

第3章 多元线性回归

内容提要

□ 线性回归

□ 局部加权线性回归

□ 岭回归和逐步线性回归

□ 预测鲍鱼年龄和玩具售价

回归的一般方法

1. 收集数据：采用任意方法收集数据。
2. 准备数据：回归需要数值型数据，标称型数据将被转成二值型数据。
3. 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。
4. 训练算法：找到回归系数。
5. 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。
6. 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。

应当怎样从一大堆数据里求出回归方程呢？假定输入数据存放在矩阵 \mathbf{X} 中，而回归系数存放在向量 \mathbf{w} 中。那么对于给定的数据 \mathbf{X}_1 ，预测结果将会通过 $\mathbf{Y}_1 = \mathbf{X}_1^T \mathbf{w}$ 给出。现在的问题是，手里有一些 \mathbf{X} 和对应的 y ，怎样才能找到 \mathbf{w} 呢？一个常用的方法就是找出使误差最小的 \mathbf{w} 。这里的误差是指预测 y 值和真实 y 值之间的差值，使用该误差的简单累加将使得正差值和负差值相互抵消，所以我们采用平方误差。

平方误差可以写做：

$$SS = \sum_{i=1}^m (y_i - x_i^T \mathbf{w})^2 \quad (3.1)$$

用矩阵表示还可以写做 $(y - \mathbf{X}\mathbf{w})^T (y - \mathbf{X}\mathbf{w})$ 。如果对 \mathbf{w} 求导，得到 $\mathbf{X}^T (y - \mathbf{X}\mathbf{w})$ ，令其等于零，解出 \mathbf{w} 如下：

$$\hat{\mathbf{w}} = (\mathbf{X}^T \mathbf{X})^{-1} \mathbf{X}^T y \quad (3.2)$$

值得注意的是，上述公式中包含 $(\mathbf{X}^T \mathbf{X})^{-1}$ ，也就是需要对矩阵求逆，因此这个方程只在逆矩阵存在的时候适用。然而，矩阵的逆可能并不存在，因此必须要在代码中对此作出判断。

上述的最佳 \mathbf{w} 求解是统计学中的常见问题，除了矩阵方法外还有很多其他方法可以解决。通过调用 NumPy 库里的矩阵方法，我们可以仅使用几行代码就完成所需功能。该方法也称作 OLS。

下面看看实际效果，对于图3.1中的散点图，下面来介绍如何给出该数据的最佳拟合直线。

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

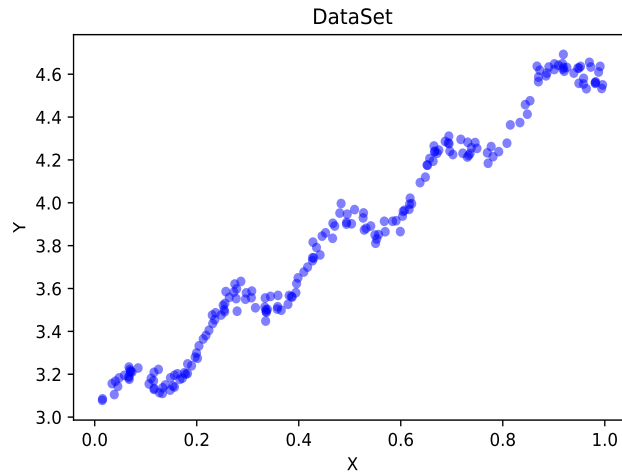


图 3.1: 样例数据图

```
# 加载数据
def loadDataSet(fileName):
    """
    Parameters:
        fileName - 文件名
    Returns:
        xArr - x数据集
        yArr - y数据集
    """
    numFeat = len(open(fileName).readline().split('\t')) - 1
    xArr = []; yArr = []
    fr = open(fileName)
    for line in fr.readlines():
        lineArr = []
        curLine = line.strip().split('\t')
        for i in range(numFeat):
            lineArr.append(float(curLine[i]))
        xArr.append(lineArr)
        yArr.append(float(curLine[-1]))
    return xArr, yArr

# 绘制数据集
def plotDataSet():
    xArr, yArr = loadDataSet('ex0.txt') #加载数据集
    n = len(xArr) #数据个数
    xcord = []; ycord = [] #样本点
    for i in range(n):
        xcord.append(xArr[i][1]); ycord.append(yArr[i]) #样本点
    fig = plt.figure()
    ax = fig.add_subplot(111) #添加subplot
    ax.scatter(xcord, ycord, s = 20, c = 'blue', alpha = .5) #绘制样本点
    plt.title('DataSet') #绘制title
    plt.xlabel('X')
```

```

plt.ylabel('Y')
plt.show()

if __name__ == '__main__':
    plotDataSet()

