线性回归 (Linear Regression)

Guangyao Zhao

2022-01-13

Contents

数学表示符号

给定由 m 个样本，n 个属性描述的示例 $D = \{(x_1, y_1), (x_2, y_2), (x_3, y_3), ..., (x_m, y_m)\}$，$j$ 表示第 j 个样本值。其中 $x^{(j)} \in \mathbb{R}^n$，每个样本又包含 n 个维度，即 $x^{(j)} = \{x_1^{(j)}, x_2^{(j)}, ..., x_m^{(j)}\}$，$y^{(j)} \in \mathbb{R}$。

非矩阵形式

线性模型试图学习到一个通过属性的线性组合来预测的函数，即：

$$\begin{aligned} f(x) &= w_1x_1 + w_2x_2 + w_3x_3 + ... + w_nx_n + b \\ &= w^{\mathrm{T}}x + b \end{aligned} \tag{1}$$

其中：$w = \{w_1; w_2; w_3; ..., w_n\}$，$x = \{x_1; x_2; x_3; ...; x_n\}$。

线性回归试图寻求能使损失函数 (Cost function)：

$$J(w,b) = \frac{1}{2m}\sum_{j=1}^{m}\left(f\left(x^{(j)}\right) - y^{(j)}\right)^2 \tag{2}$$

最小的 w 和 b, 对 Eq. 2 求导可得:

$$\frac{\partial J_{(w,b)}}{\partial w_i} = \frac{1}{m}\sum_{j=1}^{m}\left(w^{\mathrm{T}}x^{(j)} + b - y^{(j)}\right)x_i^{(j)} \tag{3}$$

$$\frac{\partial J_{(w,b)}}{\partial b} = \frac{1}{m}\sum_{j=1}^{m}\left(w^{\mathrm{T}}x^{(j)} + b - y^{(j)}\right) \tag{4}$$

更新 w, b :

$$w_i := w_i - \alpha\frac{\partial J_{(w,b)}}{\partial w_i}$$

$$b := b - \alpha\frac{\partial J_{(w,b)}}{\partial b}$$

其中 α 为学习率。

矩阵形式

w 的表示如下:

$$w = \{w_1; w_2; w_3; ...; w_n\}$$

所有样本 $\mathbf{X}$ 的表示如下:

$$\mathbf{X} = \begin{pmatrix} x_1^{(1)} & x_2^{(1)} & ... & x_n^{(1)} & 1 \\ x_1^{(2)} & x_2^{(2)} & ... & x_n^{(2)} & 1 \\ \vdots & \vdots & \ddots & \vdots & \vdots \\ x_1^{(m)} & x_2^{(m)} & ... & x_n^{(m)} & 1 \end{pmatrix} = \begin{pmatrix} x^{(1)} & 1 \\ x^{(2)} & 1 \\ \vdots & \vdots \\ x^{(m)} & 1 \end{pmatrix}$$

y 的的标记如下:

$$y = \{y^{(1)}; y^{(2)}; y^{(3)}; ...; y^{(m)}\}$$

目标函数的推导如下:

$$
\begin{aligned}
\mathrm{L}(w) &= \frac{1}{2m}\sum_{j=1}^{m}\|w^{\mathrm{T}}x^{(j)}-y^{(j)}\|^2 \\
&= \frac{1}{2m}\left(w^{\mathrm{T}}x^{(1)}-y^{(1)}, w^{\mathrm{T}}x^{(2)}-y^{(2)}, ..., w^{\mathrm{T}}x^{(m)}-y^{(m)}\right)\begin{pmatrix} w^{\mathrm{T}}x^{(1)}-y^{(1)} \\ w^{\mathrm{T}}x^{(2)}-y^{(2)} \\ \vdots \\ w^{\mathrm{T}}x^{(1)}-y^{(m)} \end{pmatrix} \\
&= \frac{1}{2m}\left[\left(w^{\mathrm{T}}x^{(1)}, w^{\mathrm{T}}x^{(2)}, ..., w^{\mathrm{T}}x^{(m)}\right)-\left(y^{(1)}, y^{(2)}, ..., y^{(m)}\right)\right]\left[\begin{pmatrix} \left(x^{(1)}\right)^{\mathrm{T}} \\ \left(x^{(2)}\right)^{\mathrm{T}} \\ \vdots \\ \left(x^{(m)}\right)^{\mathrm{T}} \end{pmatrix} w - \begin{pmatrix} y^{(1)} \\ y^{(2)} \\ \vdots \\ y^{(m)} \end{pmatrix}\right] \\
&= \frac{1}{2m}\left(w^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}-y^{\mathrm{T}}\right)(\mathbf{X}w-y) \\
&= \frac{1}{2m}\left(\mathbf{X}w-y\right)^{\mathrm{T}}(\mathbf{X}w-y)
\end{aligned} \tag{5}
$$

