回归分析

更新日期：2019年9月14日

一、概述

在正式进入到回归分析的相关算法讨论之前，我们需要对有监督学习算法中的回归问题进行进一步的分析和理解。虽然回归问题和分类问题同属于有监督学习范畴，但实际上，回归问题要远比分类问题更加复杂。

首先是关于输出结果的对比，分类模型最终输出结果为离散变量，而离散变量本身包含信息量较少，其本身并不具备代数运算性质，因此其评价指标体系也较为简单，最常用的就是混淆矩阵以及ROC曲线。

而回归问题最终输出的是连续变量，其本身不仅能够代数运算，且还具有更 "精致" 的方法，希望对事物运行的更底层原理进行挖掘。即回归问题的模型更加全面、完善的描绘了事物客观规律，从而能够得到更加细粒度的结论。

因此，回归问题的模型往往更加复杂，建模所需要数据所提供的信息量也越多，进而在建模过程中可能遇到的问题也越多。

二、好的起点：线性回归

1. 线性回归与机器学习

线性回归是解决回归类问题最常使用的算法模型，其算法思想和基本原理都是由多元统计分析发展而来，但在数据挖掘和机器学习领域中，也是不可多得的行之有效的算法模型。一方面，线性回归蕴藏的机器学习思想非常值得借鉴和学习，并且随着时间发展，在线性回归的基础上还诞生了许多功能强大的非线性模型。因此，我们在进行机器学习算法学习过程中，仍然需要对线性回归这个统计分析算法进行系统深入的学习。但这里需要说明的是，线性回归的建模思想有很多理解的角度，此处我们并不需要从统计学的角度来理解、掌握和应用线性回归算法，很多时候，利用机器学习的思维来理解线性回归，会是一种更好的理解方法，这也将是我们这部分内容讲解线性回归的切入角度。

2. 线性回归的机器学习表示方法

2.1 核心逻辑

任何机器学习算法首先都有一个最底层的核心逻辑，当我们在利用机器学习思维理解线性回归的时候，首先也是要探究其底层逻辑。值得庆幸的是，虽然线性回归源于统计分析，但其算法底层逻辑和机器学习算法高度契合。

在给定 n 个属性描绘的客观事物 $x=(x_1,x_2,\dots,x_p)$ 中，每个 x_i 都用于描绘某一次观测时事物在某个维度表现出来的数值属性值。当我们在建立机器学习模型捕捉事物运行的客观规律时，本质上是希望能够综合这些维度的属性值来描绘事物最终运行结果，而最简单的综合这些属性的方法就是对其进行加权求和汇总，这即是线性回归的方程式表达形式：

$$\hat{y}_i = w_0 + w_1 x_{i1} + w_2 x_{i2} + \ldots + w_p x_{ip}$$

w被统称为模型的参数，其中w_0被称为截距(intercept)，w_1~w_p被称为回归系数(regression coefficient)，有时也是使用θ或者β来表示。这个表达式，其实就和我们小学时就无比熟悉的$y = ax + b$是同样的性质。其中y是我们的目标变量，也就是标签。x_{i1}~x_{ip}是样本i上的特征不同特征。如果考虑我们有n个样本，则回归结果可以被写作：

$$\hat{\boldsymbol{y}} = w_0 + w_1 \boldsymbol{x_1} + w_2 \boldsymbol{x_2} + \ldots + w_p \boldsymbol{x_p}$$

其中$\boldsymbol{y}$是包含了n个全部的样本的回归结果的列向量（结构为(n,1)，由于只有一列，以列的形式表示，所以叫做列向量）。注意，我们通常使用粗体的小写字母来表示列向量，粗体的大写字母表示矩阵或者行列式。我们可以使用矩阵来表示这个方程，其中$\boldsymbol{w}$可以被看做是一个结构为(p+1,1)的列矩阵，$\boldsymbol{X}$是一个结构为(n,p+1)的特征矩阵，则有：

$$\begin{bmatrix} \hat{y}_1 \\ \hat{y}_2 \\ \hat{y}_3 \\ \ldots \\ \hat{y}_m \end{bmatrix} = \begin{bmatrix} 1 & x_{11} & x_{12} & x_{13} & \ldots & x_{1p} \\ 1 & x_{21} & x_{22} & x_{23} & \ldots & x_{2p} \\ 1 & x_{31} & x_{32} & x_{33} & \ldots & x_{3p} \\ & & \ldots & & & \\ 1 & x_{n1} & x_{n2} & x_{n3} & \ldots & x_{np} \end{bmatrix} * \begin{bmatrix} w_0 \\ w_1 \\ w_2 \\ \ldots \\ w_p \end{bmatrix}$$

$$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$$

线性回归的任务，就是构造一个预测函数来映射输入的特征矩阵$\boldsymbol{X}$和标签值$\boldsymbol{y}$的线性关系，这个预测函数在不同的教材上写法不同，可能写作$f(x)$，$y_w(x)$，或者$h(x)$等等形式，但无论如何，**这个预测函数的本质就是我们需要构建的模型**。

2.2 优化目标

对于线性回归而言，预测函数$\hat{\boldsymbol{y}} = \boldsymbol{X}\boldsymbol{w}$就是我们的模型，在机器学习我们也称作 "**决策函数**"。其中只有$\boldsymbol{w}$是未知的，所以线性回归原理的核心就是找出模型的参数向量$\boldsymbol{w}$。但我们怎样才能够求解出参数向量呢？我们需要依赖一个重要概念：**损失函数**。

在之前的算法学习中，我们提到过两种模型表现：在训练集上的表现，和在测试集上的表现。我们建模，是追求模型在测试集上的表现最优，因此模型的评估指标往往是用来衡量模型在测试集上的表现的。然而，线性回归有着基于训练数据求解参数$\boldsymbol{w}$的需求，并且希望训练出来的模型能够尽可能地拟合训练数据，即模型在训练集上的预测准确率越靠近100%越好。

因此，我们使用”**损失函数**“这个评估指标，来**衡量系数为$\boldsymbol{w}$的模型拟合训练集时产生的信息损失的大小，并以此衡量参数$\boldsymbol{w}$的优劣**。如果用一组参数建模后，模型在训练集上表现良好，那我们就说模型拟合过程中的损失很小，损失函数的值很小，这一组参数就优秀；相反，如果模型在训练集上表现糟糕，损失函数就会很大，模型就训练不足，效果较差，这一组参数也就比较差。即是说，我们在求解参数$\boldsymbol{w}$时，追求损失函数最小，让模型在训练数据上的拟合效果最优，即预测准确率尽量靠近100%。

关键概念：损失函数

衡量参数$\boldsymbol{w}$的优劣的评估指标，用来求解最优参数的工具
损失函数小，模型在训练集上表现优异，拟合充分，参数优秀
损失函数大，模型在训练集上表现差劲，拟合不足，参数糟糕
我们追求，能够让损失函数最小化的参数组合

