1-linear-regression

2018 年 8 月 4 日

1 Linear regresison from scratch

很多讲机器学习的书或者课程都是从线性回归 (linear regression) 开始的, 我们也不例外. 以线性回归为起点, 可以很自然的过渡到逻辑回归 (logictic regression), 而多元逻辑回归又可以看作最为简单的神经网络 (neural network).

线性回归通过一个线性函数来拟合给定的数据点 x 和目标值 y:

$$y = w_1 \cdot x_1 + w_2 \cdot x_2 + ... + w_N \cdot x_N$$

其中 $w = (w_1, w_2, ..., w_N)^T$ 是待确定的参数.

既然是做拟合就会有多个数据点, 假设有 D 对数据点和目标值, 那么就可以得到如下线性方程组:

$$y_1 = w_1 \cdot x_{11} + w_2 \cdot x_{12} + \ldots + w_N \cdot x_{1N} \tag{1}$$

$$y_2 = w_1 \cdot x_{21} + w_2 \cdot x_{22} + \ldots + w_N \cdot x_{2N} \tag{2}$$

$$\ldots \tag{3}$$

$$y_D = w_1 \cdot x_{D1} + w_2 \cdot x_{D2} + \ldots + w_N \cdot x_{DN} \tag{4}$$

将 y_i 和 x_{ij} 分别用向量和矩阵表示为:

$$y = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_D \end{bmatrix}$$

和

$$X = \begin{bmatrix} x_{11} & x_{12} & \ldots & x_{1N} \\ x_{21} & x_{22} & \ldots & x_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ x_{D1} & x_{D2} & \ldots & x_{DN} \end{bmatrix}$$

于是上面的线性方程组就可以写作 $y = Xw$.

如果 $D \leq N$, w 的值可以通过求解上述线性方程组来确定. 当 $D = N$ 并且有解时, 方程组有唯一解 $w = X^{-1}y$. 但是在大部分情况下 $D \gg N$, 也就是说数据点的个数一般远远大于参数的个数. 这个时候方程组一般没有解. 下面我们将从两个视角讨论如何在这种情况下确定 w 的值.

为了方便接下来的讨论, 我们用 x_{*j} 表示矩阵 X 的第 j 列, 用 x_{i*} 表示矩阵 X 的第 i 行:

$$X = \begin{bmatrix} x_{*1} & x_{*2} & \dots & x_{*N} \end{bmatrix} = \begin{bmatrix} x_{1*} \\ x_{2*} \\ \vdots \\ x_{D*} \end{bmatrix}$$

1.1 最小二乘法

既然没有办法找到一个 w 使得 y 等于 Xw, 那么 y 和 Xw 必然有一个偏差 $\epsilon = y - Xw$. 现在我们需要考虑如何量化预测值 Wx 和目标值 y 的偏差. 一个显而易见的度量方法就是 ϵ 各部分的绝对值的和 $\sum_{i=1}^{D} |\epsilon_i|$. 这个和越小意味着预测值和目标值越接近. 于是可以令 w 等于如下极值问题的解:

$$\min_{w \in R^N} \sum_{i=1}^{D} |\epsilon_i|$$

即

$$\min_{w \in R^N} \sum_{i=1}^{D} |y_i - x_{i*}w|$$

由于函数 $f(x) = |x|$ 在原点附近不可导, 求解上面的极值问题并不容易.

我们可以稍作改变, 使用残差平方和 $RSS(w) = \sum_{i=1}^{D} (y_i - x_{i*}w)^2$ 来度量预测值和目标值的偏差, 从而得到一个更容易求解的问题:

$$\min_{w \in R^N} \sum_{i=1}^{D} (y_i - x_{i*}w)^2$$

这就是我们熟知的最小二乘法.

