MULTIPLE LINEAR REGRESSION WITH GRADIENT DESCENT

ANDREW YANG

1. Conventions

(i) Given an $m \times n$ matrix A, its entry in row i, column j will be denoted by $a_{i,j}$.

(ii) Vectors will be written in boldface.

(iii) Given a vector $\mathbf{v} \in \mathbb{R}^n$, its components will be denoted by $v_1, \dots, v_n$.

(iv) Python 3 `code` will be included at the end of every significant operation.

(v) It is assumed that the reader has access to the `numpy` module.

2. Matrix Representation of a Dataset

Let a given numeric dataset (such as a .csv file) with m training examples and n features be denoted by an $m \times (n+1)$ matrix A, so

$$A = \begin{pmatrix} a_{1,1} & \dots & a_{1,n+1} \\ \vdots & & \vdots \\ a_{m,1} & \dots & a_{m,n+1} \end{pmatrix}.$$

```
import numpy as np
...
dset = np.loadtxt('filename.txt', delimiter = ',')

#(Treat n as (n+1) for the time being)
dim = np.shape(dset)
m, n = dim[0], dim[1]
```

Assume that the rightmost column of A is a list containing the response variable in each training example, so let this column be denoted by

$$\mathbf{y} := \begin{pmatrix} y_1 \\ \vdots \\ y_m \end{pmatrix} = \begin{pmatrix} a_{1,n+1} \\ \vdots \\ a_{m,n+1} \end{pmatrix}.$$

Further, assume that in any given ith row of A, the entries $a_{i,1}, \ldots, a_{i,n}$ serve as the particular set of corresponding features to y_i. By assigning $\mathbf{x}^{(i)} := \left(x_1^{(i)}, \ldots, x_n^{(i)}\right) = (a_{i,1}, \ldots, a_{i,n})$ so that $\mathbf{x}^{(i)}$ is the row vector formed from the ith row of A, omitting the last entry $x_{n+1}^{(i)}$. The *feature matrix* X may thus be defined as

$$X = \begin{pmatrix} \text{———} \ \mathbf{x}^{(1)} \ \text{———} \\ \vdots \\ \text{———} \ \mathbf{x}^{(m)} \ \text{———} \end{pmatrix} = \begin{pmatrix} x_1^{(1)} & \ldots & x_n^{(1)} \\ \vdots & & \vdots \\ x_1^{(m)} & \ldots & x_n^{(m)} \end{pmatrix}.$$

Define the *hypothesis function* h as

$$h(\mathbf{x}) = \theta_0 + \theta_1 x_1 + \ldots \theta_n x_n$$

for some $\boldsymbol{\theta} = \begin{pmatrix} \theta_0 \\ \vdots \\ \theta_n \end{pmatrix} \in \mathbb{R}^{n+1}$.

Before implementing gradient descent, it is necessary to set $\boldsymbol{\theta}$ to a point.

To later express $h(\mathbf{x})$ as the inner product of $\mathbf{x}$ and $\boldsymbol{\theta}$, we can add $x_0 = 1$ to the beginning of each $\mathbf{x}^{(i)}$, so

$$\mathbf{x}^{(i)} := (x_0, x_1, \ldots, x_n) = (1, x_1, \ldots, x_n)$$
$$\implies X = \begin{pmatrix} 1 & x_1^{(1)} & \ldots & x_n^{(n)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{(m)} & \ldots & x_n^{(m)} \end{pmatrix} \in \mathbb{R}^{m \times (n+1)}.$$

Then, $h(\mathbf{x}^{(i)}) = \boldsymbol{\theta}^\top \mathbf{x}^{(i)}$.

```
X, y, th = [], [], []

for i in range(0, m):
    x = [1]
    for j in range(0, n - 1):
        x.append(dset[i][j])
```

```
    X.append(x)
    y.append(dset[i][n - 1])

#Set th to (1,...,1) by default
for i in range(0, n):
    th.append(1)

#h(x) will be implemented later
```

3. Gradient of the Loss Function

Given a training example $\mathbf{x}^{(i)}$, $h(\mathbf{x}^{(i)})$ should be the *expected value* of $y^{(i)}$, so the squared error between h and y is given by $\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2$.

Define the *loss function* $J(\boldsymbol{\theta})$ over every training example in our dataset as

$$J(\boldsymbol{\theta}) = \frac{1}{2m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2$$

where m is the number of training examples in our dataset, or alternatively, the length of $\mathbf{y}$.

The *gradient* of J is given by

$$\nabla J(\boldsymbol{\theta}) = \begin{pmatrix} \dfrac{\partial}{\partial\theta_0}J(\boldsymbol{\theta}) \\ \vdots \\ \dfrac{\partial}{\partial\theta_n}J(\boldsymbol{\theta}) \end{pmatrix}.$$

Note: For the purposes of this paper, we only need to use the loss function to compute its gradient.

