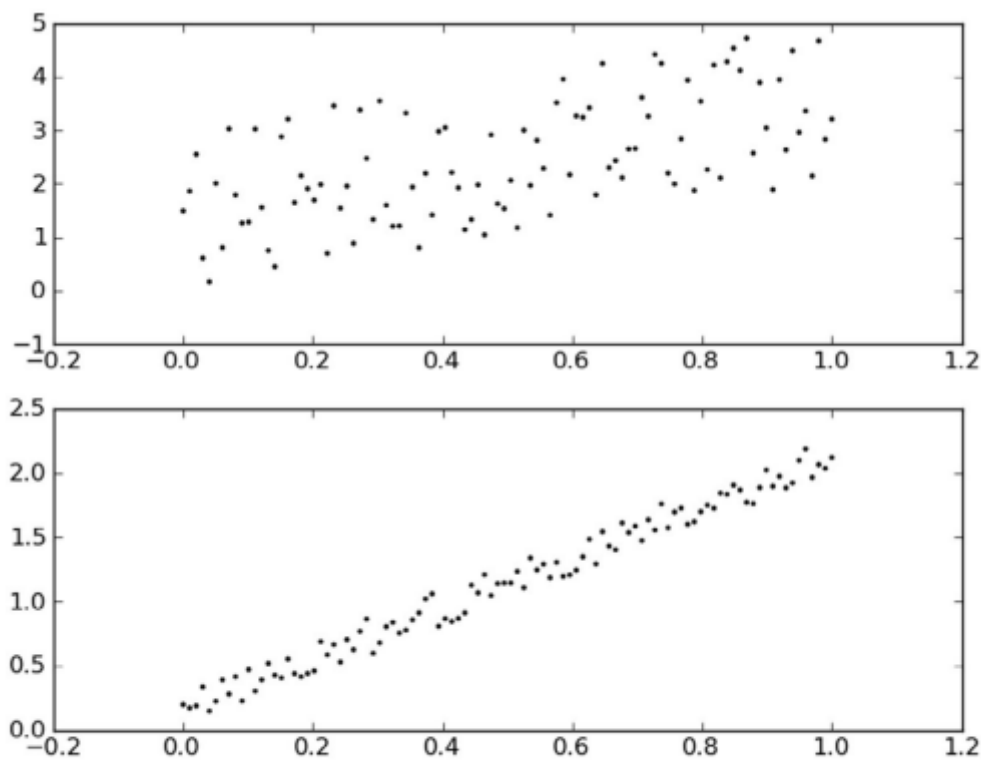


图8-3 具有相同回归系数（0和2.0）的两组数据。上图的相关系数是0.58，而下图的相关系数是0.99

把 `if __name__ == '__main__':` 改动一下即可计算相关系数. 具体如下:

```
if __name__ == '__main__':
    xArr, yArr = loadDataSet('ch08\ex0.txt')
    ws = standRegres(xArr, yArr)
    xMat = np.mat(xArr)
    yMat = np.mat(yArr)
    yHat = xMat * ws
    print(np.corrcoef(yHat.T, yMat))
```

结果如下:

```
if __name__ == '__main__':
    Linear_Regression_Hsu01 x
H:\Anaconda3\python.exe D:/机器学习实战/Linear_Regr
[[1.      0.98647356]
 [0.98647356 1.      ]]

Process finished with exit code 0
```

可以看到, 对角线上的数据是1.0, 因为yMat和自己的匹配是完全匹配的, 而YHat和yMat的相关系数为0.98, 效果也非常好.

二 局部加权线性回归

1 局部加权线性回归的基本理论

线性回归的一个问题是有可能出现欠拟合现象, 因为它求的是具有最小均方误差的无偏估计. 显而易见, 如果模型欠拟合将不能取得最好的预测效果. 所以有些方法允许在估计中引入一些偏差, 从而降低预测的均方误差.

其中的一个方法是局部加权线性回归 (Locally Weighted Linear Regression, LWLR), 在该算法中, 我们给待预测点附近的每个点赋予一定的权重; 然后与8.1节类似, 在这个子集上基于最小均方差来进行普通的回归, 与kNN一样, 这种算法每次预测均需要事先选取出对应的数据子集, 该算法解出回归系数 w 的形式如下:

$$\hat{w} = (\mathbf{X}^T \mathbf{W} \mathbf{X})^{-1} \mathbf{X}^T \mathbf{W} y$$

其中, \mathbf{W} 是一个矩阵, 用来给每个数据点赋予权重.

LWLR使用“核” (与支持向量机中的核类似) 来对附近的点赋予更高的权重, 核的类型可以自由选择, 最常用的核就是高斯核, 高斯核对应的权重如下:

$$w(i, i) = \exp\left(\frac{|x^{(i)} - x|}{-2k^2}\right)$$

这样就构建了一个只含对角元素的权重矩阵 \mathbf{W} , 并且点 x 与 $x(i)$ 越近, $w(i, i)$ 将会越大, 上述公式包含一个需要用户指定的参数 k , 它决定了对附近的点赋予多大的权重, 这也是使用LWLR时唯一需要考虑的参数, 在图8-4中可以看到参数 k 与权重的关系,

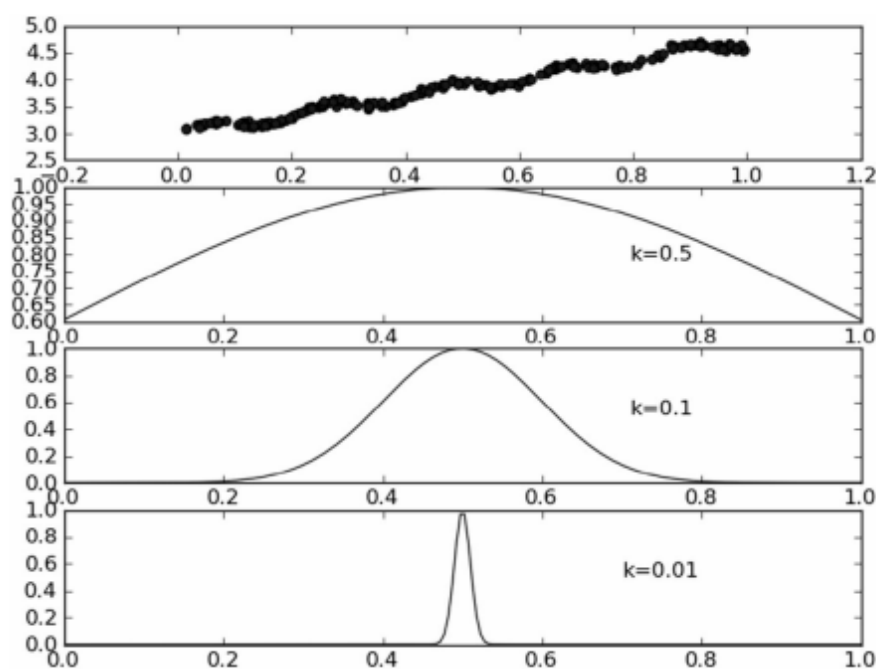


图8-4 每个点的权重图 (假定我们正预测的点是 $x = 0.5$), 最上面的图是原始数据集, 第二个图显示了当 $k = 0.5$ 时, 大部分的数据都用于训练回归模型; 而最下面的图显示当 $k = 0.01$ 时, 仅有很少的局部点被用于训练回归模型

2 局部加权线性回归的工作原理

- 读入数据, 将数据特征 和特征标签 y 存储在矩阵 x 、 y 中
- 利用高斯核构造一个权重矩阵 \mathbf{W} , 对预测点附近的点施加权重
- 验证 $\mathbf{X}^T \mathbf{W} \mathbf{X}$ 矩阵是否可逆
- 使用最小二乘法求得回归系数 w 的最佳估计

3 局部加权线性回归项目实战

数据还是上面的ex0.txt数据.