```

打开文本编辑器并创建一个新的文件 `regression.py`，添加其中的代码。

```

from numpy import *

def loadDataSet(fileName):
    """
    Parameters:
        fileName - 文件名
    Returns:
        xArr - x数据集
        yArr - y数据集
    """
    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

# 计算回归系数w
def standRegres(dataMat, labelMat):
    """
    Parameters:
        dataMat - x数据集
        labelMat - y数据集
    Returns:
        ws - 回归系数
    """
    xMat = np.mat(dataMat); yMat = np.mat(labelMat).T
    xTx = xMat.T * xMat #根据文中推导的公式计算回归系数
    if np.linalg.det(xTx) == 0.0:
        print("矩阵为奇异矩阵,不能求逆")
        return
    ws = xTx.I * (xMat.T*yMat)
    return ws

```

函数 `standRegres()` 用来计算最佳拟合直线。该函数首先读入 `x` 和 `y` 并将它们保存到矩阵中；然后计算 `XTX`，然后判断它的行列式是否为零，如果行列式为零，那么计算逆矩阵的时

候将出现错误。NumPy 提供一个线性代数的库 `linalg`，其中包含很多有用的函数。可以直接调用 `linalg.det()` 来计算行列式。最后，如果行列式非零，计算并返回 `w`。如果没有检查行列式是否为零就试图计算矩阵的逆，将会出现错误。NumPy 的线性代数库还提供一个函数来解未知矩阵，如果使用该函数，那么代码 `ws=xTx.I * (xMat.T*yMat)` 应写成 `ws=linalg.solve(xTx, xMat.T*yMatT)`。

下面看看实际效果，使用 `loadDataSet()` 将从数据中得到两个数组，分别存放在 `x` 和 `y` 中。与分类算法中的类别标签类似，这里的 `y` 是目标值。

```
import regression
from numpy import *
xArr, yArr = regression.loadDataSet('ex0.txt')
```

首先看前两条数据：

```
>>> xArr[0:2]
[[1.0, 0.067732000000000001], [1.0, 0.42781000000000002]]
```

第一个值总是等于 1.0，即 X_0 。我们假定偏移量就是一个常数。第二个值 X_1 ，也就是我们图中的横坐标值。

现在看一下 `standRegres()` 函数的执行效果：

```
>>> ws = regression.standRegress(xArr, yArr)
>>> ws
matrix([[3.00774324],
        [1.69532264]])
```

变量 `ws` 存放的就是回归系数。在用内积来预测 `y` 的时候，第一维将乘以前面的常数 X_0 ，第二维将乘以输入变量 X_1 。因为前面假定了 $X_0=1$ ，所以最终会得到 $y=ws[0]+ws[1]*X_1$ 。这里的 `y` 实际是预测出的，为了和真实的 `y` 值区分开来，我们将它记为 `yHat`。下面使用新的 `ws` 值计算 `yHat`：

```
>>> xMat = mat(xArr)
>>> yMat = mat(yArr)
>>> yHat = xMat*ws
```

现在就可以绘出数据集散点图和最佳拟合直线图：

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.scatter(xMat[:,1].flatten().A[0], yMat.T[:,0].flatten().A[0])
```

上述命令创建了图像并绘出了原始的数据。为了绘制计算出的最佳拟合直线，需要绘出 `yHat` 的值。如果直线上的数据点次序混乱，绘图时将会出现问题，所以首先要将点按照升序排列：

```
>>> xCopy = xMat.copy()
>>> xCopy.sort(0)
>>> yHat = xCopy*ws
```

```
>>> ax.plot(xCopy[:,1], yHat)
>>> plt.show()
```

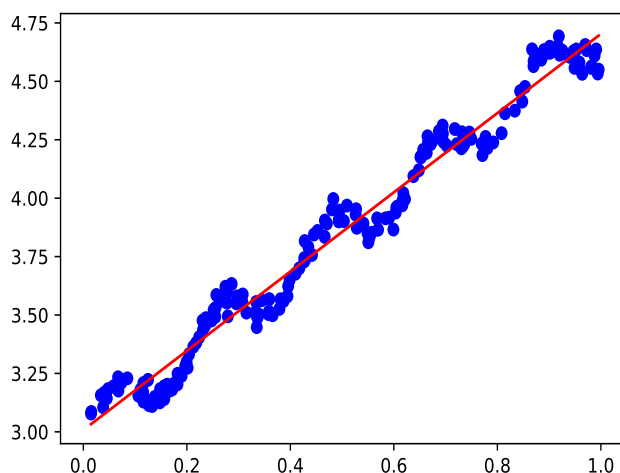


图 3.2: 样例数据图

3.1 评估模型

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。

在 Python 中，NumPy 库提供了相关系数的计算方法：可以通过命令 `corrcoef(yEstimate, yActual)` 来计算预测值和真实值的相关性。下面我们就在前面的数据集上做个实验。

与之前一样，首先计算出 y 的预测值 \hat{y} ：

