Lec 7: Logistic Regression

Ailin Zhang

2022-10-10

Agenda

- Comments on HW2
- Logistic Regression
- Maximum Likelihood
- Gradient Ascent
- Iterated Reweighed Least Squares (IRLS)

Linear Regression by QR

We rotate the matrix (**XY**) by QR decomposition, by applying the Householder reflections for $j = 1, ..., p$,

$$\begin{bmatrix} \mathbf{X} & \mathbf{Y} \end{bmatrix} \xrightarrow{Q^\top} \begin{bmatrix} R & \mathbf{Y}^* \end{bmatrix} = \begin{bmatrix} R_1 & \mathbf{Y}_1^* \\ 0 & \mathbf{Y}_2^* \end{bmatrix},$$

where R_1 is a upper triangular squared matrix.

To solve the least squares problem,

$$\min_\beta \|\mathbf{Y}^* - R\beta\|^2 = \min_\beta \left(\| Y_1^* - R_1\beta\|^2 + \|\mathbf{Y}_2^*\|^2 \right),$$

the solution $\hat{\beta} = R_1^{-1}\mathbf{Y}_1^*$ and $\mathrm{RSS} = \|\mathbf{Y}_2^*\|^2$.

Since R_1 is an upper triangular matrix, we can solve the elements of $\hat{\beta}$ in reverse order $\hat{\beta}_p, \hat{\beta}_{p-1}, \dots, \hat{\beta}_1$. It is numerically stable and efficient.

Python Code

```
n = 100
p = 5
X = np.random.random_sample((n, p))
beta = np.array(range(1, p+1))
Y = np.dot(X, beta) + np.random.standard_normal(n)

Z = np.hstack((np.ones(n).reshape((n, 1)), X, Y.reshape((n, 1))))
_, R = qr(Z)
R1 = R[:p+1, :p+1]
Y1 = R[:p+1, p+1]
beta = np.linalg.solve(R1, Y1)
#Note: in HW, you should code this part from scratch
print(beta)
```

Logistic Regression

Consider a dataset with n training examples, where $X_i^\top = (x_{i1}, \cdots, x_{ip})$ consists of p predictors or features, $y_i \in \{0, 1\}$ is the outcome or class label.

obs	$\mathbf{X}_{n\times p}$	$\mathbf{Y}_{n\times 1}$
1	$X_1^\top$	y_1
2	$X_2^\top$	y_2
...		
n	$X_n^\top$	y_n

We assume $y_i \sim \mathrm{Bernoulli}(p_i)$, i.e., $\Pr(y_i = 1) = p_i$, and we assume

$$\mathrm{logit}(p_i) = \log \frac{p_i}{1 - p_i} = X_i^\top \beta.$$

Logistic Regression

$$\mathrm{logit}(p_i) = \log \frac{p_i}{1 - p_i} = X_i^\top \beta.$$

Let $\eta_i = X_i^\top \beta$ be the score, then

$$p_i = \sigma(\eta_i) = \frac{1}{1 + e^{-\eta_i}} = \frac{1}{1 + e^{-X_i^\top \beta}} = \frac{e^{X_i^\top \beta}}{1 + e^{X_i^\top \beta}},$$

where the function $\sigma()$ is the sigmoid function, which is the inverse of the logit function.

Maximum Likelihood

The likelihood function is

$$L(\beta) = \prod_{i=1}^{n} \Pr(y_i | p_i)$$

$$L(\beta) = \prod_{i=1}^{n} p_i^{y_i} (1 - p_i)^{1-y_i} = \prod_{i=1}^{n} \frac{e^{y_i X_i^\top \beta}}{1 + e^{X_i^\top \beta}}.$$

The log-likelihood is

$$l(\beta) = \log L(\beta) = \sum_{i=1}^{n} \left[y_i X_i^\top \beta - \log(1 + \exp X_i^\top \beta) \right].$$

The maximum likelihood is to find the most plausible explanation to the observed data.

Gradient Ascent

To find the maximum of $l(\beta)$, we first calculate the gradient

$$l'(\beta) = \sum_{i=1}^{n} \left[y_i X_i - \frac{e^{X_i^\top \beta}}{1 + e^{X_i^\top \beta}} X_i \right] = \sum_{i=1}^{n} (y_i - p_i) X_i.$$

We use gradient ascent to iteratively update β,

$$\beta^{(t+1)} = \beta^{(t)} + \gamma_t \sum_{i=1}^{n} (y_i - p_i) X_i,$$

where γ is the learning rate. This is a hill climbing algorithm, where each step we take the steepest direction uphill.

Gradient Descent

If we minimize a loss function such as $-l(\beta)$, then we use the **gradient descent** algorithm, which means each step we take the steep direction downhill.

$$\beta^{(t+1)} = \beta^{(t)} + \gamma_t \sum_{i=1}^{n} (y_i - p_i) X_i,$$

The algorithm learns from mistakes by trial and error. If a mistake is made such that p_i is very different from y_i, then β accumulates X_i if $y_i - p_i$ is positive, and $-X_i$ is $y_i - p_i$ is negative.