To calculate the gradient of J, we first expand J as

$$\begin{aligned} J(\boldsymbol{\theta}) &= \frac{1}{2m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)^2 \\ &= \frac{1}{2m}\left(\boldsymbol{\theta}^\top\mathbf{x}^{(1)} - y^{(1)}\right)^2 + \cdots + \frac{1}{2m}\left(\boldsymbol{\theta}^\top\mathbf{x}^{(m)}y^{(m)}\right)^2 \\ &= \frac{1}{2m}\left(\theta_0 x_0^{(1)} + \cdots + \theta_n x_n^{(1)} - y^{(1)}\right)^2 + \dots \\ &\qquad + \frac{1}{2m}\left(\theta_0 x_0^{(m)} + \cdots + \theta_n x_n^{(m)} - y^{(m)}\right)^2. \end{aligned}$$

Then, the *partial derivative* of J with respect to some component θ_k of $\boldsymbol{\theta}$ is given by

$$\begin{aligned}\frac{\partial}{\partial\theta_k}J(\boldsymbol{\theta}) &= \frac{1}{m}\left(\theta_0 x_0^{(1)} + \dots + \theta_n x_n^{(1)} - y^{(1)}\right)\frac{\partial}{\partial\theta_k}\left(\theta_k x_k^{(1)}\right) + \dots \\ &\qquad + \frac{1}{m}\left(\theta_0 x_0^{(m)} + \dots + \theta_n x_n^{(m)} - y^{(m)}\right)\frac{\partial}{\partial\theta_k}\left(\theta_k x_k^{(m)}\right) \\ &= \frac{1}{m}\left(\boldsymbol{\theta}^\top \mathbf{x}^{(1)} - y^{(1)}\right)\left(x_k^{(1)}\right) + \dots \\ &\qquad + \frac{1}{m}\left(\boldsymbol{\theta}^\top \mathbf{x}^{(m)} - y^{(m)}\right)\left(x_k^{(m)}\right) \\ &= \frac{1}{m}\sum_{i=1}^{m}\left(\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)\cdot x_k^{(i)}\right).\end{aligned}$$

```
def h(x, y, th):
    return (np.dot(x, th) - y)

def deriv(X, y, th, k):
    e, m = 0, len(X)
    for i in range(0, m):
        e += (h(X[i], y[i], th) * X[i][k])
    return (e / len(X))
```

Note: Because $x_0^{(i)} = 1$*, the partial derivative of* J *with respect to* θ_0 *is simply*

$$\frac{1}{m}\sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right).$$

In full, the gradient of J is

$$\nabla J(\boldsymbol{\theta}) = \begin{pmatrix} \frac{\partial}{\partial\theta_0}J(\boldsymbol{\theta}) \\ \frac{\partial}{\partial\theta_1}J(\boldsymbol{\theta}) \\ \vdots \\ \frac{\partial}{\partial\theta_n}J(\boldsymbol{\theta}) \end{pmatrix} = \frac{1}{m}\begin{pmatrix} \sum_{i=1}^{m}\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right) \\ \sum_{i=1}^{m}\left(\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)\cdot x_1^{(i)}\right) \\ \vdots \\ \sum_{i=1}^{m}\left(\left(h(\mathbf{x}^{(i)}) - y^{(i)}\right)\cdot x_n^{(i)}\right) \end{pmatrix}.$$

```
def grad(X, y, th):
    g = []
    for k in range(0, len(th)):
        g.append(deriv(X, y, th, k))
    return g
```

4. Learning Rate and Updating $\boldsymbol{\theta}$

Define $\alpha \in \mathbb{R}$ as the *learning rate* and choose some value for it.

```
a = 0.01
```

In each iteration of gradient descent, we update $\boldsymbol{\theta}$ by the following rule:

$$\boldsymbol{\theta}_{\text{new}} := \boldsymbol{\theta}_{\text{old}} - \alpha \nabla(J(\boldsymbol{\theta}_{\text{old}})).$$

```
def mult(g, a):
    for i in range(0, len(g)):
        g[i] *= a
    return g

def renew(th, g):
    for i in range(0, len(th)):
        th[i] -= g[i]
    return th

g = mult(grad(X, y, th), a)
th = renew(th, g)
```

5. Execution

Define some precision value p and some maximum number of iterations allowed.

```
p = 0.1
max_it = 1000
```

Gradient descent should run until either $J(\boldsymbol{\theta}_{\text{old}}) - J(\boldsymbol{\theta}_{\text{new}}) < p$ or the maximum number of iterations has been reached.

```
p_curr = p + 1
it = 0
while p_curr > p and it < max_it:
    old = loss(X, y, th)
    th = renew(th, mult(grad(X, y, th), a))
    p_curr = old - loss(X, y, th)
    it += 1
```

6. Normalization

Before implementing gradient descent we can optionally *normalize* our data, which tends to let gradient descent converge faster. Recall the feature matrix X defined as

$$X = \begin{pmatrix} 1 & x_1^{(1)} & \dots & x_n^{(n)} \\ \vdots & \vdots & & \vdots \\ 1 & x_1^{(m)} & \dots & x_n^{(m)} \end{pmatrix}.$$

To normalize X, treat each column k except for the first one as a column vector $\mathbf{x}_k$, so

$$X = \begin{pmatrix} 1 & | & & | \\ \vdots & \mathbf{x}_1 & \dots & \mathbf{x}_n \\ 1 & | & & | \end{pmatrix}.$$

For every $\mathbf{x}_k$, define μ_k and σ_k respectively as the mean and standard deviation of the components of $\mathbf{x}_k$. Then, redefine each $\mathbf{x}_k$ as

$$\mathbf{x}_k := \frac{1}{\sigma_k}\left(\mathbf{x}_k - \begin{pmatrix} \mu_k \\ \vdots \\ \mu_k \end{pmatrix}\right).$$

```
import statistics as s

v_avg, v_dev = [None], [None]
for j in range(1, n):
    col = []
    for i in range(0, m):
        col.append(X[i][j])
    v_avg.append(s.mean(col))
    v_dev.append(s.stdev(col))
for i in range(0, m):
    for j in range(1, n):
        X[i][j] = (X[i][j] - v_avg[j]) / v_dev[j]
```