注意：对于非参数模型没有损失函数，比如KNN、决策树

对于有监督学习算法而言，建模都是依据有标签的数据集，回归类问题则是对客观事物的一个定量判别。这里以 y_i 作为第 i 行数据的标签，且 y_i 为连续变量， x_i 为第 i 行特征值所组成的向量，则线性回归建模优化方向就是希望模型判别的 $\hat{y}_i$ 尽可能地接近实际的 y_i 。而对于连续型变量而言，邻近度度量方法可采用SSE 来进行计算，SSE 称作「残差平方和」，也称作「误差平方和」或者「离差平方和」。因此我们的优化目标可用下述方程来进行表示：

$$\min_{w}\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$$
$$=\min_{w}\sum_{i=1}^{n}(y_i-\boldsymbol{X_i}\boldsymbol{w})^2$$

来看一个简单的小栗子🌰。假设现在$\boldsymbol{w}$为[1,2]这样一个向量，求解出的模型为$y=x_1+2x_2$。

样本	特征1	特征2	真实标签
0	1	0.5	3
1	-1	0.5	2

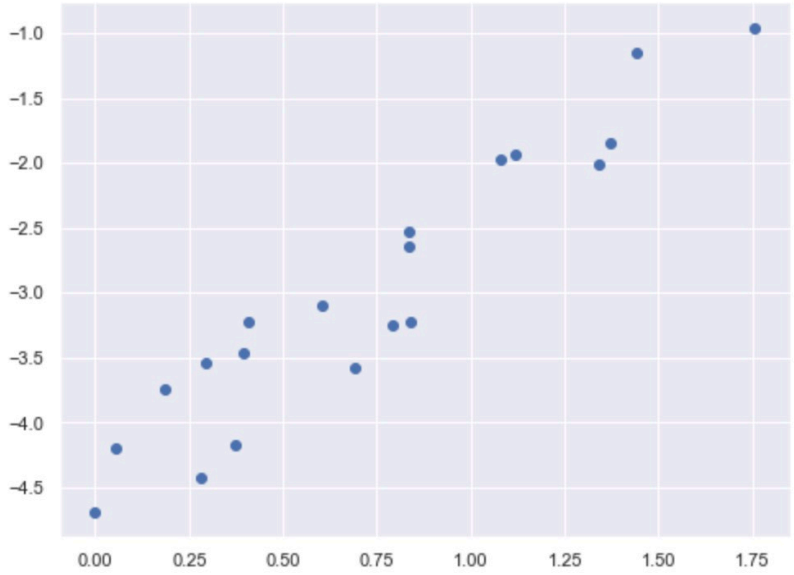
则我们的损失函数的值就是：

$$\begin{array}{ccc}(3-(1*1+2*0.5))^2 & + & (2-(1*-1+2*0.5))^2\\ (y_1-\hat{y}_1)^2 & + & (y_2-\hat{y}_2)^2\end{array}$$

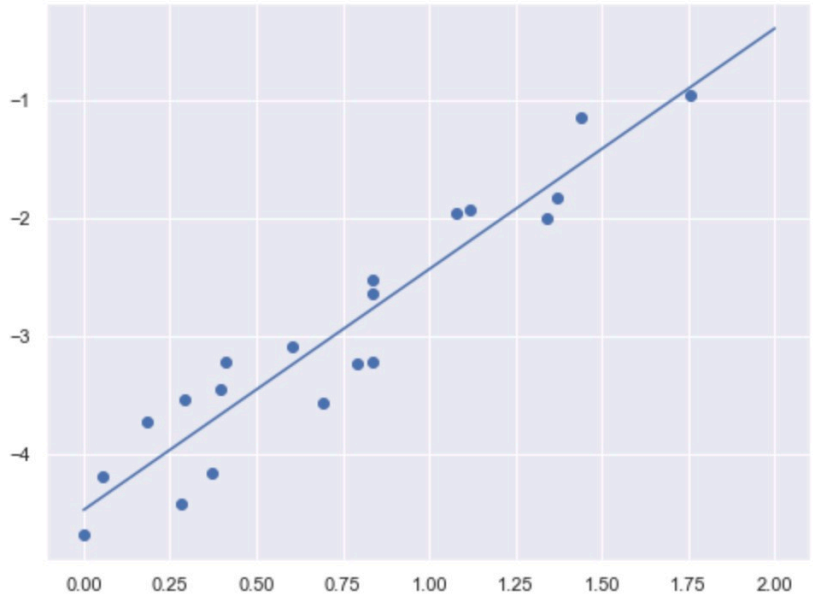
2.3 最小二乘法

现在问题转换成了求解让 SSE 最小化的参数向量$\boldsymbol{w}$，**这种通过最小化真实值和预测值之间的 SSE 来求解参数的方法叫做最小二乘法。**

2.3.1 回顾一元线性回归的求解过程



首先我们对上图的数据进行拟合，可得到：



假设该拟合直线为 $\hat{y} = w_0 + w_1 x$。现在我们的目标是使得该拟合直线的总的残差和达到最小，也就是最小化SSE 。

对于真实值 y 来说，我们可以得到：

$$y = w_0 + w_1 x + \varepsilon$$

这里的ε 即为「残差」，对其进行变形，则残差平方和SSE 就为：

$$\sum \varepsilon^2 = \sum (y - w_0 - w_1 x)^2$$

要求得残差平方和最小值，我们通过微积分求偏导算其极值来解决。这里我们计算残差最小对应的参数 w_0, w_1 ：

$$\frac{\partial}{\partial w_0} = 2\sum(y_i - w_0 - w_1 x)(-1) = 0$$
$$\sum y - \sum w_0 - w_1 \sum x = 0$$
$$\bar{y} - n w_0 - n w_1 \bar{x} = 0$$
$$w_0 = \bar{y} - w_1 \bar{x}$$

$$\frac{\partial}{\partial w_1} = 2\sum(y - w_0 - w_1 x)(-x) = 0 \qquad \text{一阶导数}$$

$$\because \quad w_0 = \bar{y} - w_1 \bar{x}$$

$$\therefore \quad \sum(y - \bar{y} + w_1 \bar{x} - w_1 x)(x) = 0$$

$$\sum(yx - \bar{y}x + w_1 \bar{x}x - w_1 x^2) = 0$$

$$\sum(yx - \bar{y}x) + \sum(w_1 \bar{x}x - w_1 x^2) = 0$$

$$\sum(yx - \bar{y}x) = \sum(w_1 x^2 - w_1 \bar{x}x)$$

$$\sum(y - \bar{y})x = w_1 \sum(x - \bar{x})x$$

则 $\quad w_1 = \dfrac{\sum(y - \bar{y})x}{\sum(x - \bar{x})x}$

根据协方差和方差的推导公式，我们将分子分母同时除以$n-1$，则可以得到：

$$\hat{w_1} = \frac{\sum(y - \bar{y})(x - \bar{x})}{\sum(x - \bar{x})^2}$$

同理，可知：

$$\hat{w_0} = \bar{y} - \hat{w_1}\bar{x}$$

此时，使得 SSE 最小的量 $\hat{w}_0, \hat{w}_1$ 称为 总体参数 w_0, w_1 的最小二乘估计值，预测方程 $\hat{y} = \hat{w}_0 + \hat{w}_1 x$ 称为最小二乘直线。

