01+LinearRegression

2019 年 3 月 24 日

1 多元回归

1.1 一、概念

回归（regression）是一种估计变量关系的统计方法。

通过回归可以建立因变量（Y）和自变量（X）的数量关系，以此进行预测、控制。

因变量（Y）通常是连续实数，自变量（X）可以是连续的也可以是离散的。

自变量和因变量之间的关系用矩阵进行刻画：$Y = X\beta + \epsilon$，其中 Y 是 $n \times 1$ 的向量，n 代表样本数，X 是 $n \times (p+1)$ 维样本矩阵，p 是自变量个数，额外一列是偏置项 $x_{i0} = 1$，β 是 $(p+1) \times 1$ 维系数向量，ϵ 是随机扰动，独立同分布于均值为零的正态分布 $N(0, \sigma^2)$。

用分量刻画则是：$y_i = \beta_0 + \beta_1 \times x_1 + ... + \beta_p \times x_p + \epsilon_i$，$\beta_i$ 是各分量的系数，ϵ_i 是第 i 个样本的随机扰动。

1.2 二、基本假定

1. $r(X) = p + 1 < n$，X 是一个满秩矩阵，且列秩线性无关（变量之间不线性相关），行秩大于列秩（样本数大于变量数 +1）$x_1, x_2, ..., x_p$ 是确定变量

2. 随机误差项具有零均值和同方差：

$$\begin{cases} E(\epsilon_i) = 0 & i = 1, 2, ..., n \\ cov(\epsilon_i, \epsilon_j) = \begin{cases} \sigma^2 & i = j \\ 0 & i \neq j \end{cases} \end{cases}$$

不同样本的随机扰动协方差为零表示不同的样本不相关，在正态假定下即是相互独立

3. 正态性假定：

$$\begin{cases} \epsilon_i \sim N(0, \sigma^2) & i = 1, 2, ..., n \\ \epsilon_1, \epsilon_2, ... \epsilon_n i.i.d \end{cases}$$

由以上假定，结合多元正态分布定义可知

$$
\begin{aligned}
E(Y) &= X\beta \\
var(Y) &= \sigma^2 E \\
Y &\sim N(X\beta, \sigma^2 E)
\end{aligned}
$$

1.3 三、求解

回归系数的求解是回归问题的关键，求解系数矩阵有可以推导的**正规方程**（Normal Equation），也可以根据**损失函数**（loss function）通过**梯度下降**（Gradient Descent）等优化方法极小化损失函数求解。

损失函数是衡量**估计值** $\hat{Y}$（回归结果）和真实数据 Y 差距的函数，在回归问题中常用的损失函数是均方误差（MSE, mean square error）:$\frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

$$
\begin{aligned}
loss &= \frac{1}{2}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2 \\
&= \frac{1}{2}\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 \times x_1 - ... - \beta_p \times x_p)^2
\end{aligned}
$$

$$
\begin{cases}
\frac{\partial loss}{\partial \beta_0} = -\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 \times x_1 - ... - \beta_p \times x_p) \\
\frac{\partial loss}{\partial \beta_1} = -\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 \times x_1 - ... - \beta_p \times x_p) \times x_1 \\
... \\
\frac{\partial loss}{\partial \beta_p} = -\sum_{i=1}^{n}(y_i - \beta_0 - \beta_1 \times x_1 - ... - \beta_p \times x_p) \times x_p
\end{cases}
$$

1. 正规方程

由于损失是关于系数的开口向上的二次函数，因此它的极小值总是存在的，令上式的各个导数均为零。

$$
\begin{cases}
\frac{\partial loss}{\partial \beta_0}|_{\beta_0=\hat{\beta}_0} = -\sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 \times x_1 - ... - \hat{\beta}_p \times x_p) = 0 \\
\frac{\partial loss}{\partial \beta_1}|_{\beta_1=\hat{\beta}_1} = -\sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 \times x_1 - ... - \hat{\beta}_p \times x_p) \times x_1 = 0 \\
... \\
\frac{\partial loss}{\partial \beta_p}|_{\beta_p=\hat{\beta}_p} = -\sum_{i=1}^{n}(y_i - \hat{\beta}_0 - \hat{\beta}_1 \times x_1 - ... - \hat{\beta}_p \times x_p) \times x_p = 0
\end{cases}
$$