具体实现的代码如下:

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

"""
函数说明：加载并解析数据
Parameters:
    filename - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""

def loadDataSet(fileName):
    numFeat = len((open(fileName).readline().split('\t')))
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat - 1):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat, labelMat

"""
函数说明：使用局部加权线性回归计算回归系数w
            局部加权线性回归， 在待预测点附近的每个点赋予一定的权重， 在子集上基于最小均方差
            来进行普通的回归
Parameters:
    testPoint - 测试样本点
    xArr - 输入的样本(x数据集)
    yArr - 输入的对应该标签(y数据集)
    k - 高斯核的k
Returns:
    testPoint * ws - 数据点与具有权重的系数相乘得到的预测点
"""

def lwlr(testPoint, xArr, yArr, k = 1):
    #转换成矩阵
    xMat = np.mat(xArr)
    yMat = np.mat(yArr).T
    #获取xMat矩阵的行数
    m = np.shape(xMat)[0]
    #初始化权重(为对角矩阵)
    #eye()返回一个对角线元素为1， 其他元素为0的二维数组， 创建权重矩阵weights， 该矩阵为每个
    #样本点初始化了一个权重
    weights = np.mat(np.eye((m)))
    #遍历数据集， 计算每个样本的权重
    for j in range(m):
        #testPoint 的形式是 一个行向量的形式
        #计算 testPoint 与输入样本点之间的距离， 然后下面计算出每个样本贡献误差的权值
```

```

        diffMat = testPoint - xMat[j, :]
        weights[j,j] = np.exp(diffMat*diffMat.T/ (-2*k**2))
    xTx = xMat.T * (weights * xMat)
    if np.linalg.det(xTx) == 0:
        print("矩阵不可逆")
        return
    #计算出回归系数ws
    ws = xTx.I * (xMat.T * (weights * yMat))
    return testPoint * ws

"""
函数说明：局部加权线性回归测试
Parameters:
    testArr - 测试数据集
    xArr - 输入的样本(x数据集)
    yArr - 输入的对应标签(y数据集)
    k - 高斯核的k
Returns:
    yHat - 预测值
"""

def lwlrTest(testArr, xArr, yArr, k = 1):
    #得到样本点总数
    m = np.shape(testArr)[0]
    #初始化yHat，构建一个全部是 0 的 1 * m 矩阵
    yHat = np.zeros(m)
    #循环所有数据点，并应用lwlr函数
    for i in range(m):
        yHat[i] = lwlr(testArr[i], xArr, yArr, k)
    return yHat

def plotlwlrRegression():
    xArr, yArr = loadDataSet('CH08\ex0.txt')
    #加载数据集
    yHat_1 = lwlrTest(xArr, xArr, yArr, 1.0)
    #根据局部加权线性回归计算yHat
    yHat_2 = lwlrTest(xArr, xArr, yArr, 0.01)
    #根据局部加权线性回归计算yHat
    yHat_3 = lwlrTest(xArr, xArr, yArr, 0.003)
    #根据局部加权线性回归计算yHat
    xMat = np.mat(xArr)
    #创建xMat矩阵
    yMat = np.mat(yArr)
    #创建yMat矩阵
    srtInd = xMat[:, 1].argsort(0)
    #排序，返回索引值
    xSort = xMat[srtInd][:,0,:]
    fig, axs = plt.subplots(nrows=3, ncols=1,sharex=False, sharey=False,
figsize=(10,8))
    axs[0].plot(xSort[:, 1], yHat_1[srtInd], c = 'red')
    #绘制回归曲线
    axs[1].plot(xSort[:, 1], yHat_2[srtInd], c = 'red')
    #绘制回归曲线
    axs[2].plot(xSort[:, 1], yHat_3[srtInd], c = 'red')
    #绘制回归曲线
    axs[0].scatter(xMat[:,1].flatten().A[0], yMat.flatten().A[0], s = 20, c =
'blue', alpha = .5)
    #绘制样本点

```

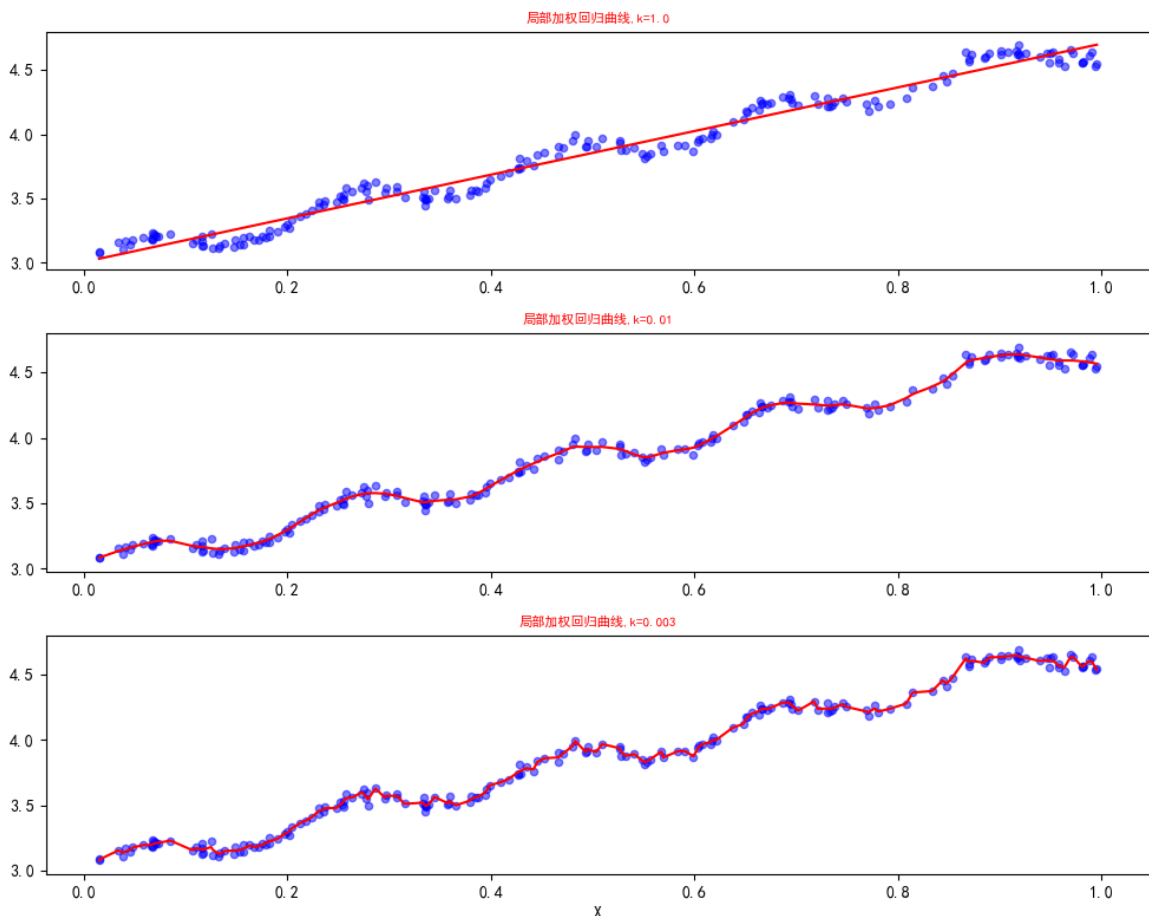
```