2.3.2 多元线性回归求解参数

更一般的情形，还记得我们刚才举的小栗子么？如果我们有两列特征属性，则损失函数为：

$$(y_1 - \hat{y}_1)^2 \quad + \quad (y_2 - \hat{y}_2)^2$$

这样的形式如果用矩阵来表达，可以写成：

$$
\begin{aligned}
&\left[(y_1-\hat{y}_1)\ (y_2-\hat{y}_2)\right] * \begin{bmatrix}(y_1-\hat{y}_1)\\(y_2-\hat{y}_2)\end{bmatrix}\\
&=(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})
\end{aligned}
$$

矩阵相乘是对应未知元素相乘相加，就会得到和上面的式子一模一样的结果。

我们同样对 $\boldsymbol{w}$ 求偏导：

$$
\frac{\partial SSE}{\partial \boldsymbol{w}}=\frac{\partial(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})}{\partial \boldsymbol{w}}
$$

$$
\because (A-B)^T=A^T-B^T \text{并且} (AB)^T=B^T*A^T
$$

$$
\begin{aligned}
\therefore &= \frac{\partial(\boldsymbol{y}^T-\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}})(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})}{\partial \boldsymbol{w}}\\
&= \frac{\partial(\boldsymbol{y}^T\boldsymbol{y}-\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}-\boldsymbol{y}^T\boldsymbol{X}\boldsymbol{w}+\boldsymbol{w}^{\boldsymbol{T}}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w})}{\partial \boldsymbol{w}}
\end{aligned}
$$

$$
\because \text{矩阵求导中，}a\text{为常数，有如下规则：}
$$

$$
\frac{\partial a}{\partial A}=0,\quad \frac{\partial A^TB^TC}{\partial A}=B^TC,\quad \frac{\partial C^TBA}{\partial A}=B^TC,\quad \frac{\partial A^TBA}{\partial A}=(B+B^T)A
$$

可得到：

$$
\begin{aligned}
&=0-\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}-\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}+2\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}\\
&=\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}-\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}
\end{aligned}
$$

我们让求导后的一阶导数为0：

$$
\begin{aligned}
\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}-\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}&=0\\
\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}&=\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}\\
\text{左乘一个}(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}&\text{则有：}\\
\boldsymbol{w}&=(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}
\end{aligned}
$$

到了这里，我们希望能够将$\boldsymbol{w}$留在等式的左边，其他与特征矩阵有关的部分都放到等式的右边，如此就可以求出$\boldsymbol{w}$的最优解了。这个功能非常容易实现，只需要我们左乘$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$的逆矩阵就可以。**在这里，逆矩阵存在的充分必要条件是特征矩阵不存在多重共线性**。我们将会在后面详细讲解多重共线性这个主题。

假设矩阵的逆是存在的，此时我们的$\boldsymbol{w}$就是我们参数的最优解。求解出这个参数向量，我们就解出了我们的$\boldsymbol{X}\boldsymbol{w}$，也就能够计算出我们的预测值$\hat{\boldsymbol{y}}$了。

三、多元线性回归Python实现

1. 利用矩阵乘法编写回归算法

多元线性回归的执行函数编写并不复杂，主要涉及大量的矩阵运算，需要借助NumPy中的矩阵数据格式来完成。首先执行标准化导入：

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

然后回顾可能用到的矩阵运算相关知识：

- 矩阵创建

  NumPy中创建矩阵需要使用mat函数，该函数需要输入二维数组。

```
a = np.array([[1,2],[3,4]])
m = np.mat(a)    # NumPy中创建矩阵需要使用mat函数，该函数需要输入二维数组
m
```

在NumPy中，矩阵和数组有非常多类似的地方，例如矩阵的索引、切片等常用方法和数组均保持一致。

- 矩阵运算

  当然，定义矩阵对象主要是为了能够执行矩阵运算，NumPy中有个linalg库专门给矩阵对象提供计算方法，下面简单回归常用矩阵运算。

  ```
  # 矩阵转置
  m.T
  # 矩阵乘法
  m * m
  a * a
  # 矩阵行列式
  np.linalg.det(m)
  # 求逆矩阵
  m.I
  ```

  然后编写线性回归函数，同理，我们假设输入数据集为DataFrame，且最后一列为标签值。

  ```
  def standRegres(dataSet):
      xMat = np.mat(dataSet.iloc[:,:-1].values)    # 提取特征
      yMat = np.mat(dataSet.iloc[:,-1].values).T    # 提取标签
      xTx = xMat.T*xMat
      if np.linalg.det(xTx) ==0:
          print('This matrix is singular,cannot do inverse')    # 行列式为0,
  则该矩阵为奇异矩阵，无法求解逆矩阵
          return
      ws = xTx.I * (xMat.T*yMat)
      return ws
  ```

这里需要提前判断xTx是否是满秩矩阵。若不满秩，则无法对其进行求逆矩阵的操作。定义完函数后，即可测试运行效果，此处我们建立线性随机数进行多元线性回归方程拟合。这里需要注意的是，当使用矩阵分解来求解多元线性回归时，必须添加一列全为1的列，用于表征线性方程截距w_0。

```
rng = np.random.RandomState(1)      # 设置随机数种子
x = 5*rng.rand(100)                             # 100个[0,5)的随机树
y = 2*x-5+rng.randn(100)                  # 真实规律的标签取值

X = pd.DataFrame(x)
Y = pd.DataFrame(y)
ex = pd.DataFrame(np.ones([100,1]))       # 添加一列全为1的列，表示截距

data = pd.concat([ex,X,Y],axis=1)
```

