Linear Regression and Gradient Descent

Michael

School of Mathematics and Statistics, UCD

School of Economics, University of Nottingham

1 Linear Regression

Consider a multivariate linear relationship between a *dependent* variable, y_t, and a number of *explanatory* variable, x_{ti} observed across $t = 1, 2, \cdots, T$. Denoting the number of explanatory variables by k, the linear regression model is specified as:

$$y_t = \beta_1 x_{t1} + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t, \quad t = 1, \cdots T$$

Set the first term as the constant, then we typically have:

$$y_t = \beta_1 + \beta_2 x_{t2} + \cdots + \beta_k x_{tk} + u_t, \quad t = 1, \cdots T$$

The term u_t is an unobserved stochastic (random) error term, and is assumed to satisfy the following classical assumptions:

$$\begin{aligned} E(u_t) &= 0 \quad \forall t \\ V(u_t) &= \sigma^2 \quad \forall t \\ C(u_t, u_s) &= 0 \quad \forall s \neq t \end{aligned}$$

In the matrix form, we have

$$y = X\beta + u$$

with assumptions:

(i) $E(u) = 0$

(ii) $V(u) = E(uu') = \sigma^2 I_T$

(iii) X is non-stochastic with full column rank ($k < T$)

The estimation for the above model has a closed form:

$$\begin{aligned} \hat{\beta} &= (X'X)^{-1}X'y \\ \hat{\sigma}^2 &= \frac{\hat{u}'\hat{u}}{T-k} \\ V(\hat{\beta}) &= \hat{\sigma}^2(X'X)^{-1} \end{aligned}$$

However, only linear regression has the closed form of estimation of coefficients. For other regression models, we need apply the algorithm so called *gradient descent* to estimate the coefficients.

2 Gradient Descent

Since so many models in machine learning apply gradient descent to estimate parameters (coefficients), we would study this algorithm thoroughly.

2.1 Newton's Method

The Newton-Raphson method, or Newton Method, is a powerful technique for solving equations numerically[1]. Suppose r is the root of equation $f(x) = 0$, then based on the linear approximation, we have

$$0 = f(r) = f(x_0 + h) \approx f(x_0) + hf'(x_0)$$
$$0 \approx f(x_0) + hf'(x_0) \quad \Rightarrow \quad h \approx -\frac{f(x_0)}{f'(x_0)}$$

This could give us the following estimation formula:

$$x_{n+1} = x_n - \frac{f(x_n)}{f'(x_n)}$$

Example 2.2. Use Newton's method to find the estimation of $x^2 = 3$. The equation is $f(x) = x^2 - 3 = 0$, its derivative is $f'(x) = 2x$. We start to estimate solution from the starting value $x_0 = 1$.

x_n	$f(x_n)$	$f'(x_n)$	$f(x_n)/f'(x_n)$	x_{n+1}
1	-2	2	-1	2
2	1	4	1/4	7/4
		$\vdots$		

```
def Square_root(value):
    """A function that solve the square root equations,
    input is the value, and supposed to be postive"""
    if value < 0:
        raise ValueError('The input value has to be positive')
    else:
        x0 = 1  # set initial guess to be 1
        while abs(x0**2 - value) >= 0.0001:
            fxn = x0**2 - value
            fxn_der = 2*x0
            x1 = x0 - fxn/fxn_der
            x0 = x1
    return x0


# Test function
Square_root(-3)
a3 = Square_root(3)
print(a3)    # # 1.7320508100147276
print(math.sqrt(3))   # 1.7320508075688772
# You can change the convergence rate from 0.0001 to 0.001
```

[1]For more detailed explanations, pease read this: Click.

A general function for implementing Newton's method was given as follow[2].

```
# General implemenation of Newton Method with lambda
def newton(f, Df, x0, epsilon, max_iter):
    '''Approximate solution of f(x)=0 by Newton's method.

    Parameters
    ----------
    f : function
        Function for which are searching for a solution f(x) = 0
    Df : function
        Derivative of f(x)
    x0 : number
        Initial guess for a solution f(x) = 0
    epsilon: number
        Stopping criteria is abs(f(x)) < epsilon
    max_iter : integer
        Maximum number of iteration of Newton's method

    Returns
    -------
    xn: number
        Implement Newton's method: compute the linear approximation
        of f(x) at xn and find x intercept by the formula
            x = xn - f(xn)/Df(xn)
        Continue until abs(f(xn)) < epsilon and return xn.
        If Df(xn) == 0, return None. If the number of iterations
        exceeds max_iter, then return None.'''

    xn = x0
    for n in range(0, max_iter):
        fxn = f(xn)
        if abs(fxn) < epsilon:
            print('Found solution after', n, 'iterations.')
            return xn
        Dfxn = Df(xn)
        if Dfxn == 0:
            print('Zero derivative. No solution found.')
            return None
        xn = xn - fxn/Dfxn
    print('Exceeded maximum iterations. No solution found.')
    return None


# Test function
def p(x):
    return x**3 - x**2 - 1


def Dp(x):
    return 3*x**2 - 2*x


approx = newton(p, Dp, 1, 1e-10, 10)
print(approx)
# Found solution after 6 iterations.
```