    axs[1].scatter(xMat[:,1].flatten().A[0], yMat.flatten().A[0], s = 20, c =
'blue', alpha = .5)
    #绘制样本点
    axs[2].scatter(xMat[:,1].flatten().A[0], yMat.flatten().A[0], s = 20, c =
'blue', alpha = .5)
    #绘制样本点
    #设置标题,x轴label,y轴label
    axs0_title_text = axs[0].set_title(u'局部加权回归曲线,k=1.0')
    axs1_title_text = axs[1].set_title(u'局部加权回归曲线,k=0.01')
    axs2_title_text = axs[2].set_title(u'局部加权回归曲线,k=0.003')
    plt.setp(axs0_title_text, size=8, weight='bold', color='red')
    plt.setp(axs1_title_text, size=8, weight='bold', color='red')
    plt.setp(axs2_title_text, size=8, weight='bold', color='red')
    plt.xlabel('x')
    plt.show()

if __name__ == '__main__':
    plotlwrRegression()

```

结果如下:



上图给出了 k 在三种不同取值下的结果图. 当 $k = 1.0$ 时权重很大, 如同将所有数据视为等权重, 得出的最佳拟合直线与标准的回归一致. 使用 $k = 0.01$ 得到了非常好的效果, 抓住了数据的潜在模式. 使用 $k = 0.003$ 纳入了太多的噪声点, 拟合的直线与数据点过于贴近. 所以, 上图中的最下图是过拟合的一个例子, 而最上图则是欠拟合的一个例子.

三 示例: 预测鲍鱼的年龄

有一份来自 UCI 的数据集合的数据, 记录了鲍鱼 (一种介壳类水生动物) 的年龄. 鲍鱼年龄可以从鲍鱼壳的层数推算得到.

先看下数据格式:

具体代码如下:

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

"""
函数说明: 加载并解析数据
Parameters:
    filename - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""

def loadDataSet(fileName):
    numFeat = len((open(fileName).readline().split('\t')))
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat - 1):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat, labelMat

"""
函数说明: 求解系数矩阵w
Parameters:
    xArr - 输入的样本(x数据集)
    yArr - 输入的对应该签(y数据集)
Returns:
    ws - 回归系数
"""

def standRegres(xArr, yArr):
    #转换成矩阵
    xMat = np.mat(xArr)
    yMat = np.mat(yArr).T
    #计算 $x^T x$ 
    xTx = xMat.T * xMat
    #判断 $xTx$ 是否可逆, 如果不可逆则无法计算
    #linalg.det() 计算行列式
    if np.linalg.det(xTx) == 0:
        print('矩阵不可逆, 无法求解逆矩阵')
        return
    #利用公式, 计算w
    ws = xTx.I * (xMat.T * yMat)
    return ws
```

```
"""
```

函数说明：使用局部加权线性回归计算回归系数 w

局部加权线性回归，在待预测点附近的每个点赋予一定的权重，在子集上基于最小均方差来进行普通的回归

Parameters:

testPoint - 测试样本点
xArr - 输入的样本(x 数据集)
yArr - 输入的对应该标签(y 数据集)
k - 高斯核的 k

Returns:

testPoint * ws - 数据点与具有权重的系数相乘得到的预测点

```
"""
```

```
def lwlr(testPoint, xArr, yArr, k = 1):
```

```
    #转换成矩阵
```

```
    xMat = np.mat(xArr)
```

```
    yMat = np.mat(yArr).T
```

```
    #获取xMat矩阵的行数
```

```
    m = np.shape(xMat)[0]
```

```
    #初始化权重(为对角矩阵)
```

#eye() 返回一个对角线元素为1，其他元素为0的二维数组，创建权重矩阵**weights**，该矩阵为每个样本点初始化了一个权重

```
    weights = np.mat(np.eye((m)))
```

```
    #遍历数据集，计算每个样本的权重
```

```
    for j in range(m):
```

```
        #testPoint 的形式是 一个行向量的形式
```

```
        #计算 testPoint 与输入样本点之间的距离，然后下面计算出每个样本贡献误差的权值
```

```
        diffMat = testPoint - xMat[j, :]
```

```
        weights[j,j] = np.exp(diffMat*diffMat.T/ (-2*k**2))
```

```
    xTx = xMat.T * (weights * xMat)
```

```
    if np.linalg.det(xTx) == 0:
```

```
        print("矩阵不可逆")
```

```
        return
```

```
    #计算出回归系数ws
```

```
    ws = xTx.I * (xMat.T * (weights * yMat))
```

```
    return testPoint * ws
```

```
"""
```

函数说明：局部加权线性回归测试

Parameters:

testArr - 测试数据集
xArr - 输入的样本(x 数据集)
yArr - 输入的对应该标签(y 数据集)
k - 高斯核的 k

Returns:

yHat - 预测值

```
"""
```

```
def lwlrTest(testArr, xArr, yArr, k = 1):
```

```
    #得到样本点总数
```

```
    m = np.shape(testArr)[0]
```

```
    #初始化yHat，构建一个全部是 0 的 1 * m 矩阵
```

```
    yHat = np.zeros(m)
```

```
    #循环所有数据点，并应用lwlr函数
```

```
    for i in range(m):
```

```
        yHat[i] = lwlr(testArr[i], xArr, yArr, k)
```

```
    return yHat
```

```
"""
```

函数说明：返回真实值与预测值误差大小

```

Parameters:
    yArr - 样本的真实值
    yHatArr - 样本的预测值
Returns:
    误差
"""
def rssError(yArr, yHatArr):
    return ((yArr - yHatArr)**2).sum()

"""
函数说明：预测鲍鱼的年龄
Parameters:
    无
Returns:
    无
"""
def abaloneTest():
    abx, aby = loadDataSet('CH08/abalone.txt')
    print('训练集与测试集相同：局部加权线性回归，核参数k的大小对预测的影响：')
    #不同的k进行预测
    oldyHat01 = lwlrTest(abx[0:99], abx[0:99], aby[0:99], 0.1)
    oldyHat1 = lwlrTest(abx[0:99], abx[0:99], aby[0:99], 1)
    oldyHat10 = lwlrTest(abx[0:99], abx[0:99], aby[0:99], 10)
    # 打印出不同的核预测值与新数据集(测试数据集)上的真实值之间的误差大小
    print('k=0.1时,误差大小为:', rssError(aby[0:99], oldyHat01.T))
    print('k=1 时,误差大小为:', rssError(aby[0:99], oldyHat1.T))
    print('k=10 时,误差大小为:', rssError(aby[0:99], oldyHat10.T))

    print("训练集与测试集不同：局部加权线性回归，和参数k的大小对预测的影响：")
    newyHat01 = lwlrTest(abx[100:199], abx[0:99], aby[0:99], 0.1)
    newyHat1 = lwlrTest(abx[100:199], abx[0:99], aby[0:99], 1)
    newyHat10 = lwlrTest(abx[100:199], abx[0:99], aby[0:99], 10)
    print('k=0.1时,误差大小为:', rssError(aby[100:199], newyHat01.T))
    print('k=1 时,误差大小为:', rssError(aby[100:199], newyHat1.T))
    print('k=10 时,误差大小为:', rssError(aby[100:199], newyHat10.T))

    print("训练集与测试集不同:简单的线性回归与k=1时的局部加权线性回归对比:")
    print("k=1时,局部加权线性回归的误差大小为：", rssError(aby[100:199], newyHat1.T))

    ws = standRegres(abx[0:99], aby[0:99])
    linearYhat = np.mat(abx[100:199]) * ws
    #注意，array可以和list计算(结果为array)，但是array和matrix不可以，因此需要.A转换成
    array
    print("简单线性回归误差大小为：", rssError(aby[100:199], linearYhat.T.A))

if __name__ == '__main__':
    abaloneTest()