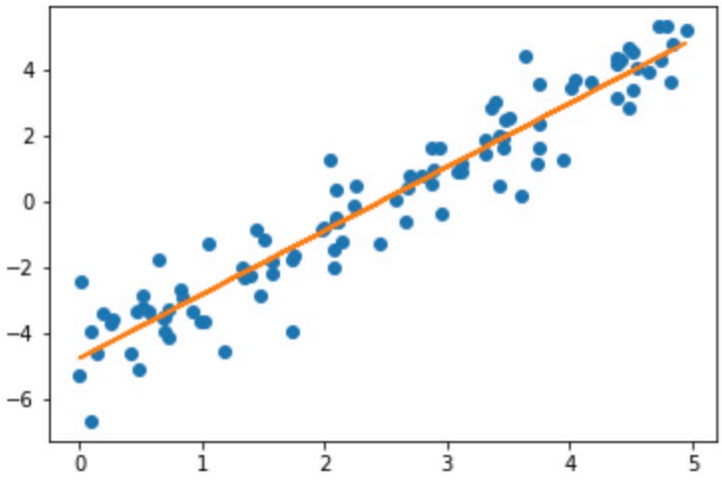
数据满足基本建模要求，然后执行函数运算：

```
ws = standRegres(data)
ws
```

返回结果即为各列特征权重，其中数据集第一列值均为1，因此返回结果的第一个分量表示截距。

然后可用可视化图形展示模型拟合效果：

```
yhat = data.iloc[:,:-1].values*ws                 # 预测标签值
plt.plot(data.iloc[:,1],data.iloc[:,2],'o')       # 原始数据点
plt.plot(data.iloc[:,1],yhat)                     # 拟合直线
```



2. 回归算法评估指标

接下来讨论回归算法的评价指标体系，首先是残差平方和 SSE ，其计算公式如下所示：

$$SSE = \sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

其中 m 为数据集记录数， y_i 为实际标签值，$\hat{y}_i$ 为预测值，则可利用下属表达式进行计算。

```
y = data.iloc[:,-1].values
yhat = yhat.flatten()
SSE = np.power(yhat-y,2).sum()
SSE
```

当然，我们也可将其编写为一个完整的函数，为了提高复用性，设置输入参数为数据集和回归方法：

```
def sseCal(dataSet,regres):
    n = dataSet.shape[0]
    y = dataSet.iloc[:,-1].values
    ws = regres(dataSet)
    yhat = dataSet.iloc[:,:-1].values*ws
    yhat = yhat.flatten()
    SSE = np.power(yhat-y,2).sum()
    return SSE
```

测试运行：

```
sseCal(data,standRegres)
```

同时，在回归算法中，为了消除数据集规模对残差平方和的影响，往往我们还会计算平均残差MSE：

$$MSE = \frac{1}{m}\sum_{i=1}^{m}(y_i - \hat{y}_i)^2$$

其中 m 为数据集样例个数，以及$RMSE$ 误差的均方根，为 MSE 开平方后所得结果。当然除此之外最常用的还是 $R-square$ 判定系数，判定系数的计算需要使用之前介绍的组间误差平方和和离差平方和的概念。

在回归分析中，SSR 表示聚类中类似的组间平方和概念，译为 Sum of squares of the regression，由预测数据与标签均值之间差值的平方和构成。

$$SSR = \sum_{i=1}^{m}(\bar{y_i} - \hat{y}_i)^2$$

而 SST （Total sum of squares）则是实际值和均值之间的差值的平方和：

$$SST=\sum_{i=1}^{m}(\bar{y_i}-y_i)^2$$

对比之前介绍的聚类分析，此处可对比理解为以点聚点。同样，和轮廓系数类似，最终的判定系数指标也同时结合了 "组内误差" 和 "组间误差" 两个指标。

$$R^2=\frac{SSR}{SST}=\frac{SST-SSE}{SST}=1-\frac{SSE}{SST}$$

判定系数 R^2 测度了回归直线对观测数据的拟合程度。

- 若所有观测点都落在直线上，残差平方和为 $SSE=0$ ，则 $R^2=1$ ，拟合是完全的；
- 如果 y 的变化与 x 无关，x 完全无助于解释 y 的变差， $\hat{y}=\bar{y}$ ，则 $R^2=0$。

可见 R^2 的取值范围是$[0,1]$ 。

- R^2 越接近1，表明回归平方和占总平方和的比例越大，回归直线与各观测点越接近，用 x 的变化来解释 y 值变差（y 取值的波动称为变差）的部分就越多，回归直线的拟合程度就越好；
- 反之，R^2 越接近0，回归直线的拟合程度就越差。

接下来，尝试计算上述拟合结果的判定系数。

```
sse = sseCal(data,standRegres)
y = data.iloc[:,-1].values
sst = np.power(y-y.mean(),2).sum()
1-sse/sst
```

结果为0.91。能够看出最终拟合效果非常好。当然，我们也可编写函数封装计算判断系数的相关操作，同样留一个调整回归函数的接口。

```
def rSquare(dataSet,regres):
    sse = sseCal(dataSet,regres)
    y = dataSet.iloc[:,-1].values
    sst = np.power(y-y.mean(),2).sum()
    return 1-sse/sst
```

然后进行测试

```
rSquare(data,standRegres)
```

四、线性回归的Scikit-learn实现

接下来尝试利用scikit-learn算法库实现线性回归算法，并计算相应评价指标。回顾前文介绍的相关知识进行下述计算。

```
from sklearn.linear_model import LinearRegression

reg = LinearRegression()
reg.fit(data.iloc[:,:-1].values,data.iloc[:,-1].values)


reg.coef_   # 查看方程系数

reg.intercept_   # 查看截距
```

对比手动计算的ws，其结果高度一致。

然后计算模型 MSE 和判断系数：

```
from sklearn.metrics import mean_squared_error,r2_score

yhat = reg.predict(data.iloc[:,:-1])
mean_squared_error(y,yhat)
r2_score(y,yhat)
```

和上述手动编写函数结果一致。

五、多重共线性

虽然在线性回归的求解过程中，通过借助最小二乘法能够迅速找到全域最优解，但最小二乘法本身使用条件较为苛刻，必须要求当 $\boldsymbol{X}^T\boldsymbol{X}$ 为满秩矩阵时才可进行逆矩阵或广义逆矩阵的求解。在实际应用中经常会遇见矩阵不存在逆矩阵或广义逆矩阵的情况，并且当 X 的各列存在线性相关关系（即多重共线性）的时候，最小二乘法的求解结果不唯一。

这里需要注意的是，在进行数据采集过程中，数据集各列是对同一个客观事物进行客观描述，很难避免多重共线性的存在，因此存在共线性是很多数据集的一般情况。当然更为极端的情况则是数据集的列比行多，此时最小二乘法无法对其进行求解。因此，寻找线性回归算法的优化方案势在必行。

首先我们来了解下多重共线性。在第二节中我们曾推导了多元线性回归使用最小二乘法的求解原理，我们对多元线性回归的损失函数求导，并得出求解系数w的式子和过程：

$$\boldsymbol{X^T X w - X^T y = 0}$$

$$\boldsymbol{X^T X w = X^T y}$$

左乘一个$(\boldsymbol{X^T X})^{-1}$则有：

$$\boldsymbol{w = (X^T X)^{-1} X^T y}$$

在最后一步中我们需要左乘$\boldsymbol{X^T X}$的逆矩阵，而**逆矩阵存在的充分必要条件是特征矩阵不存在多重共线性**。多重共线性这个词对于许多人来说都不陌生，这一节我们会来深入讲解，什么是多重共线性，对我们参数求解又会有什么影响。

- **逆矩阵存在的充分必要条件**

首先，我们需要先理解逆矩阵存在与否的意义和影响。一个矩阵什么情况下才可以有逆矩阵呢?