则，目标函数为：

$$
\underset{w}{\operatorname{argmin}}\ \frac{1}{2m}(\mathbf{X}w-y)^{\mathrm{T}}(\mathbf{X}w-y) \tag{6}
$$

对于目标函数化简：

$$
\begin{aligned}
\mathrm{J}(w) &= \frac{1}{2m}\left(\mathbf{X}w-y\right)^{\mathrm{T}}(\mathbf{X}w-y) \\
&= \frac{1}{2m}w^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}w - w^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}y - y^{\mathrm{T}}\mathbf{X}w + y^{\mathrm{T}}y \\
&= \frac{1}{2m}w^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}\mathbf{X}w - 2w^{\mathrm{T}}\mathbf{X}^{\mathrm{T}}y + y^{\mathrm{T}}y
\end{aligned} \tag{7}
$$

对上式（即目标函数）求导：

$$
\frac{\partial\,\mathrm{L}(w)}{\partial\,w} = \frac{1}{2m}\left(2\mathbf{X}^{\mathrm{T}}\mathbf{X}w - 2\mathbf{X}^{\mathrm{T}}y\right) = 0 \tag{8}
$$

求解可得到解析解：$w = \left(\mathbf{X}^{\mathrm{T}}\mathbf{X}\right)^{-1}\mathbf{X}^{\mathrm{T}}y$。在进行梯度下降的时候也是利用该公式，$w$ 的更新公式为：

$$
w := w - \frac{\alpha}{m}\mathbf{X}^{\mathrm{T}}\left(\mathbf{X}w-y\right) \tag{9}
$$

代码

非第三方库

```

import matplotlib.pyplot as plt
import numpy as np
from sklearn.datasets import load_boston


def load_dataset():
    X, y = load_boston(return_X_y=True)
    return X, y


def normalize(X):
    return (X - np.mean(X, axis=0)) / np.std(X, axis=0)


def cost_function(X, y, w):
    m = X.shape[0] # samples
    y_pre = f(X, w)  # (m, 1)
    return float(np.matmul((y - y_pre).T, (y - y_pre)) / (2 * m))


def f(X, w):
    return np.matmul(X, w)


def gradient(X, y, w, learning_rate=0.01, epoch=100):
    J_all = []  # 存放 loss
    m = X.shape[0]
    for i in range(epoch):
        y_pre = f(X, w)
        cost_ = np.matmul(X.T, y_pre - y) / m  # 重点，以矩阵形式更新参数
        w = w - learning_rate * cost_
```

```
        J_all.append(cost_function(X, y, w))
    return w, J_all


def plot_cost(J_all):
    fig = plt.figure(figsize=(8, 6),
                     facecolor='pink',
                     frameon=True,
                     edgecolor='green',
                     linewidth=2)
    ax1 = fig.add_subplot(111, xlabel='Epoch', ylabel='Loss')
    ax1.plot(J_all)


# ===========
X, y = load_dataset()  # 加载数据集
X = np.array(X)
y = np.array(y).reshape((y.size, 1))

X = normalize(X)  # 标准化
X = np.hstack((X, np.ones((X.shape[0], 1))))  # 变化 X
w = np.ones((X.shape[1], 1))  # 初始化 w
learning_rate = 0.1
epoch = 100

w, J_all = gradient(X, y, w, learning_rate, epoch)
plot_cost(J_all)
```