```
>>> yHat = xMat*ws
>>> corrcoef(yHat.T, yMat)
array([[1.    , 0.98647356],
       [0.98647356, 1.]])
```

该矩阵包含所有两两组合的相关系数。可以看到，对角线上的数据是 1.0，因为 \hat{y} 和自己的匹配是最完美的，而 \hat{y} 和 y 的相关系数为 0.98。

最佳拟合直线方法将数据视为直线进行建模，具有十分不错的表现。但是图3.2的数据当中似乎还存在其他的潜在模式。那么如何才能利用这些模式呢？我们可以根据数据来局部调整预测，下面就会介绍这种方法。

3.2 局部加权线性回归

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；然后在这个子集上基于最小均方差

来进行普通的回归。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：

$$\hat{w} = (X^T W X)^{-1} X^T W y \quad (3.3)$$

其中 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：

$$w(i, i) = \exp\left(\frac{|x^{(i)} - x|^2}{-2k^2}\right) \quad (3.4)$$

这样就构建了一个只含对角元素的权重矩阵 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3.7中可以看到参数 k 与权重的关系。

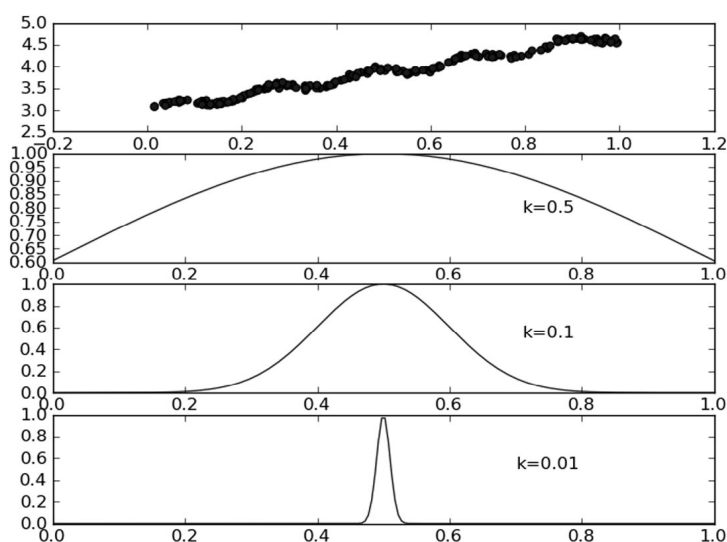


图 3.3: 每个点的权重图（假定我们正预测的点是 $x = 0.5$ ），最上面的图是原始数据集，第二个图显示了当 $k = 0.5$ 时，大部分的数据都用于训练回归模型；而最下面的图显示当 $k = 0.01$ 时，仅有很少的局部点被用于训练回归模型

下面看看模型的效果，打开文本编辑器，将代码添加到文件 `regression.py` 中。

```
# 使用局部加权线性回归计算回归系数w
def lwlr(testPoint, xArr, yArr, k=1.0):
    """
    Parameters:
        testPoint - 测试样本点
        xArr - x数据集
        yArr - y数据集
        k - 高斯核的k,自定义参数
    Returns:
        ws - 回归系数
    """
    xMat = np.mat(xArr);
    yMat = np.mat(yArr).T
```

```

m = np.shape(xMat)[0]
weights = np.mat(np.eye((m))) # 创建权重对角矩阵
for j in range(m): # 遍历数据集计算每个样本的权重
    diffMat = testPoint - xMat[j, :]
    weights[j, j] = np.exp(diffMat * diffMat.T / (-2.0 * k ** 2))
xTx = xMat.T * (weights * xMat)
if np.linalg.det(xTx) == 0.0:
    print("矩阵为奇异矩阵,不能求逆")
    return
ws = xTx.I * (xMat.T * (weights * yMat)) # 计算回归系数
return testPoint * ws

# 局部加权线性回归测试
def lwlrTest(testArr, xArr, yArr, k=1.0):
    """
    Parameters:
        testArr - 测试数据集
        xArr - x数据集
        yArr - y数据集
        k - 高斯核的k,自定义参数
    Returns:
        ws - 回归系数
    """
    m = np.shape(testArr)[0] # 计算测试数据集大小
    yHat = np.zeros(m)
    for i in range(m): # 对每个样本点进行预测
        yHat[i] = lwlr(testArr[i], xArr, yArr, k)
    return yHat

```

程序清单中代码的作用是，给定 x 空间中的任意一点，计算出对应的预测值 y_{Hat} 。函数 `lwlr()` 的开头读入数据并创建所需矩阵，之后创建对角权重矩阵 `weights`。权重矩阵是一个方阵，阶数等于样本点个数。也就是说，该矩阵为每个样本点初始化了一个权重。接着，算法将遍历数据集，计算每个样本点对应的权重值：随着样本点与待预测点距离的递增，权重将以指数级衰减。输入参数 k 控制衰减的速度。与之前的函数 `stand-Regress()` 一样，在权重矩阵计算完毕后，就可以得到对回归系数 ws 的一个估计。

另一个函数是 `lwlrTest()`，用于为数据集中每个点调用 `lwlr()`，这有助于求解 k 的大小。

下面看看实际效果。在 Python 提示符下输入如下命令：