由于上述目标函数处处可导, 那么在取得最小值的时候, 目标函数 $G(w)$ 的梯度等于零 $\frac{dG(w)}{dw} = 0$. 我们先一步步求偏导:

$$\frac{dG(w)}{dw_j} = \frac{\sum_{i=1}^{D} (y_i - x_{i*}w)^2}{dw_j} \tag{5}$$

$$= \frac{\sum_{i=1}^{D} \left(y_i - \sum_{k=1}^{N} x_{ik}w_k \right)^2}{dw_j} \tag{6}$$

$$= 2 \cdot \sum_{i=1}^{D} \left(y_i - \sum_{k=1}^{N} x_{ik}w_k \right) \cdot (-x_{ij}) \tag{7}$$

注意到上面公式中的两个求和项都是向量点积, 所以可以重新写作向量形式:

$$\frac{dG(w)}{dw_j} = 2 \cdot \sum_{i=1}^{D} (y_i - x_{i*}w) \cdot (-x_{ij}) \tag{8}$$

$$= -2 \cdot x_{*j}^T \cdot (y - Xw) \tag{9}$$

梯度的表达式就是所有偏导堆积在一列:

$$\frac{dG(w)}{dw} = \begin{bmatrix} \frac{dG(w)}{dw_1} \\ \frac{dG(w)}{dw_2} \\ \vdots \\ \frac{dG(w)}{dw_D} \end{bmatrix} = -2 \cdot X^T (y - Xw)$$

令 $\frac{dG(w)}{dw} = 0$ 可以得到:

$$-2 \cdot X^T (y - Xw) = 0 \tag{10}$$

$$X^T (y - Xw) = 0 \tag{11}$$

$$X^T y - X^T X w = 0 \tag{12}$$

$$X^T X w = X^T y \tag{13}$$

我们将上述方程称作正则方程 (noraml equation). 令 $A = X^T X$, $b = X^T y$, 正则方程可以写作 $Aw = b$. 注意到 A 是一个 $N \times N$ 的矩阵, b 是一个长度为 N 的列向量, 他们的大小只与参数的个数 N 有关, 而与数据点的个数无关. 为了得到 w 的值我们又回到求解线性方程组的问题.

接下来我们暂不讨论正则方程的解, 而先来介绍正则方程的几何解释.

1.1.1 正则方程的几何解释

首先来看一个简单的一元线性回归问题:

$$y = \beta x$$

对于这个问题我们有如下两组观测值:

x	y
1	2
2	5

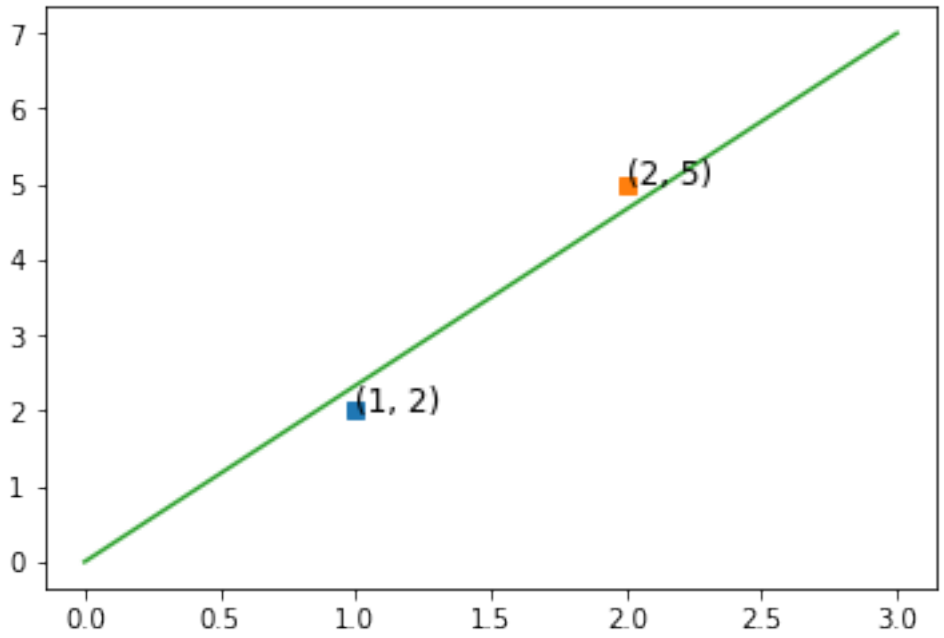
如果要将上述数据可视化, 我们一般会将每一组数据点 (即表格中的每一行) 作为二维平面上的