```

这里还有一点常常遇到的一个东西,我觉得挺重要的,就是array,list和matrix之间的计算

array可以和list计算(结果为array), 但是array和matrix不可以, 因此需要.A转换成array

运行结果为:

```
Linear_Regression_Hsu01 x
↑
↓
训练集与测试集相同：局部加权线性回归，核参数k的大小对预测的影响：
k=0.1时，误差大小为：56.78420911837208
k=1 时，误差大小为：429.89056187030394
k=10 时，误差大小为：549.1181708826065
训练集与测试集不同：局部加权线性回归，和参数k的大小对预测的影响：
k=0.1时，误差大小为：25119.459111157415
k=1 时，误差大小为：573.5261441895706
k=10 时，误差大小为：517.5711905381745
训练集与测试集不同：简单的线性回归与k=1时的局部加权线性回归对比：
k=1时，局部加权线性回归的误差大小为：573.5261441895706
简单线性回归误差大小为：518.6363153249365

Process finished with exit code 0
```

四 缩减系数来“理解”数据

如果数据的特征比样本点还多应该怎么办？是否还可以使用线性回归和之前的方法来做预测？答案是否定的，即不能再使用前面介绍的方法。

这是因为在计算 $(\mathbf{X}^T \mathbf{X})^{-1}$ 的时候会出错。如果特征比样本点还多 ($n > m$)，也就是说输入数据的矩阵 \mathbf{X} 不是满秩矩阵。非满秩矩阵在求逆时会出现问题。

为解决这个问题，统计学家引入了岭回归(ridge regression)的概念，接着是lasso法，该方法效果很好但是计算复杂。

1 岭回归

简单说来，岭回归就是在矩阵 $\mathbf{X}^T \mathbf{X}$ 上加一个 $\lambda \mathbf{I}$ 从而使得矩阵非奇异，进而能对 $\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I}$ 求逆。其中矩阵 \mathbf{I} 是一个 $m \times m$ 的单位矩阵，对角线上元素全为1，其他元素全为0。而 λ 是一个用户定义的数值，后面会做介绍。在这种情况下，回归系数的计算公式将变成：

$$\hat{w} = (\mathbf{X}^T \mathbf{X} + \lambda \mathbf{I})^{-1} \mathbf{X}^T y$$

岭回归最先用来处理特征数多于样本数的情况，现在也用于**在估计中加入偏差，从而得到更好的估计**。这里通过引入 λ 来限制了所有 w 之和，通过引入该惩罚项，能够减少不重要的参数，这个技术在统计学中也叫做缩减shrinkage

缩减方法可以去掉不重要的参数，因此能更好地理解数据。此外，与简单的线性回归相比，缩减法能取得更好的预测效果。

具体代码如下：

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



```
"""
```

函数说明：加载并解析数据

Parameters:

filename - 文件名

Returns:

dataMat - 数据矩阵

labelMat - 数据标签

```
"""
```

```
def loadDataSet(fileName):
```

```
    numFeat = len((open(fileName).readline().split('\t')))
```

```
    dataMat = []; labelMat = []
```

```
    fr = open(fileName)
```

```
    for line in fr.readlines():
```

```
        lineArr = []
```

```
        curLine = line.strip().split('\t')
```

```
        for i in range(numFeat - 1):
```

```
            lineArr.append(float(curLine[i]))
```

```
        dataMat.append(lineArr)
```

```
        labelMat.append(float(curLine[-1]))
```

```
    return dataMat, labelMat
```

```
def ridgeRegres(xMat, yMat, lam=0.2):
```

```
    """
```

岭回归

:param xMat: x数据集

:param yMat: y数据集

:param lam: 缩减系数

:return: ws-回归系数

```
    """
```

```
    xTx = xMat.T * xMat
```

```
    #岭回归就是在矩阵 xTx 上加一个  $\lambda I$  从而使得矩阵非奇异，进而能对  $xTx + \lambda I$  求逆
```

```
    denom = xTx + np.eye(np.shape(xMat)[1]) * lam
```

```
    # 检查行列式是否为零，即矩阵是否可逆，行列式为0的话就不可逆，不为0的话就是可逆
```

```
    if np.linalg.det(denom) == 0:
```

```
        print("矩阵是奇异矩阵，不可逆")
```

```
        return
```

```
    ws = denom.I * (xMat.T * yMat)
```

```
    return ws
```

```
def ridgeTest(xArr, yArr):
```

```
    """
```

岭回归的测试

:param xArr:x数据集

:param yArr:y数据集

:return:wMat - 回归系数矩阵

```
    """
```

```
    xMat = np.mat(xArr)
```

```
    yMat = np.mat(yArr).T
```

```
    #中心化
```

```
    yMean = np.mean(yMat, 0)
```

```
    yMat = yMat - yMean
```

```
    xMeans = np.mean(xMat, 0)
```

```
    xVar = np.var(xMat, 0)
```

```
    xMat = (xMat - xMeans) / xVar
```

```
    numTests = 30
```

```
    wMat = np.zeros((numTests, np.shape(xMat)[1]))
```

```
    for i in range(numTests):
```

```

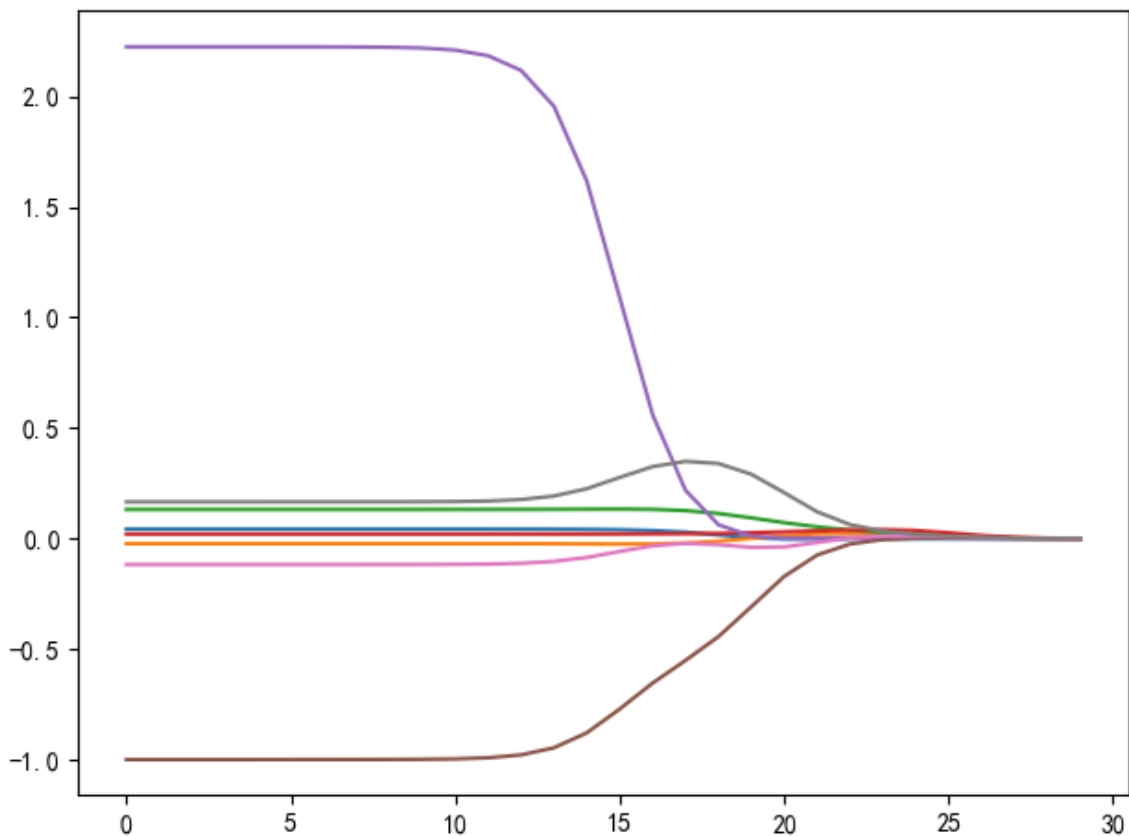
ws = ridgeRegres(xMat, yMat, np.exp(i-10))
wMat[i,:] = ws.T
return wMat

def regression_ridge():
    xArr, yArr = loadDataSet('CH08/abalone.txt')
    ridgeWeights = ridgeTest(xArr,yArr)
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.plot(ridgeWeights)
    plt.show()

if __name__ == '__main__':
    regression_ridge()

