Andrew Ng Course Note 4

Shu Wang

May 17 2023

1 Multiple Linear Regression

1.1 Notation, Logistics

Example:

Size (feet2)	Number of bedrooms	Number of floors	Age of home (years)	Price ($1000)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

Notations:

x_j = the j^{th} feature

n = number of features

$\overrightarrow{x}^{(i)}$ = features of i^{th} training example

The above is called a row vector.

$\overrightarrow{x}_j^{(i)}$ = value of feature j in i^{th} training example

Model:

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

$f_{w,b}(x) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$

In general, $f_{w,b}(x) = w_1x_1 + w_2x_2 + ... + w_nx_n + b$

$\overrightarrow{w} = [w_1, w_2, w_3, ...w_n]$

This is a row vector.

b is a number.

$\overrightarrow{x} = [x_1, x_2, x_3, ...x_n]$

In a simpler way, $f_{\overrightarrow{w},b}(x) = \overrightarrow{w} \cdot \overrightarrow{x} + b$, where $\cdot$ is the dot product.

$f_{\overrightarrow{w},b}(x) = \overrightarrow{w} \cdot \overrightarrow{x} + b = w_1x_1 + w_2x_2 + ... + w_nx_n + b$

Parameters and features:

$\overrightarrow{w} = [w_1, w_2, w_3]$

In linear algebra, the index start from 1.

In python, the index start from 0.

In python, we can use np.array function in numpy package to write a vector:

```
import numpy as np


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

Without vectorization, you might write the code in this way:

```
import numpy as np


f = w[0] * x[0] + w[1] * x[2] + w[3] * x[3] + b
```

But this is inefficient and time consuming.

In math,you can write like this:

$f_{\vec{w},b}(x) = \sum_{j=1}^{n} w_j x_j + b$

Or write a loop:

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

Vectorization:

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

It runs much faster than ways without vectorization.

Numpy dot function uses parallel hardware in the computer. Makes it more efficient than for loop.

Without vectorization, for loop calculates one step at a time. In contrast, vectorization computes all numbers at the same time.

Gradient Descent:

$\vec{w} = [w_1, w_2, w_3, ...w_16]$

$\vec{d} = [d_1, d_2, d_3, ...d_16]$

compute $w_j = w_j - 0.1d_j$

Without Vectorization:

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

With vectorization:

w = w − 0.1 ∗ d

1.2 Gradient Descent for Multiple Linear Regression

Title	Previous Notation	Vector Notation
Parameters	$w_1, w_2, ...w_n$	$\overrightarrow{w} = [w_1, w_2, w_3, ...w_n]$
Model	$f_{\overrightarrow{w},b}(x) = w_1x_1 + w_2x_2 + ... + w_nx_n + b$	$f_{\overrightarrow{w},b}(x) = \overrightarrow{w} \cdot \overrightarrow{x} + b$
Cost function	$J(w_1, w_2, ..w_n, b)$	$J(\overrightarrow{w}, b)$
Gradient descent	Repeat $w_j = w_j - \alpha * \frac{\partial}{\partial w_j} J(w_1, w_2, ..w_n, b)$, $b = b - \alpha * \frac{\partial}{\partial b} J(w_1, w_2, ..w_n, b)$	Repeat $w_j = w_j - \alpha * \frac{\partial}{\partial w_j} J(\overrightarrow{w}, b)$, $b = b - \alpha * \frac{\partial}{\partial b} J(\overrightarrow{w}, b)$

Recall that in simple linear regression, we have:

$w = w - \alpha * \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) * x^{(i)}$

$b = b - \alpha * \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})$

Now with n feature, we have:

$w_1 = w_1 - \alpha * \frac{1}{m} \sum_{i=1}^{m} (f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}) * x_1^{(i)}$

The formula is the derivative: $\frac{\partial}{\partial w_1} J(\overrightarrow{w}, b)$

$w_n = w_n - \alpha * \frac{1}{m} \sum_{i=1}^{m} (f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)}) * x_n^{(i)}$

$b = b - \alpha * \frac{1}{m} \sum_{i=1}^{m} (f_{\overrightarrow{w},b}(\overrightarrow{x}^{(i)}) - y^{(i)})$

Simultaneously update w_j for $j = 1, 2, ...n$ and b

$x_1^{(i)}$ stands for all values in feature 1

An alternative to gradient descent: Normal equation.

Only for linear regression;

Solve for w, b without iterations.

Disadvantages:

Not able to generalize to other learning algorithms;

Quite slow when number of features is large ($> 10,000$)

2 Feature Scaling

Take our previous house price prediction example. We have x_1 the size(in squared feet) and x_2 for number of bedrooms.

Take an observation: $x_1 = 2000, x_2 = 5$

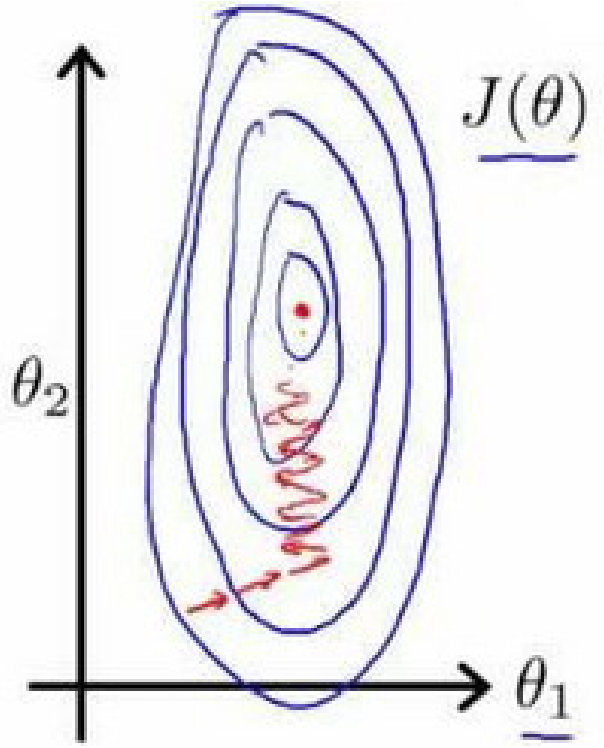
Option 1: Size of parameters w_1, w_2 equals $50, 0.1$ respectively.

Price = 50 * 2000 + 0.1*5 + 50

Option 2: Size of parameters w_1, w_2 equals $0.1, 50$ respectively.

Price = 0.1* 2000k + 50*5 + 50

When the possible range of values of a feature is large, it's more likely that a good model will choose a relatively small parameter value, like 0.01, vise versa.



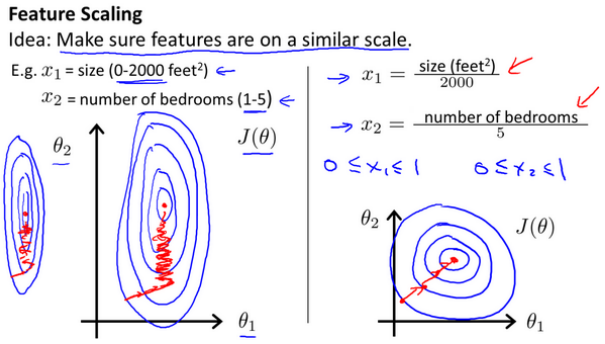
This is the contour plot where the horizontal axis has a much narrower range, between 0 and 1 whereas the vertical axis takes on much larger values, 10 and 1000. The contour forms an ellipse– short on x-axis, long on the other.

A very small change to w_1 will have a large impact on J.

In contrast, it takes a much larger change in w_2 to change the cost function.