一个点. 那么线性回归就是要找一条尽可能靠近这些点的直线. 注意到的直线 $y = \beta x$ 一定经过原点, 如下图所示.

```
In [1]: import numpy as np
        import matplotlib.pyplot as plt
        %matplotlib inline

        p1 = (1, 2)
        p2 = (2, 5)
        line = zip(np.array([0, 0]), np.array([3, 7]))
        plt.plot(*p1, 's', *p2, 's', *line)
        plt.text(*p1, "(%d, %d)" % (*p1,), fontsize='large')
        plt.text(*p2, "(%d, %d)" % (*p2,), fontsize='large')
```

```
Out[1]: Text(2,5,'(2, 5)')
```



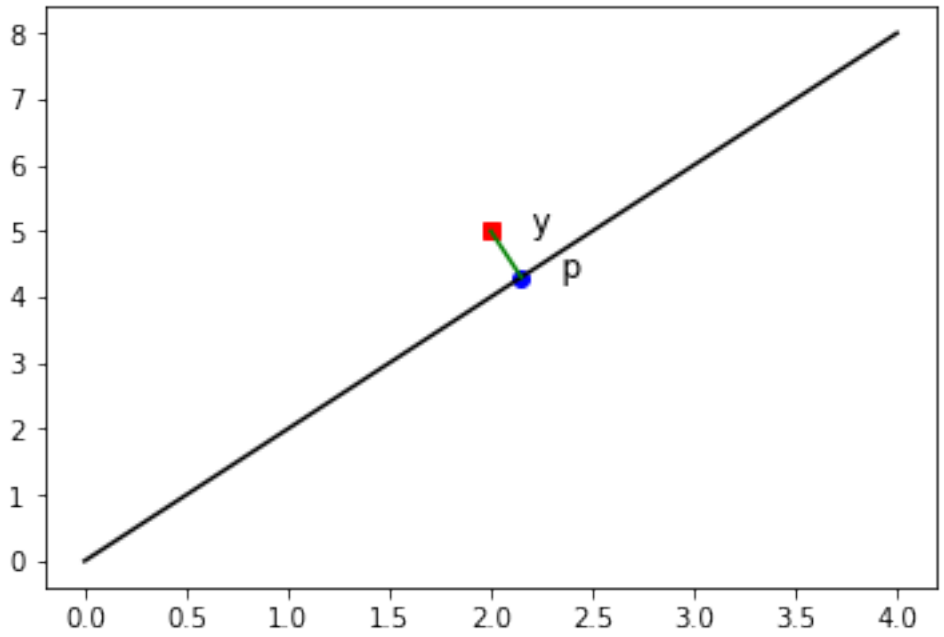
上图能给我们一些直观的感受, 比如这些点到直线的距离. 但是我们无法一眼看出哪条直线最靠近这些点.

从另一个角度来看, 我们可以去观察表格中的每一列, 这样做能给我们带来更深刻的认识. $y = (2,5)^T$ 这一列代表一个固定的点, βx 代表一条直线上的任意一个点 $p = (\beta, 2\beta)^T$. 而他们的差 $\epsilon = y - \beta x$ 代表一条连接 y 和 p 的线段. 实际上 ϵ 就是最小二乘法中定义的偏差, 所以最小二乘法做的就是通过移动 p 点 (即选取 β 的值), 让 y 和 p 尽可能的靠近.