```

运行结果如下:



2 套索方法

在增加如下约束时, 普通的最小二乘法回归会得到与岭回归一样的公式:

$$\sum_{k=1}^n w_k^2 \leq \lambda$$

上式限定了所有回归系数的平方和不能大于 λ . 使用普通的最小二乘法回归在当两个或更多的特征相关时, 可能会得到一个很大的正系数和一个很大的负系数. 正是因为上述限制条件的存在, 使用岭回归可以避免这个问题.

与岭回归类似, 另一个缩减方法lasso也对回归系数做了限定, 对应的约束条件如下:

$$\sum_{k=1}^n |w_k| \leq \lambda$$

唯一的不同点在于, 这个约束条件使用绝对值取代了平方和. 虽然约束形式只是稍作变化, 结果却大相径庭: 在 λ 足够小的时候, 一些系数会因此被迫缩减到 0. 这个特性可以帮助我们更好地理解数据.

关于lasso方法, 实现起来很复杂, 然后书上有有个简单版的, 就是奇纳香逐步回归算法. 它可以得到与lasso差不多的效果. 是一种贪心算法, 即每一步都尽可能减少误差. 一开始, 所有权重都设置为 0, 然后每一步所做的决策是对某个权重增加或减少一个很小的值.

伪代码如下:

```
数据标准化, 使其分布满足 0 均值 和单位方差
在每轮迭代过程中:
    设置当前最小误差 lowestError 为正无穷
    对每个特征:
        增大或缩小:
            改变一个系数得到一个新的 w
            计算新 w 下的误差
            如果误差 Error 小于当前最小误差 lowestError: 设置 wbest 等于当前的 w
    将 w 设置为新的 wbest
```

虽然说起来简单, 但我不想敲了, 后续再说吧.

五 权衡偏差和方差

任何时候, 一旦发现模型和测量值之间存在差异, 就说出现了误差. 当考虑模型中的“噪声”或者说误差时, 必须考虑其来源.

- 在处理过程种, 可能会对**复杂的过程进行简化**, 这将导致在模型和测量值之间出现“噪声”或**误差**, 若无法理解数据的真实生成过程, 也会导致差异的发生.
- 另外, **测量过程本身也可能产生“噪声”或者问题**.

比如: 简单线性回归和局部加权线性回归中的数据ex0.txt是作者通过随机产生的一个数据, 具体生成公式为: $y = 3.0 + 1.7x + 0.1\sin(30x) + 0.06N(0,1)$

其中, $N(0,1)$ 是一个均值为0, 方差为1的正态分布. 对于简单线性回归来说, 拟合的就是 $3.0 + 0.7x$ 这部分, 而误差部分就是 $0.1\sin(30x) + 0.06N(0,1)$. 而局部加权线性回归通过多组不同的局部权重来找到具有最小误差的解.

下图给出了**训练误差**和**测试误差**的曲线图, **上面的曲线就是测试误差**, **下面的曲线是训练误差**. 根据第三小节局部加权线性回归的实验我们知道: **如果降低核的大小, 那么训练误差将变小**. 从图中来看, **从左到右就表示了核逐渐减小的过程**.

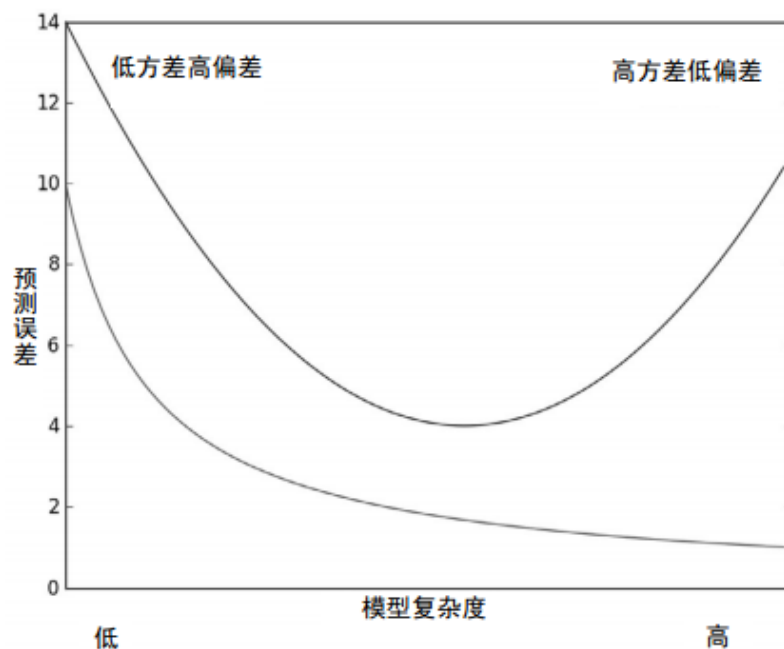


图8-8 偏差方差折中与测试误差及训练误差的关系。上面的曲线就是测试误差，在中间部分最低。为了做出最好的预测，我们应该调整模型复杂度来达到测试误差的最小值

一般认为, 上述两种误差由三个部分组成: **偏差**、**测量误差**和**随机噪声**. 在8.2节和8.3节, 我们通过引入了三个越来越小的核来不断增大模型的方差

在缩减系数来“理解”数据这一节中, 介绍了缩减法, 可以将一些**系数缩减成很小的值或直接缩减为 0**, 这是一个**增大模型偏差**的例子 (也就是上图的左侧). 通过把一些特征的回归系数缩减到 0, 同时也就减小了模型的复杂度. 例子中有 8 个特征, 消除其中两个后不仅使模型更易理解, 同时还降低了预测误差. 对照上图, **左侧是参数缩减过于严厉的结果, 而右侧是无缩减的效果**.

方差是可以度量的. 如果从鲍鱼数据中取一个随机样本集 (例如取其中 100 个数据) 并用线性模型拟合, 将会得到一组回归系数. 同理, 再取出另一组随机样本集并拟合, 将会得到另一组回归系数. 这些系数间的差异大小也就是模型方差的反映.

