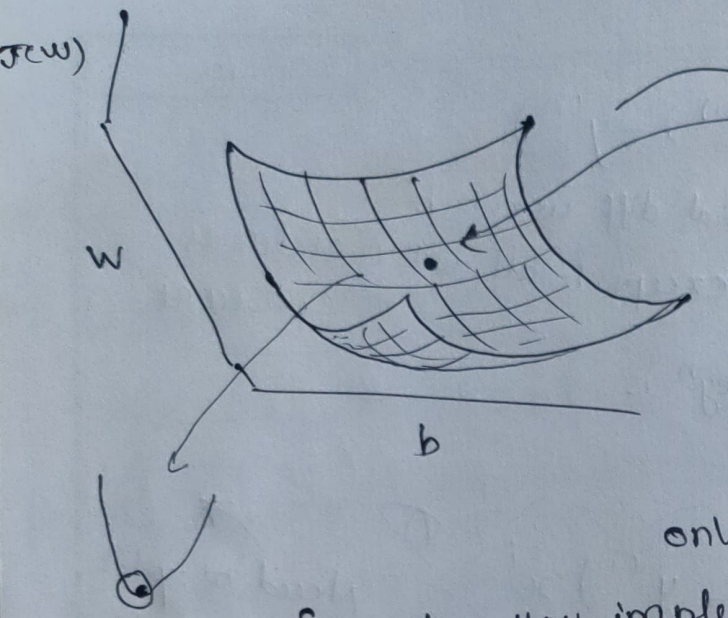
If we go for squared error cost



for this we have only one Global minima. ☆☆

We don't have local Minimas here.

(Technically, this cost function is convex function)

a bowl shaped function and it don't have local minima & it have only one global minimum

So, when you implement gradient descent on convex function, one nice property is that so as long as your learning rate is choosen appropriately, it will always converge to global minimum ☆. (☆)

Batch Gradient descent

Batch: Each step of gradient descent uses all training examples

	x size in feet^2	y price in $1000's
1	2104	400
2	1592	232
3	⋮	⋮
⋮	⋮	⋮
47	3210	870

$m = 47$

$$\sum_{i=1}^{47} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2$$

Week 2 :

MULTIPLE FEATURES : (Variables)

Linear Regn with Multiple features :

In the prev ex) → one feature.

x	y
Size in feet^2	~~size of house~~ price in $1000's
2104	400
1416	232
~~852~~	

Here we use only one feature to predict price in 1000$ So it is Linear Regn with one feature/ Variable

Multiple feature means here we use multiple features to predict the price of the house

Size in feet (X1)	No. of bedrooms (X2)	No. of floors (X3)	Age of home in year (X4)	Price ($) in $1000's (y)
~~2104~~	5	1	45	460
1416	3	2	40	232
		2	30	315

Features / independent feature

Label / o/p / Dependent feature

X_j = j^{th} feature $(X_1, X_2, X_3 \ldots)$ $j = 1$ to 4

n = no. of features ($n=4$ we have 4 features & 1 o/p)

$\vec{X}^{(i)}$ = features of i^{th} example

Vector

Ex: $\vec{X}^{(2)}$ = [1416 3 2 40] → row vector

$i=2$ i.e 2nd row.

$X_j^{(i)}$ = value of feature j in i^{th} example [Ex: $X_3^{(2)} \Rightarrow 2$]

In 3rd feature, 2nd element

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

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

$f_{w,b}(x) = 0.1x_1 + 4x_2 + 10x_3 + (-2)x_4 + 80$

x_1 ↑ size, x_2 ↑ Bedroom, x_3 ↑ floors, x_4 ↑ years, 80 ↑ Base price

Let assume price of house is 80000$

By every x square foot, the price will increase by 10%. ie 0.1
i.e $0.1 \times 1000 = 100$$

without Bedroom, floor, year, size

for each additional bathroom, the price of house ↑'es by $4 \times 1000 = 4000$$

for each additional floor, price ↑'es by $10 \times 1000 = 10000$$

for each every year, the price of house ↓'es by $-2 \times 1000 = -2000$$

If we have 'n' features then our model will looks like

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

$\vec{w} = [w_1\ w_2\ w_3\ \dots\ w_n]$ → 'w' vector } parameters of model

b is a number

vectors ←

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

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

↑ dot product of two vectors

THIS IS MULTIPLE LINEAR REGRESSION

To Implement Multiple Linear Regn we need to know neat trick called (Vectorization.)