根据逆矩阵的定理，若 $|A| \neq 0$ ，则矩阵 A 可逆，且

$$\boldsymbol{A^{-1} = \frac{1}{|A|} A^*}$$

其中 $\boldsymbol{A^*}$ 是为矩阵 A 的伴随矩阵，任何矩阵都可以有伴随矩阵，因此这一部分不影响逆矩阵的存在性。而分母上的行列式$|\boldsymbol{A}|$就不同了，位于分母的变量不能为0，一旦为0则无法计算出逆矩阵。因此**逆矩阵存在的充分必要条件是：矩阵的行列式不能为0**，对于线性回归而言，即是说$|\boldsymbol{X^T X}|$不能为0。这是使用最小二乘法来求解线性回归的核心条件之一。

- **行列式不为0的充分必要条件**

那行列式要不为0，需要满足什么条件呢？在这里，我们来复习一下线性代数中的基本知识。假设我们的特征矩阵$\boldsymbol{X}$结构为(m,n)，则$\boldsymbol{X^T X}$就是结构为(n,m)的矩阵乘以结构为(m,n)的矩阵，从而得到结果为(n,n)的方阵。

$$\boldsymbol{X^T X} = (n,m) * (m,n) = (n,n)$$

因此以下所有的例子都将以方阵进行举例，方便大家理解。首先区别一下矩阵和行列式：

$$矩阵\ A = \begin{bmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{bmatrix}$$

$$矩阵\ A的行列式 = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} = |A|$$

重要定义：矩阵和行列式
矩阵是一组数按一定方式排列成的数表，一般记作$\boldsymbol{A}$ 行列式是这一组数按某种运算法则最后计算出来的一个数，通常记作$\|A\|$或者$detA$

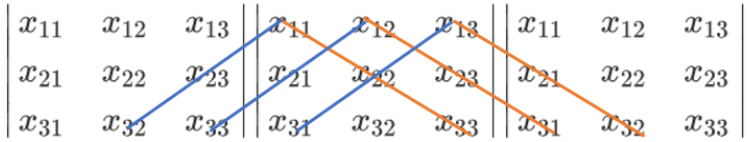
任何矩阵都可以有行列式。以一个3*3的行列式为例，我们来看看行列式是如何计算的：

$$|A| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$

$$= x_{11}x_{22}x_{33} + x_{12}x_{23}x_{31} + x_{13}x_{21}x_{32} - x_{11}x_{23}x_{32} - x_{12}x_{21}x_{33} - x_{13}x_{22}x_{31}$$

这个式子乍一看非常混乱，其实并非如此，我们把行列式$|A|$按照下面的方式排列一下，就很容易看出这个式子实际上是怎么回事了：

$$\begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix}$$



三个特征的特征矩阵的行列式就有六个交互项，在现实中我们的特征矩阵不可能是如此低维度的数据，因此使用这样的方式计算行列式就变得异常困难。在线性代数中，我们可以通过行列式的计算将一个行列式整合成一个梯形的行列式：

$$|A| = \begin{vmatrix} x_{11} & x_{12} & x_{13} \\ x_{21} & x_{22} & x_{23} \\ x_{31} & x_{32} & x_{33} \end{vmatrix} \rightarrow \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

梯形的行列式表现为，所有的数字都被整合到对角线的上方或下方（通常是上方），虽然具体的数字发生了变化（比如由x_{11}变成了a_{11}），但是**行列式的大小在初等行变换的过程中是不变的**。对于梯形行列式，行列式的计算要容易很多：

$$|A| = \begin{vmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & a_{33} \end{vmatrix}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23}a_{31} + a_{13}a_{21}a_{32} - a_{11}a_{23}a_{32} - a_{12}a_{21}a_{33} - a_{13}a_{22}a_{31}$$

$$= a_{11}a_{22}a_{33} + a_{12}a_{23} * 0 + a_{13} * 0 * 0 - a_{11}a_{23} * 0 - a_{12} * 0 * a_{33} - a_{13}a_{22} * 0$$

$$= a_{11}a_{22}a_{33}$$

不难发现，由于梯形行列式下半部分为0，整个矩阵的行列式其实就是梯形行列式对角线上的元素相乘。并且此时此刻，只要对角线上的任意元素为0，整个行列式都会为0。那只要对角线上没有一个元素为0，行列式就不会为0了。在这里，我们来引入一个重要的概念：满秩矩阵。

重要定义：满秩矩阵

满秩矩阵：A是一个n行n列的矩阵，若A转换为梯形矩阵后，没有任何全为0的行或者全为0的列，则称A为满秩矩阵。简单来说，只要对角线上没有一个元素为0，则这个矩阵中绝对不可能存在全为0的行或列。
举例来说，下面的矩阵就不是满秩矩阵，因为它的对角线上有一个0，因此它存在全为0的行。

$$\text{不是满秩矩阵：}\begin{bmatrix} a_{11} & a_{12} & a_{13} \\ 0 & a_{22} & a_{23} \\ 0 & 0 & 0 \end{bmatrix}$$

即是说，**矩阵满秩（即转换为梯形矩阵后对角线上没有0）是矩阵的行列式不为0的充分必要条件。**

- **矩阵满秩的充分必要条件**

一个矩阵要满秩，则转换为梯形矩阵后的对角线上没有0，那什么样的矩阵在转换为梯形矩阵后对角线上才没有0呢？来看下面的三个矩阵：

$$A=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix},\ B=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix},\ C=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix}$$

我们可以对矩阵做初等行变换和列变换，包括交换行/列顺序，将一列/一行乘以一个常数后加减到另一列/一行上，来将矩阵化为梯形矩阵。对于上面的两个矩阵我们可以有如下变换：

$$A=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4 \end{bmatrix},\ B=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 4.002 \end{bmatrix},\ C=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 2 & 2 & 17 \end{bmatrix} \begin{matrix} \\ \\ -\text{第一行}*2 \end{matrix}$$

$$A=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0 \end{bmatrix},\ B=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0.002 \end{bmatrix},\ C=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 13 \end{bmatrix}$$