8.3 使用sklearn实现简单线性回归

sklearn中有比较全的一般线性回归模型, 比如简单线性回归, 岭回归, lasso回归, 贝叶斯回归等等. 这里只简单介绍下简单线性回归LinearRegression模块.

sklearn.linear_model: Generalized Linear Models

The `sklearn.linear_model` module implements generalized linear models. It includes Ridge regression, Bayesian Regression, Lasso and Elastic Net estimators computed with Least Angle Regression and coordinate descent. It also implements Stochastic Gradient Descent related algorithms.

User guide: See the [Generalized Linear Models](#) section for further details.

<code>linear_model.ARDRegression</code> ([n_iter, tol, ...])	Bayesian ARD regression.
<code>linear_model.BayesianRidge</code> ([n_iter, tol, ...])	Bayesian ridge regression.
<code>linear_model.ElasticNet</code> ([alpha, l1_ratio, ...])	Linear regression with combined L1 and L2 priors as regularizer.
<code>linear_model.ElasticNetCV</code> ([l1_ratio, eps, ...])	Elastic Net model with iterative fitting along a regularization path.
<code>linear_model.HuberRegressor</code> ([epsilon, ...])	Linear regression model that is robust to outliers.
<code>linear_model.Lars</code> ([fit_intercept, verbose, ...])	Least Angle Regression model a.k.a.
<code>linear_model.LarsCV</code> ([fit_intercept, ...])	Cross-validated Least Angle Regression model.
<code>linear_model.Lasso</code> ([alpha, fit_intercept, ...])	Linear Model trained with L1 prior as regularizer (aka the Lasso)
<code>linear_model.LassoCV</code> ([eps, n_alphas, ...])	Lasso linear model with iterative fitting along a regularization path.
<code>linear_model.LassoLars</code> ([alpha, ...])	Lasso model fit with Least Angle Regression a.k.a.
<code>linear_model.LassoLarsCV</code> ([fit_intercept, ...])	Cross-validated Lasso, using the LARS algorithm.
<code>linear_model.LassoLarsIC</code> ([criterion, ...])	Lasso model fit with Lars using BIC or AIC for model selection
<code>linear_model.LinearRegression</code> ([...])	Ordinary least squares Linear Regression.
<code>linear_model.LogisticRegression</code> ([penalty, ...])	Logistic Regression (aka logit, MaxEnt) classifier.
<code>linear_model.LogisticRegressionCV</code> ([Cs, ...])	Logistic Regression CV (aka logit, MaxEnt) classifier.
<code>linear_model.MultiTaskLasso</code> ([alpha, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
<code>linear_model.MultiTaskElasticNet</code> ([alpha, ...])	Multi-task ElasticNet model trained with L1/L2 mixed-norm as regularizer
<code>linear_model.MultiTaskLassoCV</code> ([eps, ...])	Multi-task Lasso model trained with L1/L2 mixed-norm as regularizer.
<code>linear_model.MultiTaskElasticNetCV</code> ([...])	Multi-task L1/L2 ElasticNet with built-in cross-validation.
<code>linear_model.OrthogonalMatchingPursuit</code> ([...])	Orthogonal Matching Pursuit model (OMP)
<code>linear_model.OrthogonalMatchingPursuitCV</code> ([...])	Cross-validated Orthogonal Matching Pursuit model (OMP).
<code>linear_model.PassiveAggressiveClassifier</code> ([...])	Passive Aggressive Classifier
<code>linear_model.PassiveAggressiveRegressor</code> ([C, ...])	Passive Aggressive Regressor
<code>linear_model.Perceptron</code> ([penalty, alpha, ...])	Read more in the User Guide .
<code>linear_model.RANSACRegressor</code> ([...])	RANSAC (RANDOM SAMPLE Consensus) algorithm.
<code>linear_model.Ridge</code> ([alpha, fit_intercept, ...])	Linear least squares with l2 regularization.
<code>linear_model.RidgeClassifier</code> ([alpha, ...])	Classifier using Ridge regression.
<code>linear_model.RidgeClassifierCV</code> ([alphas, ...])	Ridge classifier with built-in cross-validation.
<code>linear_model.RidgeCV</code> ([alphas, ...])	Ridge regression with built-in cross-validation.
<code>linear_model.SGDClassifier</code> ([loss, penalty, ...])	Linear classifiers (SVM, logistic regression, a.o.) with SGD training.
<code>linear_model.SGDRegressor</code> ([loss, penalty, ...])	Linear model fitted by minimizing a regularized empirical loss with SGD
<code>linear_model.TheilSenRegressor</code> ([...])	Theil-Sen Estimator: robust multivariate regression model.
<code>linear_model.enet_path</code> (X, y[, l1_ratio, ...])	Compute elastic net path with coordinate descent
<code>linear_model.lars_path</code> (X, y[, Xy, Gram, ...])	Compute Least Angle Regression or Lasso path using LARS algorithm [1]
<code>linear_model.lars_path_gram</code> (Xy, Gram, n_samples)	lars_path in the sufficient stats mode [1]
<code>linear_model.lasso_path</code> (X, y[, eps, ...])	Compute Lasso path with coordinate descent
<code>linear_model.orthogonal_mp</code> (X, y[, ...])	Orthogonal Matching Pursuit (OMP)
<code>linear_model.orthogonal_mp_gram</code> (Gram, Xy[, ...])	Gram Orthogonal Matching Pursuit (OMP)
<code>linear_model.ridge_regression</code> (X, y, alpha[, ...])	Solve the ridge equation by the method of normal equations.

8.3.1 linear_model.LinearRegression()

sklearn.linear_model.LinearRegression

```
class sklearn.linear_model. LinearRegression (fit_intercept=True, normalize=False, copy_X=True, n_jobs=None)  
[source]
```

Ordinary least squares Linear Regression.

Parameters:	fit_intercept : <i>boolean, optional, default True</i> whether to calculate the intercept for this model. If set to False, no intercept will be used in calculations (e.g. data is expected to be already centered).
	normalize : <i>boolean, optional, default False</i> This parameter is ignored when <code>fit_intercept</code> is set to False. If True, the regressors X will be normalized before regression by subtracting the mean and dividing by the l2-norm. If you wish to standardize, please use <code>sklearn.preprocessing.StandardScaler</code> before calling <code>fit</code> on an estimator with <code>normalize=False</code> .
	copy_X : <i>boolean, optional, default True</i> If True, X will be copied; else, it may be overwritten.
	n_jobs : <i>int or None, optional (default=None)</i> The number of jobs to use for the computation. This will only provide speedup for <code>n_targets > 1</code> and sufficient large problems. <code>None</code> means 1 unless in a <code>joblib.parallel_backend</code> context. <code>-1</code> means using all processors. See Glossary for more details.
Attributes:	coef : <i>array, shape (n_features,) or (n_targets, n_features)</i> Estimated coefficients for the linear regression problem. If multiple targets are passed during the fit (y 2D), this is a 2D array of shape (n_targets, n_features), while if only one target is passed, this is a 1D array of length n_features.
	intercept_ : <i>array</i> Independent term in the linear model.