↓ Used for other algo as well.

Vectorization part 1:

When we use Vectorization, the code will be shorter & efficient to run.

parameters and features

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

b is a Number

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

In Linear algebra : count starts from 1

w[0] w[1] w[2]

w = np.array([1.0, 2.5, -3.3]) } In Python index starts from '0'.

b = 4

x = np.array([10, 20, 30])

Without Vectorization:

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

n=3 → No. of features

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

But if n = 100 then it is hard to write code so, here we need to use Vectorization to run the code efficiently

if we write for 100 features then it is inefficient to run & Bad to compute

$$f_{\vec{w},b}(\vec{x}) = \left(\sum_{j=1}^{n} w_j x_j\right) + b$$ ---> Without Vectorization

```
f = 0
for j in range(0, n):        ---> it is better than but it is also inefficient
    f = f + w[j] * x[j]
f = f + b
```

Using Vectorization:

$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ — dot prod → To know about dot product Refer the vector calculus book once

**
```
f = np.dot(w,x) + b
```
---> Using Vectorization. (faster, efficient)

Vectorization part2:

without Vectorization the computer takes lot of time to execute stmts if n is high

But if we use " the code will run efficiently.

Without Vectorization

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

At t0
f + w[0] * x[0]

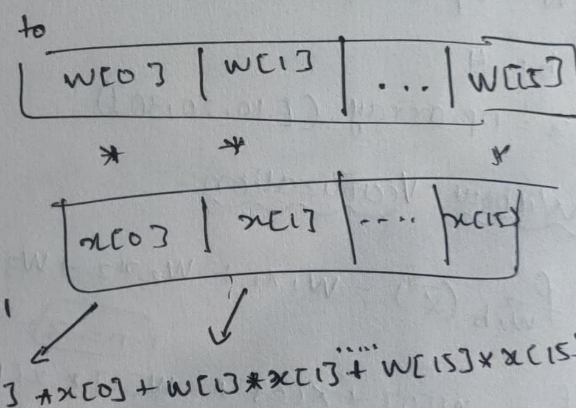
at t1
f + w[1] * x[1]

⋮

at t15
f + w[15] * x[15]

Vectorization

np.dot(w,x)



codes with Vectorization will do calculation in less time as compare to without Vectorization

This matters more when you running algorithms on large datasets or trying to train large models.

Gradient Descent

Suppose, $\vec{w} = (w_1\ w_2\ \ldots\ w_{16})$ (ignoring b for this case)

$\vec{d} = (d_1\ d_2\ \ldots\ d_{16})$ → derivative

```
w = np.array([0.5, 1.3, ... 3.4])
d = np.array([0.3, 0.2, ..., 0.4])
```

Compute $w_j = w_j - 0.1 d_j$ for $j = 1, 2, \ldots 16$

↓ Assume $\alpha = 0.1$

Without vectorization	With vectorization.
$w_1 = w_1 - 0.1 d_1$	$\vec{w} = \vec{w} - 0.1\vec{d}$
$w_2 = w_2 - 0.1 d_2$	`w = w - 0.1 * d`
⋮ ⋮	w [0.5 | 1.3 | ... | 3.4]
$w_{16} = w_{16} - 0.1 d_{16}$	(-)
Python:	0.1 * d [| | ... |]
for j in range(0,16):	0.1*0.2, 0.1*0.4, 0.1*0.3
w[j] = w[j] - 0.1 * d[j]	= [| | |]

GRADIENT DESCENT FOR MULTIPLE LINEAR REGRESSION:

	Previous Notation	Vector Notation **
parameters	$w_1, w_2, w_3, \dots, w_n$ b	$\vec{w} = [w_1 \dots w_n]$ → vector of length n b → number
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + \dots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ (↑ dot product)
cost function	$J(w_1, \dots, w_n, b)$	$J(\vec{w}, b)$

Gradient Descent

repeat (until convergence) {

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(w_1, \dots, w_n, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(w_1, \dots, w_n, b)$$