整理成向量形式即是：

$$
\begin{aligned}
X^T(Y - X\beta) &= 0 \\
X^T Y &= X^T X\beta \\
\beta &= (X^T X)^{-1} X^T Y
\end{aligned}
$$

2. 梯度下降

梯度的本意是一个向量，表示某一函数在该点处的方向导数沿着该方向取得最大 (小) 值，即函数在该点处沿着该方向（此梯度的方向）变化最快，变化率（梯度的模）最大。

$$\nabla\beta = \begin{bmatrix} \frac{\partial loss}{\partial \beta_0} \\ \frac{\partial loss}{\partial \beta_1} \\ \cdots \\ \frac{\partial loss}{\partial \beta_p} \end{bmatrix} = -X^T(Y - \hat{Y})$$

$$Repeat\ until\ convergence: \beta := \beta - step \times \nabla\beta$$

$$\Leftrightarrow Repeat\ for\ every\ j\ until\ convergence: \beta_j := \beta_j + step \times \sum_{i=1}^{n} x_{ij}(y_i - \hat{y}_i)$$

1.4 四、应用

应用数据集是经典的红酒数据集的扩大和更新，因变量是品质（quality），适用于回归和分类问题。

In [7]:
```
import pandas as pd
import numpy as np
np.random.seed(2099)

white = pd.read_csv('data_set/winequality-white.csv', sep=';')
white.head(5)
```

Out[7]:

	fixed acidity	volatile acidity	citric acid	residual sugar	chlorides	free sulfur dioxide	total sulfur dioxide	density	pH	sulphates	alcohol	quality
0	7.0	0.27	0.36	20.7	0.045	45.0	170.0	1.0010	3.00	0.45	8.8	6
1	6.3	0.30	0.34	1.6	0.049	14.0	132.0	0.9940	3.30	0.49	9.5	6
2	8.1	0.28	0.40	6.9	0.050	30.0	97.0	0.9951	3.26	0.44	10.1	6
3	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40		
4	7.2	0.23	0.32	8.5	0.058	47.0	186.0	0.9956	3.19	0.40		

```
3       9.9         6
4       9.9         6
```

```
In [8]: x = white.drop(['quality'], axis=1)
        x.insert(0, 'bias', 1)
        y = white['quality']

        n = white.shape[0]
        p = white.shape[1]

        x = np.array(x).reshape([n, p])
        y = np.array(y).reshape([n, 1])

        np.random.seed(99)
        index = np.random.permutation(n)

        n_train = int(0.7*n)
        n_test = n - n_train

        train_index = index[0:n_train]
        test_index = index[n_train:n]

        train_x = x[train_index,:]
        train_y = y[train_index,:]

        test_x = x[test_index,:]
        test_y = y[test_index,:]

        print(white.shape)
```

```
(4898, 12)
```

1. 正规方程

利用公式直接求解

```
In [9]: def normal_equation(x, y):
            computation = np.dot(x.T, x)
            computation = np.linalg.inv(computation)
            computation = np.dot(computation, x.T)
            beta = np.dot(computation, y)
```

```
            return beta

        beta_1 = normal_equation(train_x, train_y)
        print(beta_1)
```

```
[[ 2.41258269e+02]
 [ 1.31908754e-01]
 [-1.81022513e+00]
 [ 9.58902831e-02]
 [ 1.10948761e-01]
 [ 3.05027005e-01]
 [ 3.41232024e-03]
 [-8.14510194e-05]
 [-2.42591122e+02]
 [ 1.03315525e+00]
 [ 6.72193199e-01]
 [ 8.30258911e-02]]
```

计算训练集和测试集损失——-均方误差。

在处理均方误差时将平方项的和除以样本数进行规范化，这是因为样本越多损失越大，可以大胆假设一下某个样本集合的损失为定值，那么将样本集合的每个样本重复一遍，损失将会变成原来的 2 倍，尽管本质上两个样本没有显著区别。