将他们画在二维平面上就是:

```
In [2]: def plot_point_and_line(beta):
            y = np.array([2, 5])
            x = np.array([1, 2])
            z = np.array([0, 0])
            p = beta * x
            y2p = zip(y, p)
            line = zip(z, x * 4)
            plt.plot(*y, 'rs', *p, 'bo', *line, 'k-', *y2p, 'g-')
            plt.text(*y, '    y', fontsize='large')
            plt.text(*p, '    p', fontsize='large')
```

```
In [3]: plot_point_and_line(2.15)
```




```
In [4]: from ipywidgets import interactive
        interactive(plot_point_and_line, beta=(0, 4, 0.05))
```

```
interactive(children=(FloatSlider(value=2.0, description='beta', max=4.0, step=0.05), Output())
```

显然当代表偏差 ϵ 的绿色线段和黑色直线垂直的时候, 绿色线段的长度最小. 绿色线段和黑色直线垂直意味着 x 和 ϵ 的点积为零:

$$x^T(y - \beta x) = 0$$

上述方程就是一元线性回归的正则方程.

如果是多元线性回归, X 的所有列可以生成一个向量空间, Xw 便是这个向量空间中的一个向量. 我们希望 Xw 尽可能的靠近 y, 那么 Xw 就应该是 y 在该向量空间上的投影. 也就是说 $y - Xw$ 垂直于该向量空间中的任意一个向量, 它自然也和 X 的列向量垂直, 所以 $y - Xw$ 和 X 的列向量的点积都为零:

$$X^T(y - Xw) = 0$$

这正是通过最小二乘法得到的正则方程.

1.1.2 求解正则方程

由于 X^TX 是一个 $N \times N$ 的方阵, 所以求解正则方程 $X^TXw = X^Ty$ 的关键在于 X^TX 是否可逆. 如果可逆, 方程的解就是 $w = \left(X^TX\right)^{-1}X^Ty$. 接下来我们来证明只要 X 的列向量是线性独立的, X^TX 就可逆.

为了证明上述论断, 需要利用如下命题: > A^TA 和 A 有着相同的 nullspace.

矩阵 A 的 nullspace 就是方程 $Ax = 0$ 的所有解组成的向量空间. 显然如果 $Ax = 0$ 必有 $A^TAx = 0$. 也就是说 A 的 nullspace 中的向量也在 A^TA 的 nullspace 中. 另一方面, 如果 $A^TAx = 0$, 可做如下推导:

$$A^TAx = 0 \rightarrow x^TA^TAx = 0 \rightarrow \|Ax\|^2 = 0 \rightarrow Ax = 0$$

于是我们证明了这两个 nullspace 是一样的.

如果 X 的列向量是线性独立的, 那么 X 的 nullspace 就只包含 0. 应用上述命题, X^TX 的 nullspace 也只包含 0. 也就是说 X^TX 的列向量是线性独立的, 所以 X^TX 可逆.

如果 X 的列向量不是线性独立的该怎么办? X 的每一列都是针对某一特征的一系列观测值. X 的列向量不是线性独立的, 意味着其中一个特征的观测值完全可以通过对其他特征的观测值进行线性组合得到. 问题出在选取了高度相关的特征上, 可以尝试剔除掉冗余的特征.