}

repeat (until convergence) {

$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$$

}

Gradient Descent

One feature (we see about this already)

repeat {

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

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

Simultaneously update w, b

}

** n features ($n \geq 2$)

here $\vec{x}^{(i)}, x_1^{(i)}$ as vectors

repeat {

for $j = 1, 2, \dots, n$ (loop)

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

$\hookrightarrow \frac{\partial}{\partial w_1} J(\vec{w}, b)$

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

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

Simultaneously update

w_j (for $j = 1, \dots, n$) and b

}

{THIS IS THE GRADIENT DESCENT}

↳ Next We will see "An alternative to gradient descent"

An alternative to Gradient Descent:

An alternate way to find w and b for Linear regression. This method is called Normal equation.

Normal equation: We know gradient descent is best algo for minimizing ~~w and b cost~~ function J to find w and b there is one other algorithm that works only for Linear regression and pretty much none of other algorithms you see in this specialization for solving w&b. and this other method doesnot need any iterative gradient descent algorithm. Called Normal equation method, it turns out to be possible to use an advanced linear algebra library to just solve for w and b all in one goal without iterations

Some Disadvantages of normal equation method are

→ First unlike gradient descent, this is not generalized to other learning algorithms such as logistic reg^n algorithm

→ Slow when the no. of features are large (> 10,000)

(Just know' some of machine learning libraries that implement linear regn by using Normal equation method).

(But Gradient descent is the recommended method for finding parameters w,b)

Gradient Descent in Practise:

Feature Scaling Part 1:-

Feature scaling is enable gradient descent to run much faster.
Let's start by taking a look at the relationship b/w the size of a feature that is how big are the numbers for that feature and the size of its associated parameter.
As a concrete example, let's predict the price of house using two features x_1 and size of the house and x_2 the no. of bedrooms.
Let say x_1 range is (300 - 2000 ft^2) & x_2 range is (0-5)

x_1 : size ($feet^2$)
range : 300 - 2000

x_2 : # bedrooms
range : 0 - 5

$\hat{price} = w_1x_1 + w_2x_2 + b$

estimated price, size, # of bedrooms

for this example x_1 takes very large range of values & x_2 takes on relatively small range of values.

(comp x_1 term takes range of values 300-2000 decimal).

x_1: 300, 320, ..., 2000, 1982, 1600

Now let's take an example, size of house = 2000 $feet^2$, No. of bedrooms = 5 & price = $500k

What is the reasonable w_1, w_2 values for this ex?

Let's look at one possible set of parameters

say $w_1 = 50, w_2 = 0.1, b = 50$

estimated price = $50 * 2000 + 0.1 * 5 + 50$

$= 100050.5k \approx \$100,050,500$ → it is very far away from $500k.

So, these w_1, w_2 are not a good values (prediction)

Let's Look for another possibility

$w_1 = 0.1$ (small), $w_2 = 50$, $b = 50$

$\hat{price} = 0.1 * 2000 + 50 * 5 + 50$

$\hat{price} = \$500k$ → predicted price matching with actual value } More reasonable

How does it relate to gradient descent?

Feature size and parameter size

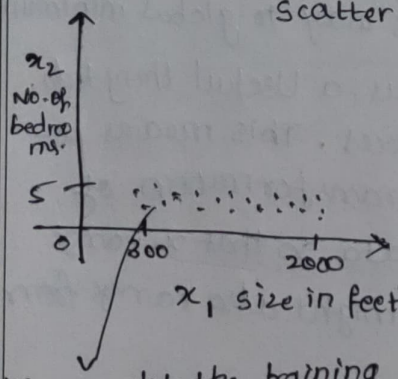
	size of feature x_j	size of parameter w_j
size in feet^2	← →	↔
# bedrooms	↔	↔

Features

Parameters

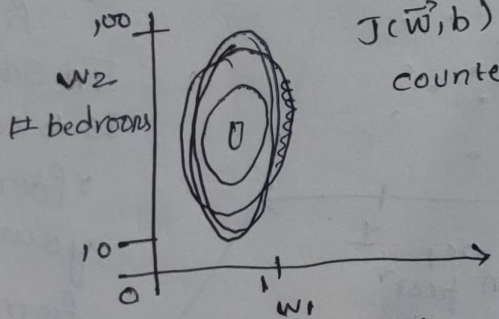
Let's take a look at scatter plot of features

Scatter plot



if we plot the training data, you notice that the horizontal axis is on a much larger scale or much larger range of values compared to vertical axis

Let's see how cost function might look in a counter plot



$J(\vec{w}, b)$ counter plot.