```
>>> reload(regression)
```

如果需要重新载入数据集，则输入：

```
>>> xArr, yArr = regression.loadDataSet('ex0.txt')
```

可以对单点进行估计：

```
>>> yArr[0]
3.1765129999999999
```

```
>>> regression.lwlr(xArr[0], xArr, yArr, 1.0)
matrix([[3.12204471]])
>>> regression.lwlr(xArr[0], xArr, yArr, 0.001)
matrix([[3.20175729]])
```

为了得到数据集里所有点的估计，可以调用 `lwlrTest()` 函数：

```
>>> yHat = regression.lwlr(xArr, xArr, yArr, 0.003)
```

下面绘出这些估计值和原始值，看看 `yHat` 的拟合效果。所用的绘图函数需要将数据点按序排列，首先对 `xArr` 排序：

```
xMat = mat(xArr)
>>> srtInd = xMat[:,1].argsort(0)
>>> xSort = xMat[srtInd][:, 0, :]
```

然后用 Matplotlib 绘图：

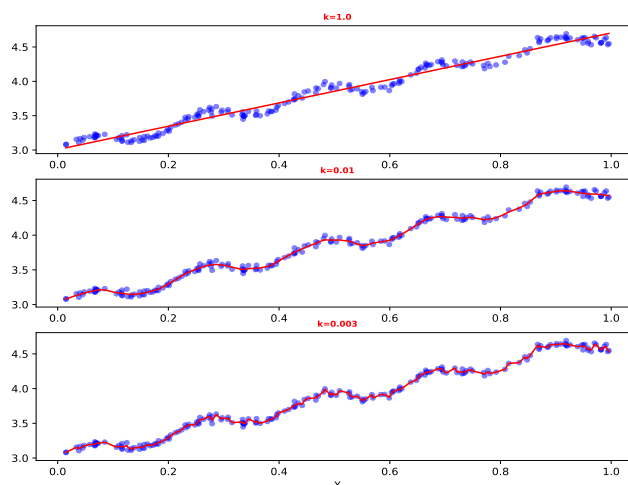


图 3.4: 使用 3 种不同平滑值绘出的局部加权线性回归结果。上图中的平滑参数 $k=1.0$ ，中图 $k=0.01$ ，下图 $k=0.003$ 。可以看到， $k=1.0$ 时的模型效果与最小二乘法差不多， $k=0.01$ 时该模型可以挖出数据的潜在规律，而 $k=0.003$ 时则考虑了太多的噪声，进而导致了过拟合现象

可以观察到如图 3.4 所示的效果。图 3.4 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。下图使用 $k=0.003$ 纳入了太多的噪声点，拟合的直线与数据点过于贴近。所以，图 3.4 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。下一节将对过拟合和欠拟合进行量化分析。

局部加权线性回归也存在一个问题，即增加了计算量，因为它对每个点做预测时都必须使用整个数据集。从图 8-5 可以看出， $k=0.01$ 时可以得到很好的估计，但是同时看一下图 3.7 中 $k=0.01$ 的情况，就会发现大多数数据点的权重都接近零。如果避免这些计算将可以减少程序运行时间，从而缓解因计算量增加带来的问题。

第 3 章 练习

1. 预测鲍鱼的年龄

数据集介绍：鲍鱼数据集可以从 UCI 中获得。此数据集数据以逗号分隔，没有列头。每个列的名字存在另外一个文件中。建立预测模型所需的数据包括性别、长度、直径、高度、整体重量、去壳后重量、脏器重量、壳的重量、环数。最后一列“环数”是十分耗时采获得的，需要锯开壳，然后在显微镜下观察得到。这是一个有监督机器学习方法通常需要的准备工作。基于一个已知答案的数据集构建预测模型，然后用这个预测模型预测不知道答案的数据。鲍鱼年龄可以从鲍鱼壳的层数推算得到。

Data Set Characteristics:	Multivariate	Number of Instances:	4177	Area:	Life
Attribute Characteristics:	Categorical, Integer, Real	Number of Attributes:	8	Date Donated	1995-12-01
Associated Tasks:	Classification	Missing Values?	No	Number of Web Hits:	1279656

图 3.5: Abalone 数据集统计图

- 从浙大云盘下载数据并编写读数据的程序；
- 建立线性模型预测鲍鱼的年龄；
- 利用局部加权线性回归预测鲍鱼的年龄；
- 可视化预测结果；
- 评估不同模型（利用前 100 个数据训练，100-200 数据用于测试）。

3.3 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。为了解决这个问题，统计学家引入了岭回归 (ridge regression) 的概念，这就是本节将介绍的第一种缩减方法。接着是 lasso 法，该方法效果很好但计算复杂。

3.3.1 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 mm 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：

$$\hat{\mathbf{w}} = (\mathbf{X}^T\mathbf{X} + \lambda\mathbf{I})^{-1}\mathbf{X}^T\mathbf{y} \quad (3.5)$$

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 \mathbf{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)。

 **笔记** 岭回归中的岭是什么？

岭回归使用了单位矩阵乘以常量 λ ，我们观察其中的单位矩阵 \mathbf{I} ，可以看到值 1 贯穿整个对角线，其余元素全是 0。形象地，在 0 构成的平面上有一条 1 组成的“岭”，这就是岭回归中的“岭”的由来。

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。

通过预测误差最小化得到 λ ：数据获取之后，首先抽一部分数据用于测试，剩余的作为训练集用于训练参数 \mathbf{w} 。训练完毕后在测试集上测试预测性能。通过选取不同的 λ 来重复上述测试过程，最终得到一个使预测误差最小的 λ 。

下面看看实际效果，打开 `regression.py` 文件并添加下列代码。