The reason is that, the contours are so tall and skinny, gradient descent may end up bouncing back and force for a long time before it find its way to the global minimum.

Under this situation, we might want to scale the features s.t the range of x_1 and x_2 are the same.



When you have different features that take on very different values, it can cause gradient descent to run slowly. Rescaling the different features so they all take on comparable range of values can speed up gradient descent significantly.

How to implement feature scaling?

if $300 \leq x_1 \leq 2000$:

divide x_1 by the maximum: 2000

$x_{1,scaled} = \frac{x_1}{2000}$

Hence the new range is now : $0.15 \leq x_1 \leq 1$

Second way: Mean Normalization

Let μ_1 denote the mean of x_1.

Let σ_1 denote the standard deviation of x_1.

$x_{1,scaled} = \frac{x_1 - \mu_1}{2000 - 300}$

Another way: Z-score normalization

$x_{1,scaled} = \frac{x_1 - \mu_1}{\sigma_1}$

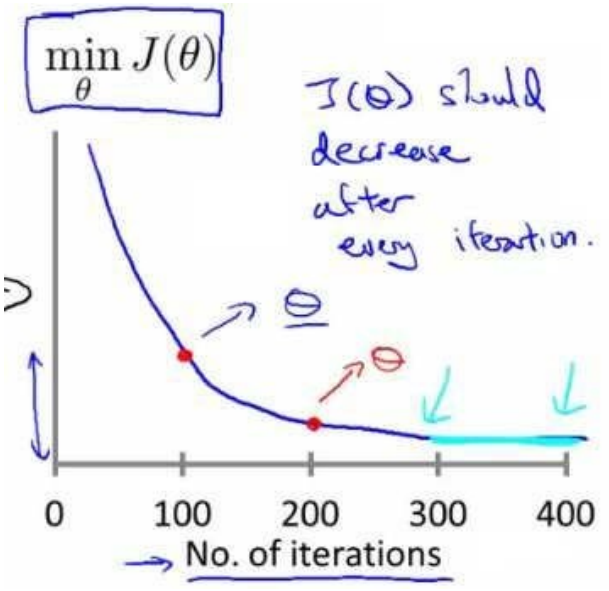
If you aim for about $-1 \leq x_j \leq 1$:

acceptable range from $-3 \leq x_1 \leq 3$, or $-0.3 \leq x_1 \leq 0.3$

You may want to rescale the variables that are too small or too large.

3 Convergence of Gradient Descent

Objective : $\underset{\vec{w},b}{\text{minimize}}\, J(\vec{w}, b)$



This curve is the learning curve.

If J increases after one iteration, then either α is chosen poorly (too large) or there exists bug in code.

By the time you reach a number of iterations(300 maybe), the cost J is leveling off. By 400 iterations, the curve has flattened.

This means that gradient descent has converged, because it's no longer decreasing.

It's difficult to know in advance how many iterations it takes to converge.

Automatic convergence test:

let $\epsilon = 10^{-3}$.

If $J(\vec{w}, b)$ decreases by $\leq \epsilon$ after one iteration, declare convergence.

We have found parameters $(\vec{w}, b)$ that is close to golbal minimum.

Choosing the right threshold for ϵ is quite difficult, would prefer to see the graph.

3.1 Choosing the right learning rate

If the cost goes up and down(oscillate), it means that learning rate is too large or code is not running properly.

To fix this, use a smaller learning rate.

Sometimes the cost continuously increases. Remember to use:

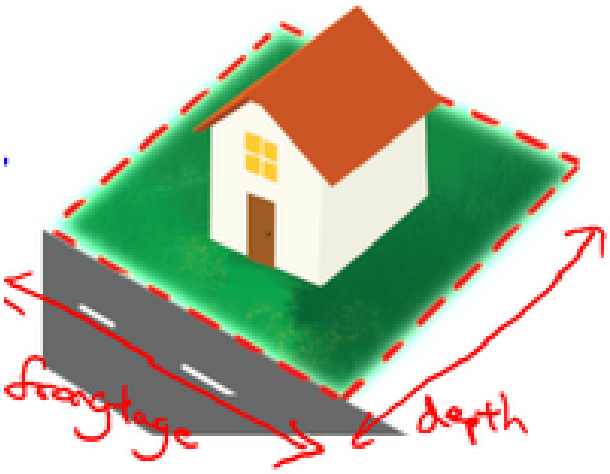
$w_1 = w_1 - \alpha * d_1$

Set α to be a very small number. Note that this should be treated as a way of debugging(i.e, to see whether there's bug in code) instead of actually implementing gradient descent. If the learning rate is too small, it takes more time and steps for the algorithm to converge. Start from $\alpha = 0.001, 0.01, ...$

Plot the cost function as a function of iterations, compare the graph and choose an α that decreases the learning rate rapidly and consistently.

Each value of α of roughly three times bigger.

3.2 Feature Engineering



For two features x_1 the frontage and x_2 the depth, you might built the model:

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

Area of the land: x_1x_2 is more predictive of the price than x_1 and x_2 as separate features.

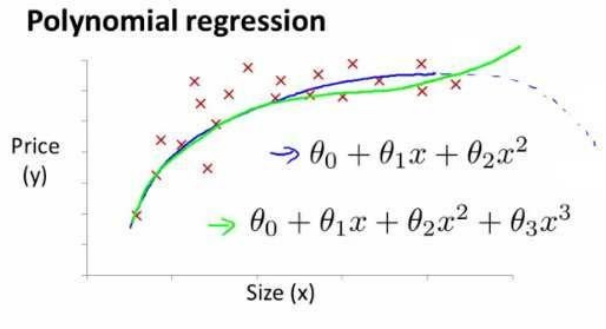
$x_3 = x_1x_2$. Update the model:

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

Feature Engineering:

Using intuition to design new features, by transforming or combining original features.

3.3 Polynomial Regression



Fit a quadratic function, x and x^2 included.

Or a cubic function:

$f_{\vec{w},b}(x) = w_1 x_1 + w_2 x^2 + w_3 x^3 + b$

Feature scaling becomes increasingly important.

The scale for the quadratic/cubic term would increase accordingly.

Or a square root:

$f_{\vec{w},b}(x) = w_1 x_1 + w_2 \sqrt{x} + b$

This would make the graph less steep as x increases, but will never decrease as quadratic terms.