ML

机器学习练习代码

Linear Regression

理论基础

最小二乘法

n	m	h(x)	J(θ)	θ	X	x
特征值数量	样本数量	预测值(函数)	损失函数	参数(搞了半天就是在求这个)	特征值矩阵(m×n)	单个特征值，一般来说带个下标

$$X = \begin{bmatrix} x_{1,1} & x_{1,2} & x_{1,3} & \cdots & x_{1,n} \\ x_{2,1} & x_{2,2} & x_{2,3} & \cdots & x_{2,n} \\ x_{3,1} & x_{3,2} & x_{3,3} & \cdots & x_{3,n} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ x_{m,1} & x_{m,2} & x_{m,3} & \cdots & x_{m,n} \end{bmatrix}$$

一般来说，x_i就是指一行的x，即上面矩阵中的$\sum_{j=1}^{n} x_{i,j}$

预测函数h(x)如下

$$h_\theta(x) = \sum_{i=1}^{m} x_i \theta_i$$

上面这个h(x)是预测值，但是我们该如何调整θ的值让h(x)预测出的值与真实值y更靠近呢?

这就要说到最小二乘法了。最小二乘法是指下面的这个函数:

$$J(\theta) = \frac{1}{2}(h_\theta(x) - y)^2$$

其实就是要让h(x)和y作差，然后让这个误差更加小就好了，平方可以消除作差后正负号带来的问题(怎么感觉有点像方差了...)，这个操作就是最小二乘法中的“二乘”了。上面的这个函数就叫做**损失函数**了。要注意这是一个关于θ的函数，而不是关于x的，因为x是数据给你的，数据中x是什么那x就是什么，在建模的时候x就像一堆常数一样。

但是实际中，样本肯定不止一个，我们要让我们的θ适合所有的样本才行呀，所以我们得把所有训练样本的损失函数值都加起来才行。所以下面这个才是损失函数的完全体：

$$J(\theta)=\frac{1}{2}\sum_{i=1}^{m}(h_\theta(x_i)-y_i)^2$$

那怎样才能让这个损失函数的值最小呢？高中知识告诉我们，一维的情况下，直接求导，然后令导数等于0，判断单调性，然后求出变量值不就好了？确实这是一个数学上的好方法。在一个样本的情况下，这个方法拟合效果非常好。但是这也会有一个问题了，那就是会求出的θ只适合于这一个样本，若来一个有点不太一样的样本就血崩了。

所以我们在让J(θ)最小的时候要考虑全部训练样本的情况，而不应该只考虑个体。所以我们可以选择让他慢慢向极值点(也就是一维上导数为0的点)靠近，这样就算J(θ)损失函数的样子有小幅度变化也可以适应过来，而不会出现只适用于一个或少数样本的**过拟合**状态。

现在我们明白要慢慢靠近了，但是具体该如何靠近呢？我们不能只考虑一维的情况了，因为有很多时候特征值x都不止一个。而多元函数的极值点怎么求呢？这个问题的答案我们可以在高等数学下册中找到，那就是求**梯度**。

下面就开始说**梯度下降**了

梯度下降是一个常常见到的方法。梯度可以理解为多元函数f(x1,x2,…,xn)中使函数值下降最快的一个**方向**，既然是方向，那就是一个**向量**了。梯度的具体定义如下：

$$\nabla=(\frac{\partial f(x_1,x_2,\cdots,x_n)}{\partial x_1},\frac{\partial f(x_1,x_2,\cdots,x_n)}{\partial x_2},\cdots,\frac{\partial f(x_1,x_2,\cdots,x_n)}{\partial x_n})$$

所以我们就让θ减去其梯度，这样不就能让J(θ)慢慢减小了嘛。所以θ的操作如下

$$\theta_j=\theta_-\frac{\partial J(\theta)}{\partial \theta_j}$$

但是很多时候，求偏导之后的值可能还是会很大，所以一般我们会在偏导前面再乘一个k当作其步长，然后我们可以修改这个k的值来人为地控制每一步走多大，防止其越过极值点或者梯度下降得太慢导致的**欠拟合**

$$\theta_j=\theta_j-k\frac{\partial J(\theta)}{\partial \theta_j}$$

然后我们将J(θ)代入，最后求得

$$\theta_j=\theta_j-k\sum_{j=1}^{n}(h_\theta(x_i)-y_i)x_{ij}$$

我们会发现，这样每下降一次都需要把所有的样本的预测值h(x)与真实值y之差求和，在样本数量很大的情况下可能会导致速度很慢，所以我们可以考虑每次只用一个样本来下降，这样省去求和会快不少，由于每次只下降一点，所以整体上下降的趋势还是和求和的算法差不多的。

$$\theta_j=\theta_j-k(h_\theta(x_i)-y_i)x_i$$

要点

```
def run_steep_gradient_descent(X, y, alpha, theta):
    prod = np.dot(X, theta) - y
    sum_grad = np.dot(prod, X)
    theta = theta - (alpha / X.shape[0]) * sum_grad
    return theta
```

整个算法的关键就在于求这个梯度的部分。写这个的时候最好要有整体的思想，只有在万不得已的时候才用迭代。如上面这种写法只用矩阵乘法算梯度，速度会比自己写迭代快一些，而且看起来也更加简洁。相乘然后求和的操作就可以往矩阵乘法思考了。