Sklearn

一般情况下是不需要自己造轮子的，而且自己写的代码并不一定在运行速度上有优势，所以大多数情况下是直接调用第三方库，比如参考sklearn 官方文档：

```
from sklearn.linear_model import LinearRegression
from sklearn.preprocessing import StandardScaler

```

```python
def load_dataset():
    X, y = load_boston(return_X_y=True)
    return X, y

X, y = load_dataset()  # 加载数据集
X = np.array(X)
y = np.array(y).reshape((y.size, 1))

X = StandardScaler().fit_transform(X) # 归一化
reg = LinearRegression(fit_intercept=True).fit(X, y)

score = reg.score(X,y) # R2
print('score: ', score)
```

正则项

当样本数较小的时候，模型容易过拟合，此时一般有三种办法解决：

1. 增加样本量
2. 提取出有效特征，减少维度
3. 正则化

其中正则化是一种最常用且最方便的方法，正则化的思想是使参数 w 的尽量小，因为当其很大的时候容易造成不稳定，比如即使 x 稍微有点波动，由于 w 太大，也会造成 y 剧烈波动。

线性回归中，L_2 正则的代价函数如下：

$$J(w) = \sum_{j=1}^{m} \|w^{\mathrm{T}}x^{(j)} - y^{(j)}\|^2 + \lambda w^{\mathrm{T}}w \tag{10}$$

将 Eq. 10 和 Eq. 7 相结合可得 w 的更新公式：

$$w := w - (\alpha \mathbf{X}^{\mathrm{T}} (\mathbf{X}w - y) - \lambda w) \tag{11}$$

非矩阵形式的更新公式如下：

$$w_i := w_i - \left(\alpha \frac{1}{m} \sum_{j=1}^{m} (w^{\mathrm{T}}x^{(j)} - y^{(j)}) x_i^{(j)} - \lambda w_i \right)$$

```python
def load_dataset():
    X, y = load_boston(return_X_y=True)
    return X, y


def normalize(X):
    return (X - np.mean(X, axis=0)) / np.std(X, axis=0)


def cost_function(X, y, w, alpha=0.1):
    m = X.shape[0]  # samples
    y_pre = f(X, w)  # (m, 1)
    return float(np.matmul((y - y_pre).T, (y - y_pre)) /
                 (2 * m)) + alpha / 2 + float(alpha / 2 * np.matmul(w.T, w))


def f(X, w):
    return np.matmul(X, w)


def gradient(X, y, w, learning_rate=0.01, epoch=100, alpha=0.1):
    J_all = []  # 存放 loss
    m = X.shape[0]
    for i in range(epoch):
        y_pre = f(X, w)
        cost_ = np.matmul(X.T, y_pre - y) / m + alpha * w  # 重点，以矩阵形式更新参数
        w = w - learning_rate * cost_
        J_all.append(cost_function(X, y, w, alpha))
    return w, J_all


def plot_cost(J_all):
    fig = plt.figure(figsize=(8, 6),
                     facecolor='pink',
                     frameon=True,
                     edgecolor='green',
```

```
                        linewidth=2)
    ax1 = fig.add_subplot(111, xlabel='Epoch', ylabel='Loss')
    ax1.plot(J_all)
    fig.savefig('test.pdf')


# ===========
X, y = load_dataset()  # 加载数据集
X = np.array(X)
y = np.array(y).reshape((y.size, 1))

X = normalize(X)  # 标准化
X = np.hstack((X, np.ones((X.shape[0], 1))))  # 变化 X
w = np.ones((X.shape[1], 1))  # 初始化 w
learning_rate = 0.1
epoch = 1000

w, J_all = gradient(X, y, w, learning_rate, epoch, alpha=1)
print('w: ', w)
plot_cost(J_all)
```