Python Code

```
def logistic(x, y, num_iteration=1000, learning_rate=1e-2):
    r, c = x.shape

    p = c + 1
    X = np.hstack((np.ones((r,1)), x))
    beta = 2*np.random.randn(p, 1)-1

    for i in xrange(num_iteration):
        pr = sigmoid(X.dot(beta))
        beta = beta + learning_rate* X.T.dot(y-pr)
        # px1 = pxn  nx1
    return beta

def sigmoid(x):
    return 1.0 / (1 + np.exp(-x))

n = 1000
p = 5
X = np.random.normal(0, 1, (n, p))
beta = np.ones((p, 1))

Y = np.random.uniform(0, 1, (n, 1)) < sigmoid(np.dot(X, beta)).reshape((n, 1))
logistic_beta = logistic(X, Y)
```

Iterated Reweighed Least Squares (IRLS)

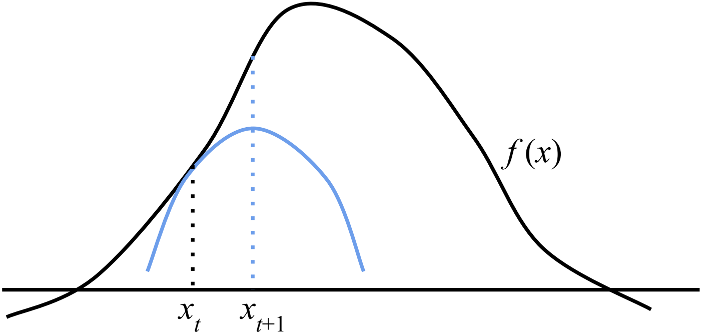

Figure 1: Second order approximation

Iterated Reweighed Least Squares (IRLS) Derivation

$l(\beta) = \log L(\beta) = \sum_{i=1}^{n} \left[y_i X_i^\top \beta - \log(1 + \exp X_i^\top \beta) \right].$

Let $L(\eta_i) = y_i X_i^\top \beta - \log(1 + \exp X_i^\top \beta)$

Perform Taylor expansion, $L(\eta_i) = L(\hat{\eta}_i) + L'(\eta_i)\Delta\eta_i + \frac{1}{2}L''(\eta_i)\Delta\eta_i^2$

Iterated Reweighed Least Squares (IRLS)

$$l(\beta) = -\sum_{i=1}^{n} \hat{w}_i(\hat{y}_i - x_i{}^T \Delta\beta)^2.$$

Where $w_i = p_i(1-p_i)$, $\hat{y}_i = \frac{\hat{e}_i}{\hat{w}_i}$

Recall linear regression:

$$\sum_{i=1}^{n}(y_i - x_i{}^T\beta)^2.$$

$$\begin{aligned}
\beta^{(t+1)} &= \beta_t + (\sum_{i=1}^{n} w_i X_i X_i^\top)^{-1}(\sum_{i=1}^{n} w_i X_i \hat{y}_i) \\
&= \left(\sum_{i=1}^{n} w_i X_i X_i^\top\right)^{-1}\left[\sum_{i=1}^{n} w_i X_i \left(X_i^\top \beta^{(t)} + \frac{y_i - p_i}{w_i}\right)\right].
\end{aligned}$$

Iterated Reweighed Least Squares (IRLS)

Consider the flow:

$$\beta^{(t)} \to \eta_i = X_i^\top \beta^{(t)} \to p_i = \sigma(\eta_i) \to w_i = p_i(1-p_i) \to \bar{y}_i = \eta_i + \frac{y_i - p_i}{w_i}$$

$$\to \tilde{X}_i = X_i\sqrt{w_i}, \tilde{y}_i = \bar{y}_i\sqrt{w_i} \to \beta^{(t+1)}.$$

we can rewrite the equation above as follows:

$$\begin{aligned}\beta^{(t+1)} &= \left(\sum_{i=1}^n w_i X_i X_i^\top\right)^{-1} \left(\sum_{i=1}^n w_i X_i \hat{y}_i\right) \\ &= \left(\sum_{i=1}^n \tilde{X}_i \tilde{X}_i^\top\right)^{-1} \left(\sum_{i=1}^n \tilde{X}_i \tilde{y}_i\right).\end{aligned}$$

Python Code

```
import numpy as np
from scipy import linalg
def mylogistic(_x, _y):
    x = _x.copy()
    y = _y.copy()
    r, c = x.shape
    beta = np.zeros((c, 1))
    epsilon = 1e-6
    while True:
        eta = np.dot(x, beta)
        pr = sigmoid(eta)
        w = pr * (1 - pr)
        z = eta + (y - pr) / w
        sw = np.sqrt(w)
        mw = np.repeat(sw, c, axis=1)

        x_work = mw * x
        y_work = sw * z

        beta_new, _, _, _ = np.linalg.lstsq(x_work, y_work)
        err = np.sum(np.abs(beta_new - beta))
        beta = beta_new
        if err < epsilon:
            break

    return beta
```