

Multiple linear regression

Multiple features

Multiple features (variables)

	Size in feet ² x_1	Number of bedrooms x_2	Number of floors x_3	Age of home in years x_4	Price (\$) in \$1000's
	2104	5	1	45	460
$i=2$	1416	3	2	40	232
	1534	3	2	30	315
	852	2	1	36	178

$x_j = j^{\text{th}}$ feature
 $n =$ number of features
 $\vec{x}^{(i)} =$ features of i^{th} training example
 $x_j^{(i)} =$ value of feature j in i^{th} training example

$j=1 \dots 4$
 $n=4$
 $\vec{x}^{(2)} = [1416 \ 3 \ 2 \ 40]$
 $x_3^{(2)} = 2$

- $x_1^{(4)} = 852$
- $x^{(1)} = [2014 \ 5 \ 1 \ 45] \rightarrow$ a row vector

Model:

Previously: $f_{w,b}(x) = wx + b$

example

$$f_{w,b}(x) = w_1 x_1 + w_2 x_2 + w_3 x_3 + w_4 x_4 + b$$

$$f_{w,b}(x) = 0.1 \underset{\substack{\uparrow \\ \text{size}}}{x_1} + 4 \underset{\substack{\uparrow \\ \text{\# bedrooms}}}{x_2} + 10 \underset{\substack{\uparrow \\ \text{\# floors}}}{x_3} + -2 \underset{\substack{\uparrow \\ \text{years}}}{x_4} + 80 \underset{\substack{\uparrow \\ \text{base price}}}{b}$$

$$f_{w,b}(x) = w_1 x_1 + w_2 x_2 + \dots + w_n x_n + b$$

- If we have n features, the model will look like this:

$$f_{x,b}(x) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

Multiple linear regression:

$$f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$$

$\vec{w} = [w_1 \ w_2 \ w_3 \ \dots \ w_n]$ parameters of the model
 b is a number
 $\vec{x} = [x_1 \ x_2 \ x_3 \ \dots \ x_n]$ vector

$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b = w_1x_1 + w_2x_2 + w_3x_3 + \dots + w_nx_n + b$$

dot product multiple linear regression

→ multiple linear regression (not multivariate regression)

- The name for this type of linear regression model with multiple input features is multiple linear regression.
- This is in contrast to univariate regression, which has just one feature. By the way, you might think this algorithm is called multivariate regression, but that term actually refers to something else that we won't be using here.

Vectorization

Parameters and features

$\vec{w} = [w_1 \ w_2 \ w_3]$ $n=3$

b is a number

$\vec{x} = [x_1 \ x_2 \ x_3]$

linear algebra: count from 1

NumPy

```
w = np.array([1.0, 2.5, -3.3])
b = 4
x = np.array([10, 20, 30])
```

code: count from 0

Without vectorization $n=100,000$

$f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + w_3x_3 + b$

```
f = w[0] * x[0] +
     w[1] * x[1] +
     w[2] * x[2] + b
```



Without vectorization

$f_{\vec{w},b}(\vec{x}) = \left(\sum_{j=1}^n w_j x_j \right) + b$ $\sum_{j=1}^n \rightarrow j=1 \dots n$
1, 2, 3

$\text{range}(0, n) \rightarrow j = 0 \dots n-1$

```
f = 0
for j in range(0, n):
    f = f + w[j] * x[j]
f = f + b
```



Vectorization

$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$

```
f = np.dot(w, x) + b
```



This NumPy dot function is a vectorized implementation of the dot product operation between two vectors and especially when n is large, this will run much faster than the two previous code examples.

- I want to emphasize that vectorization actually has two distinct benefits:
 - First, it makes code shorter, is now just one line of code.
 - Second, it also results in your code running much faster than either of the two previous implementations that did not use vectorization. The reason that the vectorized implementation is much faster is behind the scenes. The NumPy dot function is able to **use parallel hardware** in your computer and this is true whether you're running this on a normal computer, that is on a normal computer CPU or if you are using a GPU, a graphics processor unit, that's often used to accelerate machine learning jobs. The ability of the NumPy dot function to use parallel hardware makes it much more efficient than the for loop or the sequential calculation that we saw previously.

Let's take a deeper look at how a vectorized implementation may work on your computer behind the scenes.

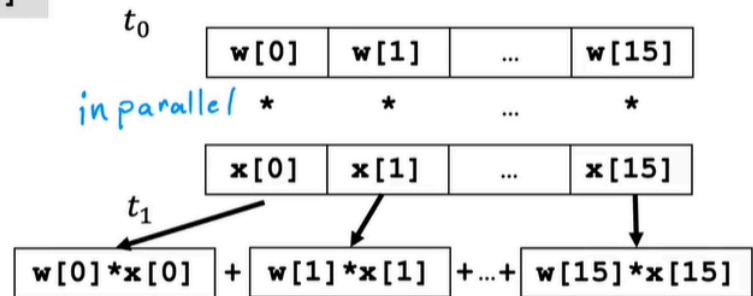
Without vectorization

```
for j in range(0,16):  
    f = f + w[j] * x[j]
```

t_0
 $f + w[0] * x[0]$
 t_1
 $f + w[1] * x[1]$
...
 t_{15}
 $f + w[15] * x[15]$

Vectorization

```
np.dot(w,x)
```



efficient → scale to large datasets

Let's look at this for loop. The for loop like this runs without vectorization.