为了对比回归结果在训练集和测试集上的表现，因此需要对损失函数值除以样本数。

```
In [10]: def mse(y, y_hat):
             diff = y-y_hat
             MSE = 0.5*np.dot(diff.T, diff)
             return MSE

         train_y_hat = np.dot(train_x, beta_1)
         test_y_hat = np.dot(test_x, beta_1)

         train_loss = mse(train_y, train_y_hat)
         print(train_loss/train_y.shape[0])
```

```
[[0.2767015]]
```

```
In [6]: test_loss = mse(test_y, test_y_hat)
        print(test_loss/test_y.shape[0])
```

```
[[0.28249369]]
```

可以看出，训练集和测试集的损失差别不大，可以认为算法在两个数据集上表现相当，具有良好的泛化能力。

2. 梯度下降求解

梯度下降需要初始化系数矩阵，并在此基础上不断迭代。

```
In [7]: def initialize(p):
            return np.random.randn(p).reshape([p, 1])
```

```
In [8]: beta_2 = initialize(p)
        print(beta_2)
```

```
[[-1.58879755]
 [ 0.27808812]
 [ 0.68357992]
 [-1.12382112]
 [ 1.29198908]
 [-0.50970193]
 [-0.1843707 ]
 [ 0.68305206]
 [ 1.00439464]
 [ 1.86755984]
 [-1.41410639]
 [-0.92758716]]
```

梯度下降的收敛判断通常是通过损失函数的下降值小于某个阈值 δ，推测这种习惯应该源于数学分析。

梯度下降中有两个需要手动设置的超参数，阈值和步长。

```
In [9]: def gradient_descent(x, y, beta, delta=0.0001, step=0.5):
            y_hat = np.dot(x, beta)

            loss_aft = mse(y, y_hat)
            loss_pre = None

            decay = 1

            while decay>delta:
```

```
            loss_pre = loss_aft

            gradient = np.dot(x.T, y_hat - y)/y.shape[0]
            beta -= step*gradient

            y_hat = np.dot(x, beta)
            loss_aft = mse(y, y_hat)

            decay = loss_pre - loss_aft

            #print(loss_pre)
            #print(loss_aft)
            #print('decay:'+str(decay))
            #print('loss:'+str(loss_aft))

        return beta, loss_aft

    beta_2, train_loss_2 = gradient_descent(train_x, train_y, beta_2)
```

```
In [10]: beta_2 = initialize(p)
         beta_2, train_loss_2 = gradient_descent(train_x, train_y, beta_2,
                                                 delta=0.1, step=0.00005)
```

```
In [11]: print(train_loss_2/train_y.shape[0])
```

```
[[0.66703132]]
```

可以看出梯度下降的要点在于步长、收敛条件等超参数的设置，步长过长会使得参数变化过大导致损失上升，这在二次函数中十分容易理解，譬如 $y = 2x^2$，在 $y_1 = y(x_1 = 1) = 2$ 处的梯度为 4，步长若设置成 1，则 $x_2 = x_1 - 4 \times 1 = -3$，$y_2 = y(x_2 = -3) = 18$，递推下去函数值会越来越大。相应地，如果步长设置为 0.5，则 x 将永远在 +1，-1 之间变动。

当我们选取了一个较大的步长时损失可能根本不会下降，当我们选取了一个较大的损失变动阈值和较小的步长时，参数每次变动较小，可能没有训练完全，也即欠拟合。只有步长和阈值均较小时，才能使得参数训练完全，但是同时需要耗费较长时间。

```
In [1]: def gradient_descent_2(x, y, beta, delta=0.1, step=0.00001):
            y_hat = np.dot(x, beta)

            loss_aft = mse(y, y_hat)
            loss_pre = None
```

```
            decay = 1
            i=0

            while decay>delta:
                i+=1
                loss_pre = loss_aft

                gradient = np.dot(x.T, y_hat - y)/y.shape[0]
                beta -= gradient*(1+1/(2+np.exp(y.shape[0]-i)))*step

                y_hat = np.dot(x, beta)
                loss_aft = mse(y, y_hat)

                decay = loss_pre - loss_aft

                #print(i)
                #print('decay:'+str(decay))
                #print('loss:'+str(loss_aft))
                #spoil alert: 加了 print 之后整个文件大小会激增，而且运行时间也会翻倍

            return beta, loss_aft
```

```
In [14]: np.random.seed(2099)
         beta_2 = initialize(p)
         beta_2, train_loss_2 = gradient_descent_2(train_x, train_y, beta_2,
                                                   delta=0.0001, step=0.00005)
         print(train_loss_2/train_y.shape[0])
```

```
c:\users\37922\appdata\local\programs\python\python36\lib\site-packages\ipykernel_launcher.py:15:
  from ipykernel import kernelapp as app


[[0.30090803]]
```