继续进行变换：

$$A=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0 \end{bmatrix},\ B=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 0.002 \end{bmatrix},\ C=\begin{bmatrix} 1 & 1 & 2 \\ 5 & 3 & 11 \\ 0 & 0 & 13 \end{bmatrix} \begin{matrix} \\ \\ -\text{第一行}*5 \end{matrix}$$

$$A=\begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 0 \end{bmatrix},\ B=\begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 0.002 \end{bmatrix},\ C=\begin{bmatrix} 1 & 1 & 2 \\ 0 & -2 & 1 \\ 0 & 0 & 13 \end{bmatrix}$$

如此就转换成了梯形矩阵。我们可以看到，矩阵A明显不是满秩的，它有全零行所以行列式会为0。而矩阵B和C没有全零行所以满秩。而矩阵A和矩阵B的区别在于，A中存在着完全具有线性关系的两行（1，1，2和2，2，4），而B和C中则没有这样的两行。而矩阵B虽然对角线上每个元素都不为0，但具有非常接近于0的元素0.02，而矩阵C的对角线上没有任何元素特别接近于0。

矩阵A中第一行和第三行的关系，被称为**“精确相关关系”，即完全相关**，一行可使另一行为0。在这种精确相关关系下，矩阵A的行列式为0，则矩阵A的逆不可能存在。在我们的最小二乘法中，如果矩阵$\boldsymbol{X^TX}$中存在这种精确相关关系，则逆不存在，最小二乘法完全无法使用，**线性回归会无法求出结果**。

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{0}(\boldsymbol{X^TX})^*\rightarrow\text{除零错误}$$

矩阵B中第一行和第三行的关系不太一样，他们之间非常接近于”精确相关关系“，但又不是完全相关，一行不能使另一行为0，这种关系被称为**”高度相关关系“**。在这种高度相关关系下，矩阵的行列式不为0，但是一个非常接近0数，矩阵A的逆存在，不过接近于无限大。在这种情况下，最小二乘法可以使用，不过得到的逆会很大，直接影响我们对参数向量w的求解：

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{\text{非常接近0的数}}(\boldsymbol{X^TX})^*\rightarrow\infty$$

$$\boldsymbol{w}=(\boldsymbol{X^TX})^{-1}\boldsymbol{X^Ty}\rightarrow\infty$$

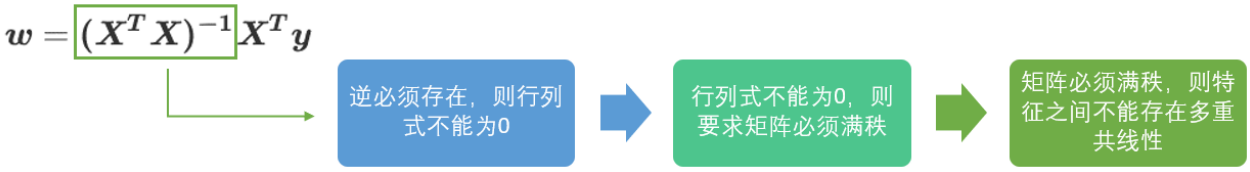
这样求解出来的参数向量w会很大，因此会影响建模的结果，造成模型有偏差或者不可用。**精确相关关系和高度相关关系并称为"多重共线性"**。在多重共线性下，模型无法建立，或者模型不可用。

相对的，矩阵C的行之间结果相互独立，梯形矩阵看起来非常正常，它的对角线上没有任何元素特别接近于0，因此其行列式也就不会接近0或者为0，因此矩阵C得出的参数向量w就不会有太大偏差，对于我们拟合而言是比较理想的。

$$(\boldsymbol{X^TX})^{-1}=\frac{1}{|\boldsymbol{X^TX}|}(\boldsymbol{X^TX})^*\rightarrow\frac{1}{\text{不是非常接近于0的常数}}(\boldsymbol{X^TX})^*\rightarrow\text{逆矩阵的大小正常}$$

$$\boldsymbol{w}=(\boldsymbol{X^TX})^{-1}\boldsymbol{X^Ty}\rightarrow\text{拟合出适合的}w$$

从上面的所有过程我们可以看得出来，**一个矩阵如果要满秩，则要求矩阵中每个向量之间不能存在多重共线性**，这也构成了线性回归算法对于特征矩阵的要求。



六、岭回归和Lasso

总的来说，解决共线性的问题的方法主要有以下两种：

- 其一是在建模之前对各特征进行相关性检验，若存在多重共线性，则可考虑进一步对数据集进行SVD分解或PCA主成分分析，在SVD或PCA执行的过程中会对数据集进行正交变换，最终所得数据集各列将不存在任何相关性。当然此举会对数据集的结构进行改变，且各列特征变得不可解释。
- 其二则是采用逐步回归的方法，以此选取对因变量解释力度最强的自变量，同时对于存在相关性的

自变量加上一个惩罚因子，削弱其对因变量的解释力度，当然该方法不能完全避免多重共线性的存在，但能够绕过最小二乘法对共线性较为敏感的缺陷，构建线性回归模型。

- 其三则是在原有的算法基础上进行修改，放弃对线性方程参数无偏估计的苛刻条件，使其能够容忍特征列存在多重共线性的情况，并且能够顺利建模，且尽可能的保证 SSE 取得最小值。

通常来说，能够利用一个算法解决的问题尽量不用多个算的组合来解决，因此此处我们主要考虑后两个解决方案，其中逐步回归我们将放在线性回归的最后一部分进行讲解，而第三个解决方案，则是我们接下来需要详细讨论的岭回归算法和Lasso算法。

1. 岭回归

1.1 基本原理

在之前讨论中，我们知道岭回归算法实际上是针对线性回归算法局限性的一个改进类算法，优化目的是要解决系数矩阵 X^TX 不可逆的问题，客观上同时也起到了克服数据集存在多重共线性的情况，而岭回归的做法也非常简单，就是在原方程系数计算公式中添加了一个扰动项 λI ，原先无法求广义逆的情况变成可以求出其广义逆，使得问题稳定并得以求解，其中 λ 是自定义参数，I 则是单位矩阵。

岭回归在多元线性回归的损失函数上加上了正则项，表达为系数w的L2范式（即系数w的平方项）乘以正则化系数λ。岭回归的损失函数的完整表达式写作：

$$\min_{w} \ {||\boldsymbol{Xw}-\boldsymbol{y}||_2}^2+\lambda{||\boldsymbol{w}||_2}^2$$

我们仍然使用最小二乘法求解，可得：

$$\hat{w}^*=(X^TX+\lambda I)^{-1}X^Ty$$

此举看似简单，实则非常精妙，用一个满秩的对角矩阵和原系数矩阵计算进行相加，实际上起到了两个作用，其一是使得最终运算结果 $(X^TX+\lambda I)$ 满秩，即降低了原数据集特征列的共线性影响，其二也相当于对所有的特征列的因变量解释程度进行了惩罚，且 λ 越大惩罚作用越强。