- If j ranges from 0 to say 15, this piece of code performs operations one after another.
- On the first timestamp which I'm going to write as t_0 . It first operates on the values at index 0.
- At the next time-step, it calculates values corresponding to index 1 and so on until the 15th step, where it computes that.
- In other words, it calculates these computations one step at a time, one step after another.

In contrast, this function in NumPy is implemented in the computer hardware with vectorization.

- The computer can get all values of the vectors w and x , and in a single-step, it multiplies each pair of w and x with each other all at the same time in parallel.
- Then after that, the computer takes these 16 numbers and uses specialized hardware to add them altogether very efficiently, rather than needing to carry out distinct additions one after another to add up these 16 numbers.
- This means that codes with vectorization can perform calculations in much less time than codes without vectorization. This matters more when you're running algorithms on large data sets or trying to train large models, which is often the case with machine learning.

- That's why being able to vectorize implementations of learning algorithms, has been a key step to getting learning algorithms to run efficiently, and therefore scale well to large datasets that many modern machine learning algorithms now have to operate on.

Gradient descent $\vec{w} = (w_1 \ w_2 \ \dots \ w_{16})$ ~~b~~ parameters
 derivatives $\vec{d} = (d_1 \ d_2 \ \dots \ d_{16})$
`w = np.array([0.5, 1.3, ... 3.4])`
`d = np.array([0.3, 0.2, ... 0.4])`
 compute $w_j = w_j - \underbrace{0.1}_{\text{learning rate } \alpha} d_j$ for $j = 1 \dots 16$

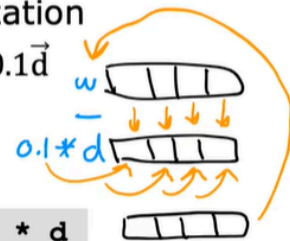
Without vectorization

$$\begin{aligned} w_1 &= w_1 - 0.1d_1 \\ w_2 &= w_2 - 0.1d_2 \\ &\vdots \\ w_{16} &= w_{16} - 0.1d_{16} \end{aligned}$$

```
for j in range(0,16):
    w[j] = w[j] - 0.1 * d[j]
```

With vectorization

$$\vec{w} = \vec{w} - 0.1\vec{d}$$



```
w = w - 0.1 * d
```

Gradient descent for multiple linear regression

"Normal equation" trong tiếng Việt được dịch là "**phương trình chuẩn**" hoặc "**phương trình bình phương tối thiểu chuẩn**", thường dùng trong bối cảnh học máy và thống kê.

Trong hồi quy tuyến tính (linear regression), *normal equation* là công thức giúp tìm nghiệm tối ưu (vector trọng số θ) mà không cần dùng đến thuật toán lặp như Gradient Descent. Công thức này là:

$$\theta = (X^T X)^{-1} X^T y$$

Ở đây:

- X : ma trận đặc trưng (features),
- y : vector đầu ra (labels),
- θ : vector trọng số cần tìm.

Dưới đây là bản soạn phù hợp để **bạn copy vào Notion** (giữ định dạng rõ ràng, dễ đọc và hỗ trợ công thức LaTeX):

Ví dụ: Dự đoán giá nhà theo diện tích

Giả sử bạn có tập dữ liệu:

Diện tích (x)	Giá nhà (y)
40	170
60	210
80	250

Tạo ma trận đặc trưng X và vector kết quả y :

$$X = \begin{bmatrix} 1 & 40 \\ 1 & 60 \\ 1 & 80 \end{bmatrix}, \quad y = \begin{bmatrix} 170 \\ 210 \\ 250 \end{bmatrix}$$

Áp dụng công thức Normal Equation:


$$\theta = (X^T X)^{-1} X^T y$$

👉 Kết quả tính được:

$$\theta = \begin{bmatrix} 90 \\ 2 \end{bmatrix}$$

Phương trình hồi quy:

$$\hat{y} = 90 + 2x$$

 Ý nghĩa:

- Khi diện tích = 0, mô hình vẫn dự đoán giá nhà là **90 triệu VND**.
- Mỗi mét vuông tăng → giá tăng thêm **2 triệu VND**.

Tại sao lại có cột **111** trong ma trận XX?

Trả lời ngắn gọn:

Cột toàn số 1 được thêm vào để mô hình hồi quy tuyến tính có thể học được hệ số chặn (bias / intercept).

Giải thích chi tiết:

Hồi quy tuyến tính tổng quát có dạng:

$$\hat{y} = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$$

Trong đó:

- θ_0 là **hệ số chặn** – giá trị dự đoán khi các đặc trưng $x_i = 0$.
- $\theta_1, \theta_2, \dots$ là hệ số của từng đặc trưng.

Để viết dưới dạng ma trận:

Ta thêm một cột **toàn số 1** vào ma trận đặc trưng XX:

$$X = \begin{bmatrix} 1 & x_1^{(1)} & x_2^{(1)} & \dots \\ 1 & x_1^{(2)} & x_2^{(2)} & \dots \\ \vdots & \vdots & \vdots & \ddots \end{bmatrix}$$

Vector trọng số:

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \end{bmatrix}$$

Khi đó:

$$\hat{y} = X\theta$$

! Nếu KHÔNG có cột 1 thì sao?

Mô hình chỉ học được:

$$\hat{y} = \theta_1 x_1 + \theta_2 x_2 + \dots$$

→ Tức là **không có hệ số chặn**, và mô hình **bị ép phải đi qua gốc tọa độ (0,0)** — điều này thường **gây sai số lớn** trong thực tế.