参数说明:

LinearRegression中的主要参数如下:

- **fit_intercept**: 是否需要截距, bool类型, 默认为True. 也就是是否求解b
- **normalize**: 是否先进行归一化, bool类型, 默认为False. 如果为真, 则回归X将在回归之前被归一化. 当fit_intercept设置为False时, 将忽略此参数. 当回归量归一化时, 注意到这使得超参数学习更加鲁棒, 并且几乎不依赖于样本的数量. 相同的属性对标准化数据无效. 然而, 如果你想标准化, 请在调用 `normalize = False` 训练估计器之前, 使用 `preprocessing.StandardScaler` 处理数据.
- **copy_X**: 是否复制X数组, bool类型, 默认为True, 如果为True, 将复制X数组; 否则, 它覆盖原数组X.
- **n_jobs**: 是否启动所有CPU, int类型, 默认值为1

主要方法:

同时, LinearRegression中还有一些方法, 具体如下:

Methods

fit (self, X, y[, sample_weight])	Fit linear model.
get_params (self[, deep])	Get parameters for this estimator.
predict (self, X)	Predict using the linear model
score (self, X, y[, sample_weight])	Returns the coefficient of determination R^2 of the prediction.
set_params (self, **params)	Set the parameters of this estimator.

对于 8.1 小节, 我们可以重写成:

```
import numpy as np  
import matplotlib.pyplot as plt  
from sklearn.linear_model import LinearRegression
```

```

"""
函数说明：加载并解析数据
Parameters:
    filename - 文件名
Returns:
    dataMat - 数据矩阵
    labelMat - 数据标签
"""

def loadDataSet(fileName):
    numFeat = len((open(fileName).readline().split('\t')))
    dataMat = []; labelMat = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat - 1):
            lineArr.append(float(curLine[i]))
        dataMat.append(lineArr)
        labelMat.append(float(curLine[-1]))
    return dataMat, labelMat

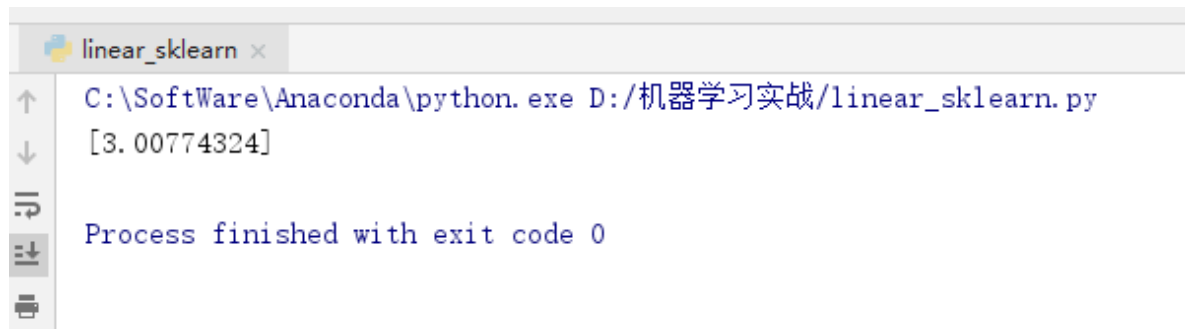
def sklearnLinear(xArr, yArr):
    # 转换成矩阵
    xMat = np.mat(xArr)
    yMat = np.mat(yArr).T
    clf = LinearRegression()
    clf.fit(xMat, yMat)
    a = clf.intercept_
    print(a)

if __name__ == '__main__':
    xArr, yArr = loadDataSet('CH08\ex0.txt')
    sklearnLinear(xArr, yArr)

```

sklearnLinear函数中, 我返回了截距项, 即 `clf.intercept_`

结果如下:



```

linear_sklearn x
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/linear_sklearn.py
[3.00774324]
Process finished with exit code 0

```

可以看出和书上是一致的.

最后提一点

关于是否有截距的问题, 如果有截距的话, 我们就把 b 收进 w 中, 那么向量 w 就是 $(w_0; w_1; \dots; w_d)$. 对应的 (x_0, x_1, \dots, x_d) . 此时, $x_0=1$. 目的就是为了简化操作.

所以, 可以看到数据集 `ex0.txt` 和 `ex1.txt` 本身第一列都已经是1了, 直接求解得到 w_0 , 而对于鲍鱼的数据集 `abalone.txt` 第一列不是1, 回归得来的就没有截距项, 如果想要截距项, 那么需要对原来的数据进行处理, 第一列设置成全为1. (这是针对书上的算法来说的, 书上就是根据公式来的)

用书上写的算法, 如果数据集第一列全是1的话, 那么默认是有截距项的, 如果第一列不是1, 那么默认是没有截距项的.

因为如果全是1, 那么根据公式就知道, $w_0 \cdot x_0$, x_0 恒为1, w_0 是我们算出的参数, 是定值, 所以, w_0 就是截距项,

$w_0=b$

如果用 `sklearn.linear_model` 中的 `LinearRegression` 模块, 里面有参数 `fit_intercept` 可以设置有无截距项, 默认为 `True`.

然后, 我用了分别用书上和 `sklearn` 做了验证, 我先对数据进行了处理, 把 `ex0.txt` 第一列(这一列全是1)删除, 得到一个新的文件 `ex0 - 副本.txt`. `ex0.txt` 和 `ex0 - 副本.txt` 文件格式分别如下:

ex0.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

1.000000	0.067732	3.176513
1.000000	0.427810	3.816464
1.000000	0.995731	4.550095
1.000000	0.738336	4.256571
1.000000	0.981083	4.560815
1.000000	0.526171	3.929515
1.000000	0.378887	3.526170
1.000000	0.033859	3.156393
1.000000	0.132791	3.110301
1.000000	0.138306	3.149813
1.000000	0.247809	3.476346
1.000000	0.648270	4.119688
1.000000	0.731209	4.282233
1.000000	0.236833	3.486582
1.000000	0.969788	4.655492
1.000000	0.607492	3.965162
1.000000	0.358622	3.514900
1.000000	0.147846	3.125947
1.000000	0.637820	4.094115
1.000000	0.230372	3.476039

ex0 - 副本.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

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

1. 先用书上的算法 `standRegress` 函数, 结果分别如下:


```
if __name__ == '__main__':
    xArr, yArr = loadDataSet('CH08\ex0.txt')
    ws1 = standRegres(xArr, yArr)
    plotRegression(xArr, yArr)
    print(ws1)
    xArr_1, yArr_1 = loadDataSet('CH08\ex0 - 副本.txt')
    ws2 = standRegres(xArr_1, yArr_1)
    plotRegression(xArr_1, yArr_1)
    print(ws2)
```

plotRegression()