You might see a counter plot where the horizontal axis has a much narrower range, say b/w 0 & 1 Whereas Vertical axis takes on much larger values say 10 & 100.

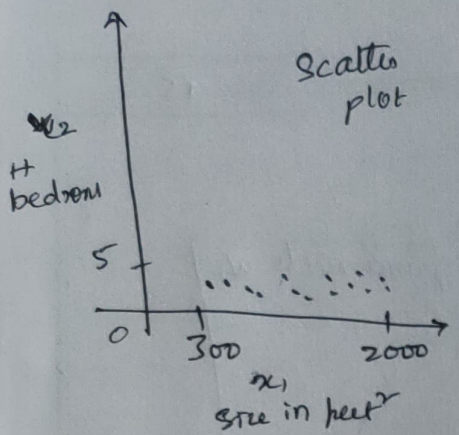
so contours form ovals or ellipses and they're short on one side (narrow) & longer on other (vertical)

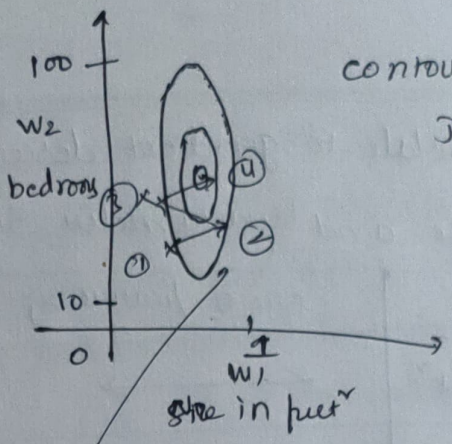
And this because very small change to w_1 can have very large impact on estimated price & cost f^n J also. Because w_1 tends to multiplied by very large number, the size & Square feet.

In contrast, it takes a much larger change w_2 in order to change the prediction much, And thus small changes to w_2, don't change cost f^n nearly as much.

So, where does this leave us?

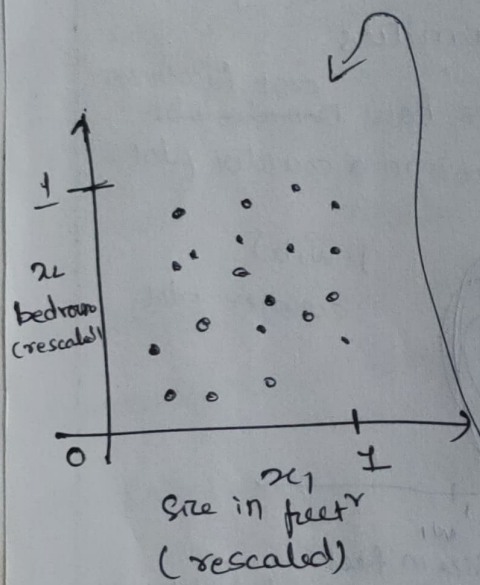
☆ (Next page)





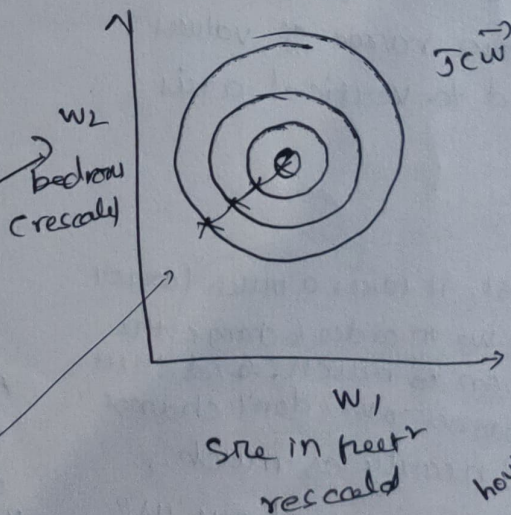
This is what we end up happening if you were to run gradient descent, if you were to use your training data as is. Because the contours are so tall and skinny may end up bouncing back & forth for a long time before it can finally find its way to global minimum.