1.2 极大似然估计

一般来说对于一个统计量, 存在着多种估计方法. 极大似然法是估计线性回归参数的另一种方法. 使用极大似然法需要对目标值 y_i 的概率分布做出假设. 在线性回归中我们假设 y_i 都是相互独立的, 并且符合正态分布 $y_i \sim \mathcal{N}(x_{i*}w, \sigma^2)$. 该分布的密度函数为:

$$\varphi(y) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(y - x_{i*}w)^2}{2\sigma^2}\right)$$

那么一组数据 (x_i, y_i) 的似然函数就是:

$$L(w) = \prod_{i=0}^{D} L\left(w|x_i, y_i\right) \tag{14}$$

$$= \prod_{i=0}^{D} \varphi(y_i) \tag{15}$$

w 的极大似然估计就是如下最大值问题的解:

$$\max_{w} L(w)$$

因为 $\log(x)$ 是单调递增函数, 所以为了计算方便, 可以最大化对数似然函数:

$$\max_{w} LL(w)$$

其中 $LL(w)$

$$LL(w) = \log L(w) \tag{16}$$

$$= \log \prod_{i=0}^{D} \varphi(y_i) \tag{17}$$

$$= \sum_{i=0}^{D} \log \varphi(y_i) \tag{18}$$

$$= \sum_{i=0}^{D} \left(\log \left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{(y - x_{i*}w)^2}{2\sigma^2} \right) \tag{19}$$

$$= D \log \left(\frac{1}{\sigma\sqrt{2\pi}} \right) - \frac{1}{2\sigma^2} \sum_{i=0}^{D} (y - x_{i*}w)^2 \tag{20}$$

$$\tag{21}$$

上述函数的第一项是常数, 在求解最大值的时候可以将其舍去. 第二项前面的常数 $\frac{1}{2\sigma^2}$ 同样不会改变最大值问题的解. 于是简化为求解如下最大值问题:

$$\max_{w} - \sum_{i=0}^{D} (y - x_{i*}w)^2$$

即

$$\min_{w} \sum_{i=0}^{D} (y - x_{i*}w)^2$$

这正是最小二乘法对应的极值问题.

参数 w 的最大似然估计和通过最小二乘法得到的值是一样的. 这是正态分布概率密度函数的表达式决定的. 如果换一个概率分布, 使用两种方法得到的估计值就可能不同. 在机器学习中, LL(w) 和 RSS(w) 被叫做损失函数, 那么恰巧这两个损失函数取极值的时候 w 的值相同.

1.3 梯度下降法

```
In [5]: num_inputs = 2
        num_outputs = 1
        num_examples = 10000
```

```
In [6]: def real_fn(X):
            return 2 * X[:, 0] - 3.4 * X[:, 1] + 4.2
```