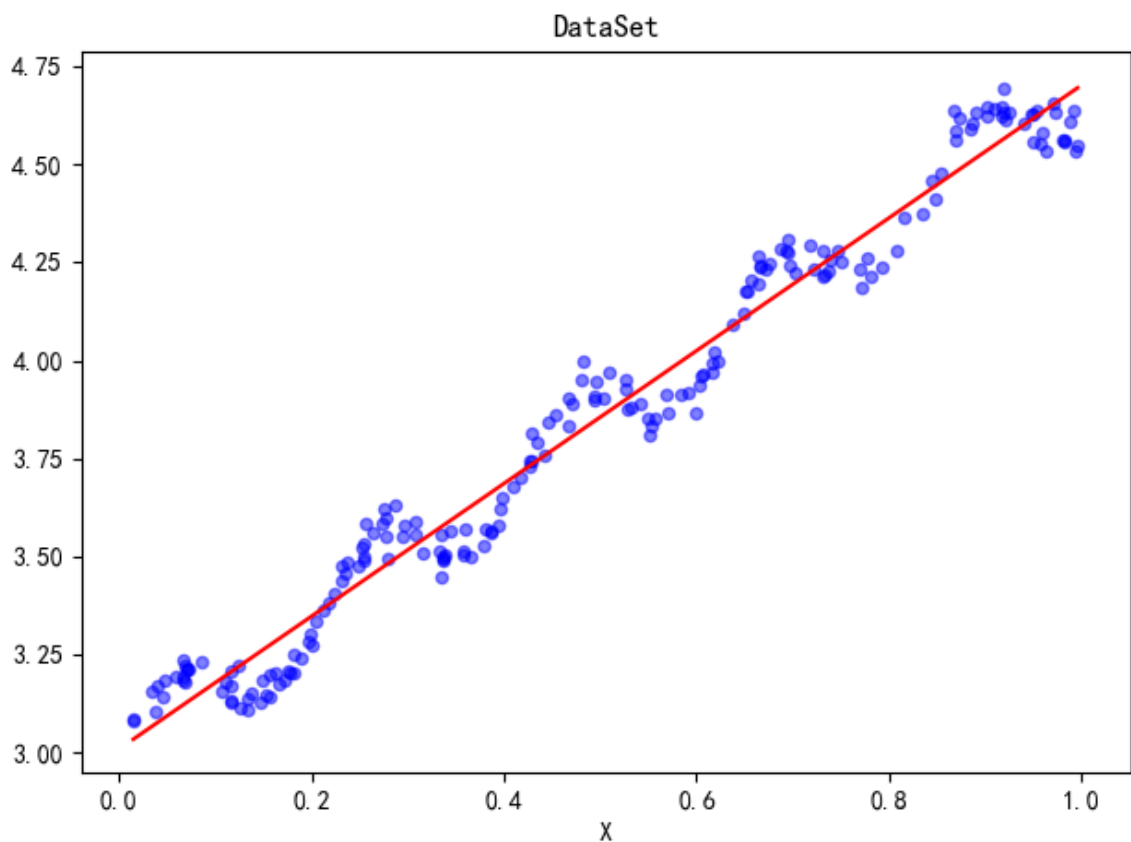
Linear_Regression_Hsu01 x

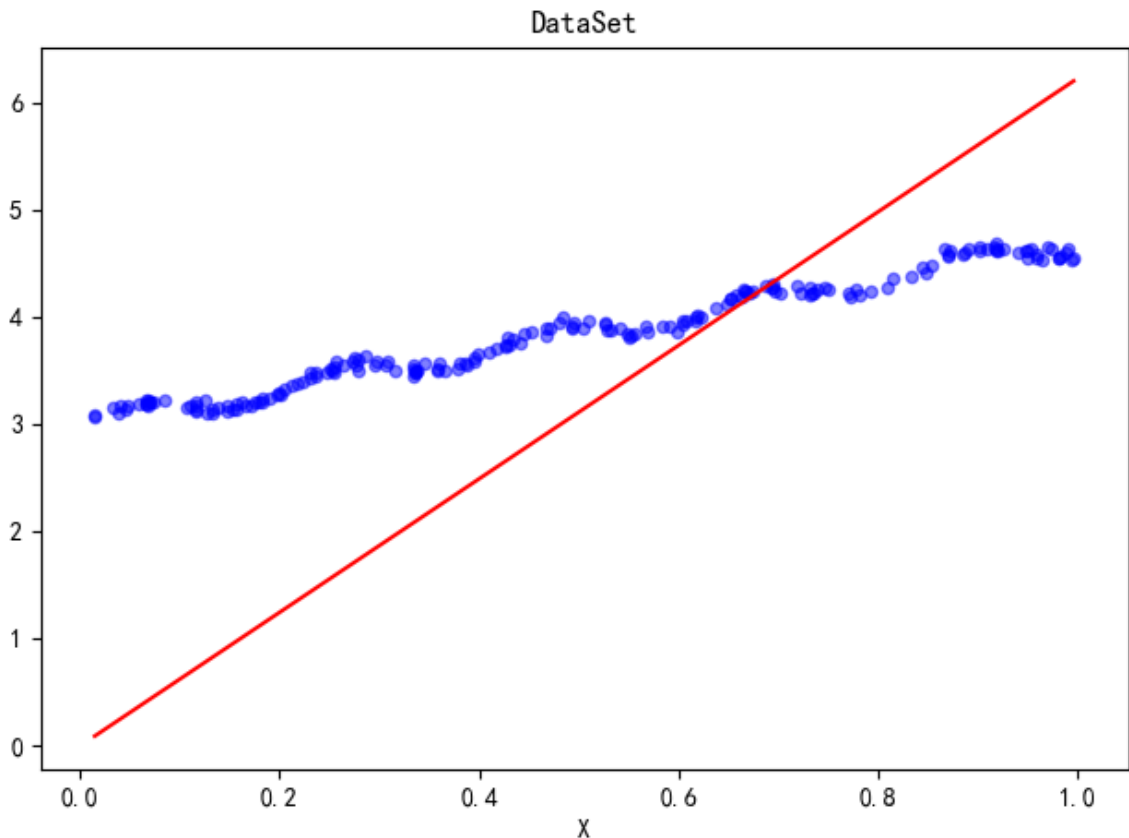
C:\SoftWare\Anaconda\python.exe D:/机器学习实战/Linear_Regression_Hsu01.py

```
[[3.00774324]
 [1.69532264]]
[[6.23067797]]
```

Process finished with exit code 0

可以看到, ws1就是书上的, 有截距项, 为3.00774324, 而ws2只有一个w, 没有截距项, 画出的图分别如下:





可以看出差别还是很大的, 所以, 针对书上的standRegress函数(也就是计算ws)来说, 如果数据集实际是有截距项的, 那么第一列必须是1, 不然, 结果很明显是不正确的.

2. 用sklearn模块, 其中默认是有截距项的, 即参数 `fit_intercept=True`.

```
def sklearnLinear(xArr, yArr):
    # 转换成矩阵
    xMat = np.mat(xArr)
    yMat = np.mat(yArr).T
    clf = LR()
    clf.fit(xMat, yMat)
    a = clf.intercept_
    print(a)

if __name__ == '__main__':
    xArr, yArr = loadDataSet('CH08\ex0.txt')
    sklearnLinear(xArr, yArr)
    xArr_1, yArr_1 = loadDataSet('CH08\ex0 - 副本.txt')
    sklearnLinear(xArr_1, yArr_1)
```

linear_sklearn x

C:\SoftWare\Anaconda\python.exe D:/机器学习实战/linear_sklearn.py
[3.00774324]
[3.00774324]

Process finished with exit code 0

可以看到, 都可以得到正确的结果, 为什么这样, 我考虑是这样的, 如果默认有截距项, 那么对于ex0 - 副本.txt来说, sklearn在处理时就已经把第一项全为1考虑进去了(即加了第一列全为1), 显然能得到结果; 而对于ex0.txt来说, 也是一样, 也会把第一项全为1考虑进去(也加了第一列全为1), 但是这时原来的第一列变成第二列, 而第二列还时为1, 最后的结果就是 w_0, w_1 都是截距项, 合并一起, 肯定时一样的结果了.

总结来说, 书上的已经把截距项收纳进去了, 一起计算出 w , 如果原数据集第一列都为1, 那么最后的 w s的第一项就是截距项, 第一列不是全为1, 那么 w s最后就没有截距项.

而对于sklearn来说, 这些不成问题, 他已经先期处理过了, 考虑进去过了