```
# 岭回归
def ridgeRegres(xMat, yMat, lam = 0.2):
    """
    Parameters:
        xMat - x数据集
        yMat - y数据集
        lam - 缩减系数
    Returns:
        ws - 回归系数
    """
    xTx = xMat.T * xMat
    denom = xTx + np.eye(np.shape(xMat)[1]) * lam
    if np.linalg.det(denom) == 0.0:
        print("矩阵为奇异矩阵,不能转置")
        return
    ws = denom.I * (xMat.T * yMat)
    return ws

# 岭回归测试
def ridgeTest(xArr, yArr):
    """
    Parameters:
        xMat - x数据集
        yMat - y数据集
    Returns:
        wMat - 回归系数矩阵
    """
    xMat = np.mat(xArr); yMat = np.mat(yArr).T
    #数据标准化
    yMean = np.mean(yMat, axis = 0)           #行与行操作，求均值
    yMat = yMat - yMean                       #数据减去均值
    xMeans = np.mean(xMat, axis = 0)           #行与行操作，求均值
    xVar = np.var(xMat, axis = 0)              #行与行操作，求方差
    xMat = (xMat - xMeans) / xVar              #数据减去均值除以方差实现标准化
    numTestPts = 30                           #30个不同的lambda测试
    wMat = np.zeros((numTestPts, np.shape(xMat)[1])) #初始回归系数矩阵
    for i in range(numTestPts):                #改变lambda计算回归系数
        ws = ridgeRegres(xMat, yMat, np.exp(i - 10)) #lambda以e的指数变化，最初是一个非常小的数，
        wMat[i, :] = ws.T                      #计算回归系数矩阵
    return wMat
```

函数 `ridgeRegres()` 用于计算回归系数，而函数 `ridgeTest()` 用于在一组 λ 上测试结果。

第一个函数 `ridgeRegres()` 实现了给定 λ 下的岭回归求解。如果没指定 λ ，则默认为 0.2。由于 λ 是 Python 保留的关键字，因此程序中使用了 `lam` 来代替。该函数首先构建矩阵 $\mathbf{X}^T\mathbf{X}$ ，然后用 `lam` 乘以单位矩阵（可调用 NumPy 库中的方法 `eye()` 来生成）。在普通回归方法可能会产生错误时，岭回归仍可以正常工作。那么是不是就不再需要检查行列式是否为零，对吗？不完全对，如果 λ 设定为 0 的时候一样可能会产生错误，所以这里仍需要做一个检查。最后，如果矩阵非奇异就计算回归系数并返回。

为了使用岭回归和缩减技术，首先需要对特征做标准化处理。回忆一下，第 2 章已经用过标准化处理技术，使每维特征具有相同的重要性（不考虑特征代表什么）。程序中的第二个函数 `ridgeTest()` 就展示了数据标准化的过程。具体的做法是所有特征都减去各自的均值并除以方差。

处理完成后就可以在 30 个不同的 λ 下调用 `ridgeRegres()` 函数。注意，这里的 λ 应以指数级变化，这样可以看出 λ 在取非常小的值时和取非常大的值时分别对结果造成的影响。最后将所有的回归系数输出到一个矩阵并返回。

下面看一下鲍鱼数据集上的运行结果。

```
>>> reload(regression)
>>> abX, abY = regression.loadDataSet('abalone.txt')
>>> ridgeWeights = regression.ridgeTest(abX, abY)
```

这样就得到了 30 个不同 `lambda` 所对应的回归系数。为了看到缩减的效果，在 Python 提示符下输入如下代码：