现在，只要$\boldsymbol{(X^TX+\lambda I)}$存在逆矩阵，我们就可以求解出$\boldsymbol{w}$。一个矩阵存在逆矩阵的充分必要条件是这个矩阵的行列式不为0。假设原本的特征矩阵中存在共线性，则我们的方阵$\boldsymbol{X^TX}$就会不满秩（存在全为零的行）：

$$\boldsymbol{X^TX}=\begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & \dots & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix}$$

此时方阵$\boldsymbol{X^TX}$就是没有逆的，最小二乘法就无法使用。然而，加上了$\lambda\boldsymbol{I}$之后，我们的矩阵就大不一样了：

$$\lambda \boldsymbol{I} = \lambda * \begin{vmatrix} 1 & 0 & 0 & \dots & 0 \\ 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & 1 \end{vmatrix} = \begin{vmatrix} \lambda & 0 & 0 & \dots & 0 \\ 0 & \lambda & 0 & \dots & 0 \\ 0 & 0 & \lambda & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & \lambda \end{vmatrix}$$

$$\boldsymbol{X^T X} + \lambda \boldsymbol{I} = \begin{vmatrix} a_{11} & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} & \dots & a_{3n} \\ & . & & & \\ 0 & 0 & 0 & \dots & 0 \end{vmatrix} + \begin{vmatrix} \lambda & 0 & 0 & \dots & 0 \\ 0 & \lambda & 0 & \dots & 0 \\ 0 & 0 & \lambda & \dots & 0 \\ & \dots & & & \\ 0 & 0 & 0 & \dots & \lambda \end{vmatrix}$$

$$= \begin{vmatrix} a_{11} + \lambda & a_{12} & a_{13} & \dots & a_{1n} \\ 0 & a_{22} + \lambda & a_{23} & \dots & a_{2n} \\ 0 & 0 & a_{33} + \lambda & \dots & a_{3n} \\ & & \dots & & \\ 0 & 0 & 0 & \dots & \lambda \end{vmatrix}$$

最后得到的这个行列式还是一个梯形行列式，然而它的已经不存在全0行或者全0列了，除非：

(1) λ等于0，或者

(2) 原本的矩阵$\boldsymbol{X^T X}$中存在对角线上元素为$-\lambda$，其他元素都为0的行或者列

否则矩阵$\boldsymbol{X^T X} + \lambda \boldsymbol{I}$永远都是满秩。在sklearn中，λ的值我们可以自由控制，因此我们可以让它不为0，以避免第一种情况。而第二种情况，如果我们发现某个λ的取值下模型无法求解，那我们只需要换一个λ的取值就好了，也可以顺利避免。也就是说，矩阵的逆是永远存在的！有了这个保障，我们的$\boldsymbol{w}$就可以写作：

$$(\boldsymbol{X^T X} + \lambda \boldsymbol{I})\boldsymbol{w} - \boldsymbol{X^T y} = 0$$

$$(\boldsymbol{X^T X} + \lambda \boldsymbol{I})\boldsymbol{w} = \boldsymbol{X^T y}$$

左乘一个$(\boldsymbol{X^T X} + \lambda \boldsymbol{I})^{-1}$则有：

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

如此，正则化系数λ就非常爽快地避免了”精确相关关系“带来的影响，至少最小二乘法在λ 存在的情况下是一定可以使用了。对于存在”高度相关关系“的矩阵，我们也可以通过调大λ，来让$\boldsymbol{X^T X} + \lambda \boldsymbol{I}$矩阵的行列式变大，从而让逆矩阵变小，以此控制参数向量w的偏移。当λ越大，模型越不容易受到共线性的影响。

$$(\boldsymbol{X^T X} + \lambda \boldsymbol{I})^{-1} = \frac{1}{|\boldsymbol{X^T X} + \lambda \boldsymbol{I}|}(\boldsymbol{X^T X} + \lambda \boldsymbol{I})^*$$

如此，多重共线性就被控制住了：最小二乘法一定有解，并且这个解可以通过λ来进行调节，以确保不会偏离太多。当然了，λ挤占了w中由原始的特征矩阵贡献的空间，因此λ如果太大，也会导致w的估计出现较大的偏移，无法正确拟合数据的真实面貌。我们在使用中，需要找出λ让模型效果变好的最佳取值。

1.2 Python实践

接下来，尝试在Python中手动编写岭回归函数，当然在借助矩阵运算的情况下岭回归函数的编写也不会太复杂，这里唯一需要注意的是对于单位矩阵的生成，需要借助NumPy中的eye函数，该函数需要输入对角矩阵规模参数。

虽说生成单位矩阵，但实际上返回的仍然是array对象，若要真正意义上执行矩阵运算，则还需要进行矩阵转化，接下来编写岭回归函数，此处默认设置岭回归系数为0.2，当该系数不为0的时候不会存在不可逆的情况，因此可免去 if 语句判别是否满秩的设置。

```
def ridgeRegres(dataSet,lam=0.2):
    xMat = np.mat(dataSet.iloc[:,:-1].values)
    yMat = np.mat(dataSet.iloc[:,-1].values).T
    xTx = xMat.T * xMat
    denom = xTx + np.eye(xMat.shape[1])*lam
    ws = denom.I * (xMat.T * yMat)
    return ws
```

然后读取`abalone`数据集进行岭回归建模，该数据集源于UCI，记录了鲍鱼的生物属性，目标字段是鲍鱼的年龄：

```
aba = pd.read_csv('abalone.csv',header=None)
aba.head()
```

其中一列为标签列，同时，由于数据集第一列是分类变量，且没有用于承载截距系数的列。为了简化教学流程，此处直接将第一列的值全部修改为1，然后在进行建模：

```
aba.iloc[:,0]=1
aba.head()
```

接着带入模型进行计算并返回各列系数：

```
rws = ridgeRegres(aba)
rws
```

由于数据集本身性质满足最小二乘法求解条件，因此可对比最小二乘法求解结果：

```
standRegres(aba)
```

能够发现由于惩罚因子存在，最终输出结果存在细微差别。同时我们可比较二者模型的评价指标：

```
rSquare(aba,ridgeRegres)
rSquare(aba,standRegres)
```

这里由于数据集本身性质原因，岭回归和线性回归返回结果并没有太大区别。接下来，调用scikit-learn中岭回归算法验证手动建模有效性。

```
from sklearn.linear_model import Ridge

ridge  = Ridge(alpha = 0.2)
ridge.fit(aba.iloc[:,:-1],aba.iloc[:,-1])
```

查看模型相关参数：

```
ridge.coef_  #查看系数

ridge.intercept_  # 查看截距
```

对比手动建模结果，能够看出结果基本保持一致。

接着我们使用交叉验证来选择最佳的正则化系数。在scikit-learn中，我们有带交叉验证的岭回归可以使用，我们来看一看：

class `sklearn.linear_model.RidgeCV`(*alphas=(0.1, 1.0, 10.0), fit_intercept=True, normalize=False, scoring=None, cv=None, gcv_mode=None, store_cv_values=False*)