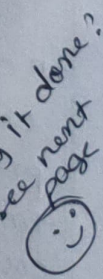
In situations like this, a useful thing to do is to scale the features. This means performing some transformation of your training data so that x_1 range from 0 to 1 & x_2 might also range from 0 to 1.



rescaling x_1 and x_2 both now taking comparable range of values to each other if you run gradient descent a cost fⁿ to find on this. rescaled x_1 & x_2 using this transformed data, then the contours will looks like circles & less tall & skinny.

And gradient descent can find a much more direct path to the global minimum.





RECAP: When you have different features that take on very different range of values, it can cause gradient descent to run slowly but rescaling diff features so they all take on comparable range of values. becau speed, upgrade & descent significally.

Feature Scaling part 2:

Implementation of feature scaling:

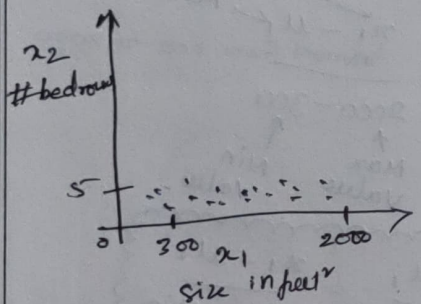
To take features that take on very different range of values and and skill them to have comparable ranges of values to each other.

How do you actually scale features?

if x_1 ranges from 300 to 2000

$$300 \leq x_1 \leq 2000$$

$$x_{1\,scaled} = \frac{x_1}{2000} \leftarrow \text{Max value}$$



Every value from x_1 feature will divide by Max value

Then scaled x features ranges from

$$\frac{300}{2000} \leq x_1 \leq \frac{2000}{2000}$$

$$0.15 \leq x_1 \leq 1$$

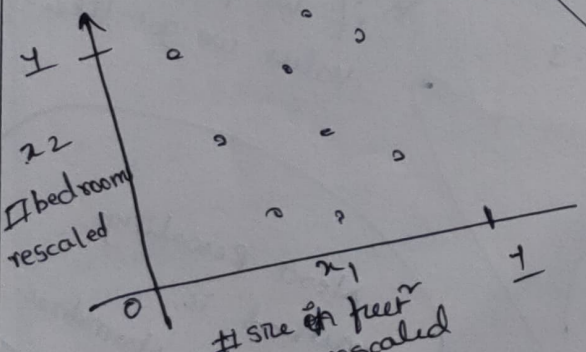
x_2 ranges from $0 \leq x_2 \leq 5$ (0 ← Min, 5 ← Max)

$$x_{2\,scaled} = \frac{x_2}{5} \leftarrow \text{Max value}$$

$$\frac{0}{5} \leq x_2 \leq \frac{5}{5}$$

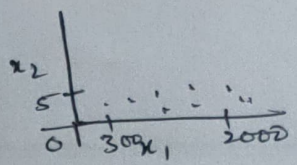
$$0 \leq x_2 \leq 1$$

if we plot these values in graph (x_1 vs x_2)



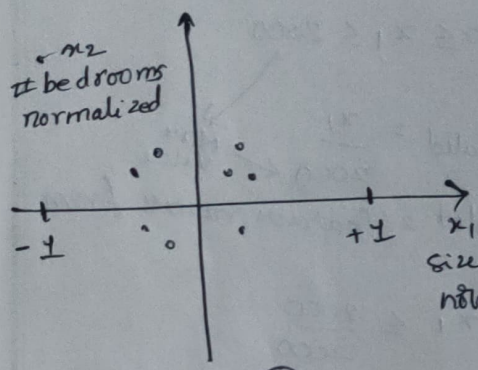
In Addition to dividing by Maximum value, You can also do what's called "mean normalization"

Mean Normalization:



First we started with original features and we rescale them so that both of them are centred around zero. whereas before (~~before~~ earlier scaling method) they only had values (>0) greater than zero But now they have both +ve and -ve values (usually b/w -1 to +1) ~~-1 to +1~~)

after mean normalization:



To calculate Mean Normalization of x_1,

1) Find Mean of x_1 (μ_1)