```
>>> import matplotlib.pyplot as plt
>>> fig = plt.figure()
>>> ax = fig.add_subplot(111)
>>> ax.plot(ridgeWeights)
>>> plt.show()
```

运行之后应该看到一个类似图 8-6 的结果图，该图绘出了回归系数与 $\log(\lambda)$ 的关系。在最左边，即最小时，可以得到所有系数的原始值（与线性回归一致）；而在右边，系数全部缩减成 0；在中间部分的某值将可以取得最好的预测效果。为了定量地找到最佳参数值，还需要进行交叉验证。另外，要判断哪些变量对结果预测最具有影响力，在图 8-6 中观察它们对应的系数大小就可以。

3.3.2 lasso 回归

不难证明，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的一样的公式：

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

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，可能会得出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。

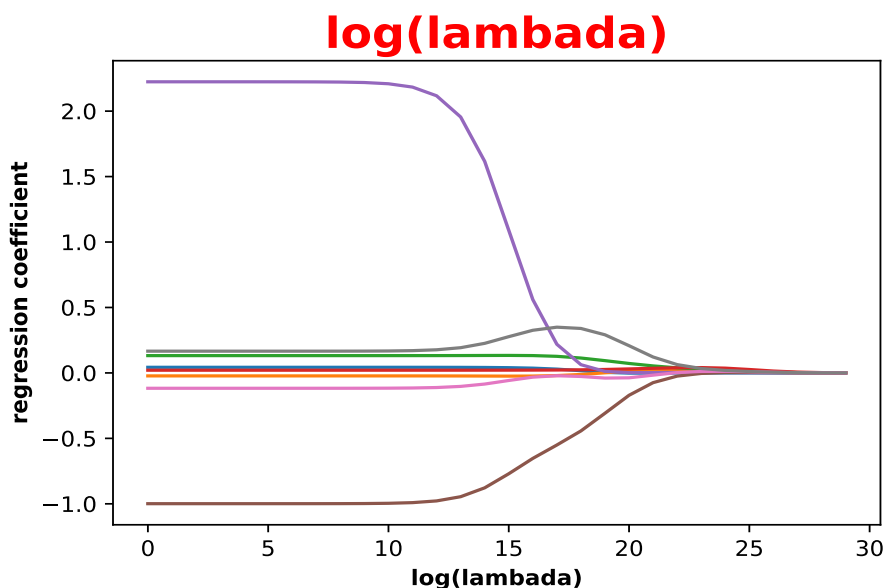


图 3.6: 岭回归的回归系数变化图。 λ 非常小时, 系数与普通回归一样。而 λ 非常大时, 所有回归系数缩减为 0。可以在中间某处找到使得预测的结果最好的 λ 值

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。这两个约束条件在公式上看起来相差无几, 但细微的变化却极大地增加了计算复杂度 (为了在这个新的约束条件下解出回归系数, 需要使用二次规划算法)。下面将介绍一个更为简单的方法来得到结果, 该方法叫做前向逐步回归。

3.3.3 前向逐步回归

前向逐步回归算法可以得到与 lasso 差不多的效果, 但更加简单。它属于一种贪心算法, 即每一步都尽可能减少误差。一开始, 所有的权重都设为 1, 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。

该算法的伪代码如下所示:

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`

下面看看实际效果，打开 regression.py 文件并加入下列程序清单中的代码。

```
# 前向逐步线性回归
def stageWise(xArr, yArr, eps=0.01, numIt=100):
    """
    Parameters:
        xArr - x输入数据
        yArr - y预测数据
        eps - 每次迭代需要调整的步长
        numIt - 迭代次数

    Returns:
        returnMat - numIt次迭代的回归系数矩阵
    """
    xMat = np.mat(xArr);
    yMat = np.mat(yArr).T # 数据集
    xMat, yMat = regularize(xMat, yMat) # 数据标准化
    m, n = np.shape(xMat)
    returnMat = np.zeros((numIt, n)) # 初始化numIt次迭代的回归系数矩阵
    ws = np.zeros((n, 1)) # 初始化回归系数矩阵
    wsTest = ws.copy()
    wsMax = ws.copy()
    for i in range(numIt): # 迭代numIt次
        # print(ws.T) #打印当前回归系数矩阵
        lowestError = float('inf'); # 正无穷
        for j in range(n): # 遍历每个特征的回归系数
            for sign in [-1, 1]:
                wsTest = ws.copy()
                wsTest[j] += eps * sign # 微调回归系数
                yTest = xMat * wsTest # 计算预测值
                rssE = rssError(yMat.A, yTest.A) # 计算平方误差
                if rssE < lowestError: # 如果误差更小，则更新当前的最佳回归系数
                    lowestError = rssE
                    wsMax = wsTest
        ws = wsMax.copy()
        returnMat[i, :] = ws.T # 记录numIt次迭代的回归系数矩阵
    return returnMat
```

函数 stageWise() 是一个逐步线性回归算法的实现，它与 lasso 做法相近但计算简单。该函数的输入包括：输入数据 xArr 和预测变量 yArr。此外还有两个参数：一个是 eps，表示每次迭代需要调整的步长；另一个是 numIt，表示迭代次数。

函数首先将输入数据转换并存入矩阵中，然后把特征按照均值为 0 方差为 1 进行标准化处理。在这之后创建了一个向量 ws 来保存 w 的值，并且为了实现贪心算法建立了 ws 的两份副本。接下来的优化过程需要迭代 numIt 次，并且在每次迭代时都打印出 w 向量，用于分析算法执行的过程和效果。贪心算法在所有特征上运行两次 for 循环，分别计算增加或减少该特征对误差的影响。这里使用的是平方误差，通过之前的函数 rssError() 得到。该误差初始值设为正无穷，经过与所有的误差比较后取最小的误差。整个过程循环迭代进行。

下面看一下实际效果，