梯度下降有多种改进方法，主要集中在梯度和步长两个角度，从梯度出发的优化在神经网络中运用较多，比如增加动量（momentum）等，这种优化是为了避免困在局部最优解；由于均方误差函数的性质，回归问题存在唯一的最优解，且前期梯度大，后期梯度小形似一个底部平坦四周陡峭的盆地，因此主要考虑通过增加一个单调递增的函数使步长变大的方法加快后期的训练。

这里采用的是类似于 logistics 回归的函数形式，采用 $\frac{1}{2+e^{-x}}$ 的形式（分母常数如果是 1 依然存在梯度过大的问题），函数有上确界 1/2 使后期步长相对于不加函数之前长了 0.5 倍。

对于多元回归而言求最优解仍然是**建议选用正规方程**求解

1.5 五、回归方程的显著性检验

线性方程是统计学中最精华方法最多的部分，检验回归方程的显著性更多是统计学方面的内容，在机器学习领域体现较少，但作为回归不可分割的一部分，仍然值得介绍。

对多元线性回归的检验主要有两种：一是针对回归方程**整体是否显著**的 **F 检验**、二是回归**系数是否显著**的 **t 检验**。

1.5.1 1. F 检验

1）抽样分布-F 分布 F 分布定义为两个卡方分布除以自由度的比值：

$$F = \frac{\chi_1^2/m}{\chi_2^2/n} \sim F(m,n)$$

其中卡方分布是多个独立标准正态分布的平方和，其自由度就是正态分布的个数：

$$\chi^2(n) = X_1^2 + X_2^2 + \cdots + X_n^2, \ \ X_i \ i.i.d \ N(0,1)$$

2）平方和分解 回忆方差的表达式：

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

我们取求和部分进行分解：

$$\begin{aligned}\sum_{i=1}^{n}(y_i-\bar{y})^2 &= \sum_{i=1}^{n}(y_i-\hat{y}_i+\hat{y}_i-\bar{y}) \\ &= \sum_{i=1}^{n}(y_i-\hat{y}_i)^2 + \sum_{i=1}^{n}(\hat{y}_i-\bar{y})^2 + \sum_{i=1}^{n}2(y_i-\hat{y}_i)(\hat{y}_i-\bar{y}) \\ &= \sum_{i=1}^{n}(y_i-\hat{y}_i)^2 + \sum_{i=1}^{n}(\hat{y}_i-\bar{y})^2\end{aligned}$$

其中，- $\sum_{i=1}^{n}(y_i-\bar{y})^2$ 称为总平方和，SST（Sum of Square for Total），

- $\sum_{i=1}^{n}(\hat{y}_i-\bar{y})^2$ 称为回归平方和，SSR（Sum of Square for Regression），
- $\sum_{i=1}^{n}(y_i-\hat{y}_i)^2$ 称为残差平方和，SSE（Sum of Square for Error），