可以看到，这个类与普通的岭回归类Ridge非常相似，不过在输入正则化系数α的时候我们可以传入元组作为正则化系数的备选，非常类似于我们在画学习曲线前设定的for i in 的列表对象。来看RidgeCV的重要参数，属性和接口：

重要参数	**含义**
alphas	需要测试的正则化参数的取值的元祖
scoring	用来进行交叉验证的模型评估指标，默认是R^2，可自行调整
store_cv_values	是否保存每次交叉验证的结果，默认False
cv	交叉验证的模式，默认是None，表示默认进行**留一交叉验证** 可以输入Kfold对象和StratifiedKFold对象来进行交叉验证 注意，仅仅当为None时，每次交叉验证的结果才可以被保存下来 当cv有值存在（不是None）时，store_cv_values无法被设定为True
重要属性	**含义**
alpha_	查看交叉验证选中的alpha
cv_values_	调用所有交叉验证的结果，只有当store_cv_values=True的时候才能够调用，因此返回的结构是(n_samples, n_alphas)
重要接口	**含义**
score	调用Ridge类不进行交叉验证的情况下返回的R平方

这个类的使用也非常容易，依然使用我们之前建立的鲍鱼年龄数据集：

```
from sklearn.linear_model import RidgeCV

Ridge_ = RidgeCV(alphas=np.arange(1,1001,100)
                 ,scoring="r2"
                  ,store_cv_values=True
                 #,cv=5
                 ).fit(aba.iloc[:,:-1],aba.iloc[:,-1])

#无关交叉验证的岭回归结果
Ridge_.score(aba.iloc[:,:-1],aba.iloc[:,-1])

#调用所有交叉验证的结果
Ridge_.cv_values_.shape

#进行平均后可以查看每个正则化系数取值下的交叉验证结果
Ridge_.cv_values_.mean(axis=0)

#查看被选择出来的最佳正则化系数
Ridge_.alpha_
```

2. Lasso回归

2.1 基本原理

在岭回归中，对自变量系数进行平方和处理也被称作L2正则化，由于此原因，领回归中自变量系数虽然会很快衰减，但很难归为零，且存在共线性的时候衰减过程也并非严格递减，这就是为何岭回归能够建模、判断共线性，但很难进行变量筛选的原因。为了弥补岭回归在这方面的不足，Tibshirani（1996）提出了Lasso（The Least Absolute Shrinkage and Selectionatoroperator）算法，将岭回归的损失函数中的自变量系数L2正则化修改为L1正则化，即Lasso回归损失函数为：

$$\min_{w} \ {||\boldsymbol{X}\boldsymbol{w}-\boldsymbol{y}||_2}^2+\lambda||\boldsymbol{w}||_1$$

我们来看看Lasso的数学过程。当我们使用最小二乘法来求解Lasso中的参数w，我们依然对损失函数进行求导：

$$\frac{\partial(RSS+||\boldsymbol{w}||_1)}{\partial \boldsymbol{w}}=\frac{\partial(\ ||\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w}||_2{}^2+\lambda||\boldsymbol{w}||_1)}{\partial \boldsymbol{w}}$$
$$=\frac{\partial(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})^T(\boldsymbol{y}-\boldsymbol{X}\boldsymbol{w})}{\partial \boldsymbol{w}}+\frac{\partial\lambda||\boldsymbol{w}||_1}{\partial \boldsymbol{w}}$$

前半部分我们推导过，后半部分对w求导和岭回归有巨大的不同

假设所有的系数都为正：

$$=0-2\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}+2\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}+\lambda$$

将含有w的项合并，其中λ为常数

为了实现矩阵相加，让它乘以一个结构为$n*n$的单位矩阵I：

$$=\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}-\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}+\frac{\lambda \boldsymbol{I}}{2}$$
$$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}\boldsymbol{w}=\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}-\frac{\lambda \boldsymbol{I}}{2}$$

大家可能已经注意到了，现在问题又回到了要求$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$的逆必须存在。在岭回归中，我们通过正则化系数λ能够向方阵$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$加上一个单位矩阵，以此来防止方阵$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$的行列式为0，而现在L1范式所带的正则项λ在求导之后并不带有w这个项，因此它无法对$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$造成任何影响。也就是说，**Lasso无法解决特征之间"精确相关"的问题**。当我们使用最小二乘法求解线性回归时，如果线性回归无解或者报除零错误，换Lasso不能解决任何问题。

岭回归 vs Lasso
岭回归可以解决特征间的精确相关关系导致的最小二乘法无法使用的问题，而Lasso不行。

幸运的是，在现实中我们其实会比较少遇到"精确相关"的多重共线性问题，大部分多重共线性问题应该是"高度相关"，而如果我们假设方阵$\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X}$的逆是一定存在的，那我们可以有：

$$\boldsymbol{w}=(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{X})^{-1}(\boldsymbol{X}^{\boldsymbol{T}}\boldsymbol{y}-\frac{\lambda \boldsymbol{I}}{2})$$

通过增大λ，我们可以为w的计算增加一个负项，从而限制参数估计中w的大小，而防止多重共线性引起的参数w被估计过大导致模型失准的问题。**Lasso不是从根本上解决多重共线性问题，而是限制多重共线性带来的影响**。

2.2 Python实现

接下来，尝试在scikit-learn中执行Lasso算法，仍然是采用abalone数据集，Lasso算法模型也同样是保存在linear_model模块中。

```
from sklearn.linear_model import Lasso

las = Lasso(alpha=0.01)
las.fit(aba.iloc[:,:-1],aba.iloc[:,-1])
```

然后查看执行结果：

```
las.coef_

las.intercept_
```

不同于岭回归，Lasso算法的处理结果中自变量系数会更加倾向于迅速递减为0，因此Lasso算法在自变量选择方面要优于岭回归。

能够看出Lasso回归自变量衰减速度非常快，当 λ 取0.01的时候就已经出现部分自变量系数为0的情况，从中也能看出自变量的相对重要性。

七、结语

本章之中，大家学习了多元线性回归，岭回归，Lasso三个算法，它们都是围绕着原始的线性回归进行的拓展和改进。其中岭回归和Lasso是为了解决多元线性回归中使用最小二乘法的各种限制，主要用途是消除多重共线性带来的影响并且做特征选择。除此之外，本章还定义了多重共线性和各种线性相关的概念，并为大家补充了一些线性代数知识。回归算法属于原理简单，但操作困难的机器学习算法，在实践和理论上都还有很长的路可以走，希望大家继续探索，让线性回归家族中的算法真正成为大家的武器。