```
>>> reload(regression)
>>> xArr, yArr = loadDataSet('abalone.txt')
>>> returnMat = stageWise(xArr, yArr, 0.01, 200)

array([[ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       ...,
       [ 0.05,  0. ,  0.09, ..., -0.64,  0. ,  0.36],
       [ 0.04,  0. ,  0.09, ..., -0.64,  0. ,  0.36],
       [ 0.05,  0. ,  0.09, ..., -0.64,  0. ,  0.36]])
```

上述结果中值得注意的是 w_1 和 w_6 都是 0，这表明它们不对目标值造成任何影响，也就是说这些特征很可能是不需要的。另外，在参数 ϵ 设置为 0.01 的情况下，一段时间后系数就已经饱和并在特定值之间来回震荡，这是因为步长太大的缘故。这里会看到，第一个权重在 0.04 和 0.05 之间来回震荡。

下面试着用更小的步长和更多的步数：

```
returnMat = stageWise(xArr, yArr, 0.001, 5000)

array([[ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       [ 0. ,  0. ,  0. , ...,  0. ,  0. ,  0. ],
       ...,
       [ 0.043, -0.011, 0.12 , ..., -0.963, -0.105, 0.187],
       [ 0.044, -0.011, 0.12 , ..., -0.963, -0.105, 0.187],
       [ 0.043, -0.011, 0.12 , ..., -0.963, -0.105, 0.187]])
```

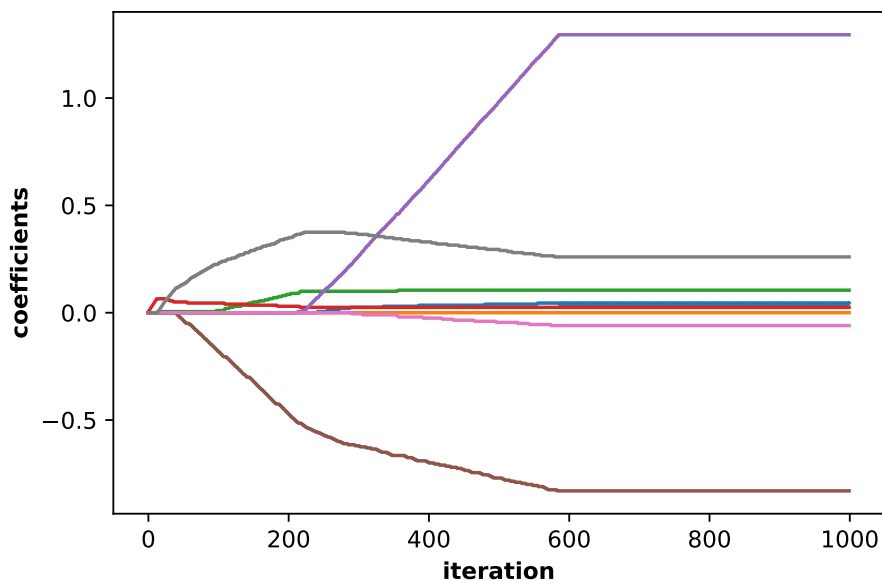


图 3.7: 鲍鱼数据集上执行逐步线性回归法得到的系数与迭代次数间的关系。逐步线性回归得到了与 lasso 相似的结果，但计算起来更加简便

逐步线性回归算法主要的优点在于它可以帮助人们理解现有的模型并做出改进。当构建

了一个模型后，可以运行该算法找出重要的特征，这样就有可能及时停止对那些不重要特征的收集。最后，如果用于测试，该算法每 100 次迭代后就可以构建出一个模型，可以使用类似于 10 折交叉验证的方法比较这些模型，最终选择使误差最小的模型。

第3章 练习

1. 预测乐高玩具套装的价格

- (a). 收集数据：用 Google Shopping 的 API 收集数据。
- (b). 准备数据：从返回的 JSON 数据中抽取价格。
- (c). 分析数据：可视化并观察数据。：构建不同的模型，采用逐步线性回归和直接的线性回归模型。
- (d). 测试算法：使用交叉验证来测试不同的模型，分析哪个效果最好。
- (e). 使用算法：这次练习的目标就是生成数据模型。

在这个例子中，我们将从不同的数据集上获取价格，然后对这些数据建立回归模型。需要做的第一件事就是如何获取数据。

收集数据：使用 Google 购物的 API

Google 已经为我们提供了一套购物的 API 来抓取价格。在使用 API 之前，需要注册一个 Google 账号，然后访问 Google API 的控制台来确保购物 API 可用。完成之后就可以发送 HTTP 请求，API 将以 JSON 格式返回所需的产品信息。Python 提供了 JSON 解析模块，我们可以从返回的 JSON 格式里整理出所需数据。详细的 API 介绍可以参见[网址](#)：从[浙大云盘](#)上下载数据

打开 regression.py 文件并加入如下代码。