第二行到第三行等式成立的条件是（5）方程组。

3） 对多元回归模型的显著性检验就是看各个变量 $x_1, x_2, ..., x_p$ 整体上是否对随机变量 y 具有明显的影响，因此检验问题的原假设为：

$$H_0: \ \beta_1 = \beta_2 = \cdots = \beta_p = 0$$

备择假设是其否命题，即系数不全为 0。

构造 F 统计量：

$$F = \frac{SSR/p}{SSE/(n-p-1)}$$

F 统计量服从 F（p, n-p-1），确定显著性水平后查表即可判断模型是否显著

1.5.2 2. t 检验

1）抽样分布-t 分布 设随机变量 X_1，X_2 独立，且 $X_1 \sim N(0,1)$，$X_2 \sim \chi^2(n)$，

$$t = \frac{X_1}{\sqrt{X_2/n}}$$

为自由度为 n 的 t 分布，$t \sim t(n)$

2）假设检验 回归系数的显著性检验是针对单个系数，检验这个变量是否对随机变量 y 有影响，原备择假设为：

$$H_0 : \beta_i = 0, \quad H_1 : \beta_i \neq 0$$

如果接受原假设则变量 x_i 对 y 的影响不显著，拒绝原则设则是有显著影响。
回归方程的系数向量满足多元正态分布：

$$\beta \sim N(\beta, \sigma^2(X^TX)^{-1})$$

记

$$(X^TX)^{-1} = (c_{ij}), i, j = 1, \ldots, p$$

于是有

$$E(\hat{\beta}_j) = \beta_j, Var(\hat{\beta}_j) = c_{jj}\sigma^2$$
$$\hat{\beta}_j \sim N(\beta_j, c_{jj}\sigma^2), j = 1, \ldots, p$$

据此可以构造 t 统计量

$$t_j = \frac{\hat{\beta}_j}{\sqrt{c_{jj}}\hat{\sigma}}$$

其中

$$\hat{\sigma} = \sqrt{\frac{1}{n-p-1}\sum_{i=1}^{n} e_i^2} = \sqrt{\frac{1}{n-p-1}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

1.5.3 3. 拟合优度

拟合优度是衡量回归方程对样本观测程度的量，它的定义很好理解，即是模型可解释的偏差比上数据总的偏差。这里的偏差不是模型偏差-方差权衡（Bias-Variance Tradeoff）的偏差，而是指预测值和观测值远离均值的程度。

也有将拟合优度称为样本决定系数的说法，这也反映了拟合优度是样本各个变量能决定的（能从样本的变量中学习到的）信息占总信息的比值;

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

在实际情况中 R 方能达到 0.7 已经属于拟合程度很好的情况，一些过高的 R 方反而不能说明模型很好，倒【有可能】说明数据是编造得到的。

我们还没有讨论到正则化，事实上增加变量几乎总是会改善模型的表现，或者说提升 R 方的值。正如我们先前做的 t 检验，一些变量或许会提升表现，但是可能是不显著的，因此我们不能断言变量越多模型就越好，在未知的数据上多余的变量反而可能会拖后腿，因此有了调整后的 R 方（Adjusted R-square），能对非显著的变量进行惩罚。

$$Adjusted - R^2 = 1 - \frac{(1-R^2)(n-1)}{n-p-1}$$

```
In [22]: def f_test(y, y_hat):
             dim = y.shape
             dim_hat = y_hat.shape
             if dim == dim_hat:
                 ssr = np.var(y_hat)*dim[0]
                 sse = np.dot((y-y_hat).T, y-y_hat)
                 f = (ssr/dim[1])/(sse/(dim[0]-dim[1]-1))
                 return f
             else:
                 print("Dimension not match")

         f_test(train_y, train_y_hat)
```

```
Out[22]: array([[1392.96660424]])
```

```
In [30]: def t_test(x, y, beta, y_hat):
             n, p = x.shape
             c = np.linalg.inv(np.dot(x.T, x))
             c = np.diag(c).reshape(p, 1)
             sse = np.dot((y-y_hat).T, y-y_hat)
             sigma = np.sqrt(sse/(n-p-1))
             t = beta/c/sigma
             return t
```

```
t_test(train_x, train_y, beta_1, train_y_hat)
```

```
(12, 1)
```

```
Out[30]: array([[ 2.42875825e-01],
                [ 1.34259783e+02],
                [-7.36466390e+01],
                [ 5.28961961e+00],
                [ 7.90648232e+02],
                [ 5.00916520e-01],
                [ 2.59578235e+03],
                [-2.97790532e+02],
                [-2.38010893e-01],
                [ 4.41834376e+01],
                [ 3.40475709e+01],
                [ 5.17330563e+01]])
```

```
In [31]: def adjusted_r2(y, y_hat):
             n, p = y.shape
             ssr = np.var(y_hat)*n
             sst = np.dot((y-y_hat).T, y-y_hat)
             r2 = ssr/sst
             adj_r2 = 1-(1-r2)*(n-1)/(n-p-1)
             return r2, adj_r2

         adjusted_r2(train_y, train_y_hat)
```

```
Out[31]: (array([[0.40658687]]), array([[0.40641366]]))
```

参考资料：

[1] 何晓群等应用回归分析（第三版）[M]. 北京：中国人民大学出版社，2011

```
In [ ]:
```