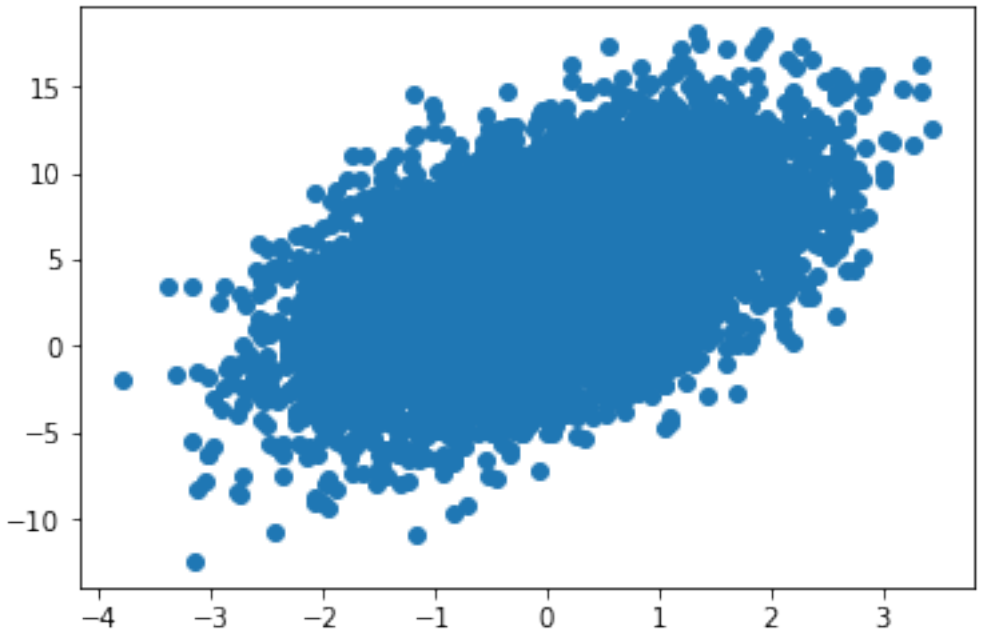
```
In [7]: x = np.random.randn(num_examples, num_inputs)
        noise = .1 * np.random.randn(num_examples)
        y = real_fn(x) + noise
```

```
In [8]: x.shape, y.shape
```

```
Out[8]: ((10000, 2), (10000,))
```

```
In [9]: plt.scatter(x[:, 0], y)
```

```
Out[9]: <matplotlib.collections.PathCollection at 0x1ef9f625fd0>
```




```
In [10]: def linear_regression(x, w, b):
             return np.dot(x, np.squeeze(w)) + b
```

```
In [11]: def square_loss(yhat, y):
             return np.mean((yhat - y) ** 2)

In [12]: def gradient(w, b, x, y):
             # obj func sum(G(w, b)) / N
             #   where G(w, b) = [(wx + b) - y].^2
             # DG(w, b) / dw
             #   = 2 * (wx + b - y) * D(wx + b - y) / dw
             #   = 2 * (wx + b - y) * x
             # DG(w, b) / db
             #   = 2 * (wx + b - y) * D(wx + b - y) / db
             #   = 2 * (wx + b - y)
             z = 2 * (np.dot(x, np.squeeze(w)) + b - y)
             #print([z.shape, x.shape, np.squeeze(w).shape])
             return np.mean(z[:, np.newaxis] * x, axis=0), np.mean(z)

In [13]: def sgd_update(w, b, x, y, rate):
             grad_w, grad_b = gradient(w, b, x, y)
         #     print(grad_w)
         #     print(grad_b)
             w[:] = w - rate * grad_w[:, np.newaxis]
             b[:] = b - rate * grad_b
             return w, b

In [14]: z = np.array(list(zip(x, y)))
         np.random.shuffle(z)
         z.size, z[0]

Out[14]: (20000,
          array([array([1.59624683, 0.88651093]), 4.565802713974116], dtype=object))

In [15]: batch_size = 4
         num_batches = num_examples / batch_size
         batches = np.array_split(z, num_batches)
         num_batches, batches[0]

Out[15]: (2500.0, array([[array([1.59624683, 0.88651093]), 4.565802713974116],
                 [array([0.16996406, 0.39040957]), 3.111253237981017],
                 [array([0.3209733 , 0.17646628]), 4.506180005847527],
                 [array([1.37973322, 0.60450899]), 4.636836359209813]], dtype=object))
```

```
In [16]: bx, by = zip(*batches[0])
         bx, by = np.array(bx), np.array(by)

In [17]: epochs = 10
         learning_rate = .0001

In [18]: w = np.random.randn(num_inputs, num_outputs)
         b = np.random.randn(num_outputs)

In [19]: for e in range(epochs):
             cumulative_loss = 0
             for batch in batches:
                 bx, by = zip(*batch)
                 bx, by = np.array(bx), np.array(by)
                 yhat = linear_regression(bx, w, b)
                 loss = square_loss(yhat, by)
                 cumulative_loss += loss
                 sgd_update(w, b, bx, by, learning_rate)
             print(cumulative_loss / num_batches)
```

```
22.958831407088788
8.411253075186549
3.0858344122525985
1.136141137498014
0.42225645670649326
0.16083666439365543
0.06509601627715997
0.030028923718637787
0.01718359276072464
0.01247785778453791
```

```
In [20]: w

Out[20]: array([[ 1.98243805],
                [-3.37367563]])

In [21]: b

Out[21]: array([4.17515803])
```