```
from bs4 import BeautifulSoup

# 从页面读取数据，生成retX和retY列表
def scrapePage(retX, retY, inFile, yr, numPce, origPrc):
    """
    Parameters:
        retX - 数据X
        retY - 数据Y
        inFile - HTML文件
        yr - 年份
        numPce - 乐高部件数目
        origPrc - 原价
    Returns:
        无
    """
    # 打开并读取HTML文件
    with open(inFile, encoding='utf-8') as f:
        html = f.read()
    soup = BeautifulSoup(html)
    i = 1
    # 根据HTML页面结构进行解析
    currentRow = soup.find_all('table', r="%d" % i)
```

```

while (len(currentRow) != 0):
    currentRow = soup.find_all('table', r="%d" % i)
    title = currentRow[0].find_all('a')[1].text
    lwrTitle = title.lower()
    # 查找是否有全新标签
    if (lwrTitle.find('new') > -1) or (lwrTitle.find('nisb') > -1):
        newFlag = 1.0
    else:
        newFlag = 0.0
    # 查找是否已经标志出售, 我们只收集已出售的数据
    soldUnicode = currentRow[0].find_all('td')[3].find_all('span')
    if len(soldUnicode) == 0:
        print("商品 # %d 没有出售" % i)
    else:
        # 解析页面获取当前价格
        soldPrice = currentRow[0].find_all('td')[4]
        priceStr = soldPrice.text
        priceStr = priceStr.replace('$', '')
        priceStr = priceStr.replace(',', '', '')
        if len(soldPrice) > 1:
            priceStr = priceStr.replace('Free shipping', '')
        sellingPrice = float(priceStr)
        # 去掉不完整的套装价格
        if sellingPrice > origPrc * 0.5:
            print("%d\t%d\t%d\t%f\t%f" % (yr, numPce, newFlag, origPrc, sellingPrice))
            retX.append([yr, numPce, newFlag, origPrc])
            retY.append(sellingPrice)
    i += 1
    currentRow = soup.find_all('table', r="%d" % i)

# 依次读取六种乐高套装的数据, 并生成数据矩阵
def setDataCollect(retX, retY):
    # 2006年的乐高8288, 部件数目800, 原价49.99
    scrapePage(retX, retY, './setHtml/lego8288.html', 2006, 800, 49.99)
    # 2002年的乐高10030, 部件数目3096, 原价269.99
    scrapePage(retX, retY, './setHtml/lego10030.html', 2002, 3096, 269.99)
    # 2007年的乐高10179, 部件数目5195, 原价499.99
    scrapePage(retX, retY, './setHtml/lego10179.html', 2007, 5195, 499.99)
    # 2007年的乐高10181, 部件数目3428, 原价199.99
    scrapePage(retX, retY, './setHtml/lego10181.html', 2007, 3428, 199.99)
    # 2008年的乐高10189, 部件数目5922, 原价299.99
    scrapePage(retX, retY, './setHtml/lego10189.html', 2008, 5922, 299.99)
    # 2009年的乐高10196, 部件数目3263, 原价249.99
    scrapePage(retX, retY, './setHtml/lego10196.html', 2009, 3263, 249.99)

```

训练算法：建立模型

利用岭回归和前向逐步回归预测乐高的价格, 并对回归系数进行分析。哪些是重要的特征, 哪些是不重要的?

2. 线性回归预测糖尿病

(1) 糖尿病数据集

Sklearn 机器学习包提供了**糖尿病数据集** (Diabetes Dataset)，该数据集主要包括 442 行数据，10 个特征值，分别是：年龄 (Age)、性别 (Sex)、体质指数 (Body mass index)、平均血压 (Average Blood Pressure)、S1 S6 一年后疾病级数指标。预测指标为 Target，它表示一年后患疾病的定量指标。

```
from sklearn import datasets
diabetes = datasets.load_diabetes()

print(diabetes.data)
print(diabetes.target)
```

调用 load_diabetes() 函数载入糖尿病数据集，然后输出其数据 data 和类标 target。输出总行数 442 行，特征数共 10 个，类型为 (442L, 10L)。

(2) 代码实现现在我们将糖尿病数据集划分为训练集和测试集，整个数据集共 442 行，我们取前 422 行数据用来线性回归模型训练，后 20 行数据用来预测。其中取预测数据的代码为 diabetes_x_temp[-20:]，表示从后 20 行开始取值，直到数组结束，共取值 20 个数。整个数据集共 10 个特征值，为了方便可视化画图我们只获取其中一个特征进行实验，这也可以绘制图形，而真实分析中，通常经过降维处理再绘制图形。这里获取第 3 个特征，对应代码为：diabetes_x_temp = diabetes.data[:, np.newaxis, 2]。

(3) 代码优化

对代码进行优化。可视化绘图增加了散点到线性方程的距离线，增加了保存图片设置像素代码等。