[2]Referece: https://www.math.ubc.ca/~pwalls/math-python/roots-optimization/newton/

```
# 1.4655712318767877


# divergent case
def L(x):
    return x**(1/3)


def Dl(x):
    return (1/3)*x**(-2/3)


approx = newton(L, Dl, 0.1, 1e-2, 100)
# Newton's method diverges in certain cases. For example,
# if the tangent line at the root is vertical
```

2.3 Loss Function and Algorithm

For now, we define the **loss function** (cost function):

$$J(\theta) = \frac{1}{2}\sum_{i=1}^{m}[h_\theta(x^i) - y^i]^2$$

We want to choose θ so as to minimize $J(\theta)$. Let's consider the **gradient descent** algorithm, which starts with some initial θ, and repeatedly performs the update:

$$\theta_{j+1} = \theta_j - \alpha\frac{\alpha}{\partial\theta_j}J(\theta) \tag{2.1}$$

We know, when the loss function $J(\theta)$ reached to its minimum, then its derivative in terms of θ is close to zero, in which our parameters would converge at the same time. In equation (2.1), α is called the **learning rate**. Equation (2.1) can be summarized as: *when the loss is minimized, the parameters would converge to a certain set of numbers.*

When we are training more than one example, we use the following algorithm:

$$\begin{aligned}&\text{Repeat until convergence } \{\\ &\qquad \theta_{j+1} = \theta_j + \alpha\sum_{i=1}^{m}[y^i - h_\theta(x^i)]x_j^i \quad \text{(for every } j)\\ &\}\end{aligned} \tag{2.2}$$

This is called **batch gradient descent**.

2.4 Feature Scaling

When we have more than one independent variables, the feature value of X in equation (2.2) will affect the step size of gradient descent. Hence, we need scale our input before running the algorithm to solve the regression model[3].

Based on the different situations, you should:

[3]Reference:https://www.analyticsvidhya.com/blog/2020/04/feature-scaling-machine-learning-normalization-standardization/

- Normalization is good to use when you know that the distribution of your data does not follow a Gaussian distribution. This can be useful in algorithms that do not assume any distribution of the data like K-Nearest Neighbors and Neural Networks.
- Standardization, on the other hand, can be helpful in cases where the data follows a Gaussian distribution. However, this does not have to be necessarily true. Also, unlike normalization, standardization does not have a bounding range. So, even if you have outliers in your data, they will not be affected by standardization.

2.5 Shrinkage Methods

Due to the bias-variance tradeoff, we want to impose some penalties on the coefficients. When there are many correlated variables in a linear regression model, their coefficients can become poorly determined and exhibit high variance. There are two ways to do this:

- Ridge Regression
- Lasso

3 Stochastic Gradient Descent

In equation (2.2), there is a big 'sum', which means the basic gradient descent will use the whole dataset. This will slow down our calculations. Rather than using the whole batch of the dataset, we can select some data randomly and train our model. It turns out the loss of accuracy by using randomly selected sample is not that much.

Why stochastic gradient descent is so amazing? Please watch the following video:

- Open Lecture from Professor Suvrit Sra at MIT: Link.

Before you are trying to tackle the algorithm of stochastic gradient descent, I would like to put a few words about its intuition.

When we are doing basic gradient descent, we start to 'search' our parameters by giving an initial value we guessed. Then we trained our model until the estimated parameters converged. In this process, *we have to use the whole dataset to update our parameters in each iteration.* In the end, we stop our iteration once we got converged estimations. However, stochastic gradient descent avoids to use the whole dataset for each iteration. That's the key point!

```
def Stcst_gd(x, y, theta, samplesize, learning=0.1, iterations=100):
    '''
    Implementing the algorithm to do stochastic gradient descent
    Input
    ------
    x : dataset of dependent variables, an n by m matrix
    y : a vector of independent variable, an n by 1 vector
    theta : parameters (or estimations), m by 1 vetor
    samplesize : sample size of reshuffle, sample size <= x.shape[0]
    learning : learning rate, default value = 0.1
    interations : the maximum of interation

    Output
```

```
    -------
    theta: the convergent parameters
    cost_history : cost vector
    theta_history : trace of theta updating
    '''

    if samplesize > x.shape[0]:
        raise ValueError('Sample size must be less than population size')

    n = x.shape[0]
    m = x.shape[1]
    current_theta = theta
    alpha = learning
    cost_history = np.zeros([iterations, 1])
    theta_history = np.zeros([iterations, m])
    for it in range(iterations):
        randomIndex = random.sample(range(samplesize), samplesize)
        xtrain = x[randomIndex]
        ytrain = y[randomIndex]
        for i in range(samplesize):
            x_i = xtrain[i].reshape(1, -1)
            y_i = ytrain[i].reshape(-1, 1)
            fx = x_i @ current_theta
            update_theta = (current_theta
                            - alpha * 2
                            * x_i.transpose() @ (fx - y_i))
            current_theta = update_theta
        theta_history[it, :] = update_theta.transpose()
        cost_history[it, :] = square_loss(x, y, update_theta)

    return current_theta, cost_history, theta_history
```

You need practice more to understand gradient descent deeply!