Ex: Mean = $\mu_1 = 600$

Then $x_1 = \dfrac{x_1 - \mu_1}{\text{ranges from 300 to 2000}}$ ← old value of feature; ← Mean of x_1

$2000 - 300$

↑ Max value ↑ Min value

x_1 (rescaled) $= \dfrac{x_1 - \mu_1}{2000 - 300} = \dfrac{x_1 - \mu_1}{\text{Max} - \text{Min}}$

$-0.18 \leq x_1 \leq 0.82$

$\dfrac{300 - 600}{2000 - 300} \leq x_1 \leq \dfrac{2000 - 600}{2000 - 300}$

Similarly,

for x_2, also, x_2 ranges: $0 \leq x \leq 5$

$x_2 = \dfrac{x_2 - \mu_2}{\text{Max} - \text{Min}}$ → each value of feature x_2

Let's say $\mu_2 = 2.3$

Then

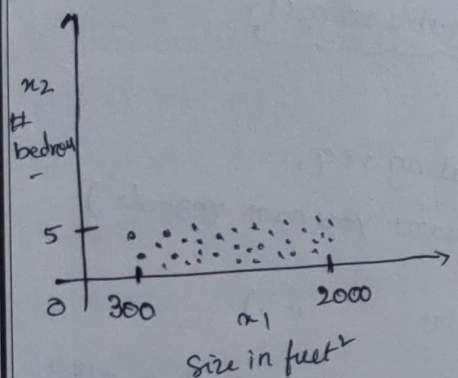
$\dfrac{0 - 2.3}{5 - 0} \leq x_2 \leq \dfrac{5 - 2.3}{5 - 0}$

$-0.46 \leq x_2 \leq 0.54$

if we plot these values we got like:

Next Rescaling method is Z-score Normalizati.

Z-Score Normalization :



$300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

To implement Z-score normalization we need to calculate standard deviation of each feature

$$Z = \frac{x - \mu}{\sigma}$$

For instance if x_1 feature has $\sigma_1 = 450$

& $\mu_1 = 600$

Then Z-score Normalization of x_1 is → each value of feature

$$x_1 = \frac{x_1 - \mu_1}{\sigma_1}$$

$$\frac{300 - 600}{450} \le x_1 \le \frac{2000 - 600}{450}$$

$$-0.666 \le x_1 \le 3.111$$

Illy, for x_2

$$x_2 = \frac{x_2 - \mu_2}{\sigma_2}$$ rescaled

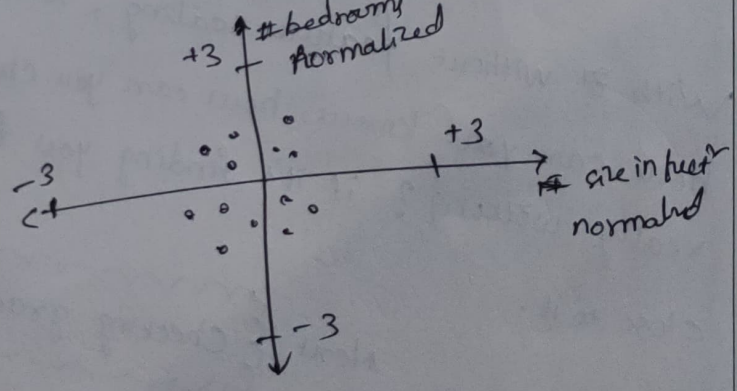
for instance std-dell of x_2 feature is $\sigma_2 = 1.4$

$\mu_2 = 2.3$

$$\frac{0 - 2.3}{1.4} \le x_2 \le \frac{5 - 2.3}{1.4}$$

$$-1.64 \le x_2 \le 1.928$$

if we plot x_1 & x_2



Next we will see about Thumb rule of Normalization

As a Thumb rule, when you perform feature scaling, you might want aim for getting the features to range from ~~-1 to +1~~

aim for about $-1 \le x_j \le 1$ for each feature x_j

$-3 \le x_j \le 3$

$-0.3 \le x_j \le 0.3$

{Acceptable ranges}

$0 \le x_j \le 3$ (Okay, no rescaling req, But if you want you can rescale)

$-2 \le x_2 \le 0.5$ (" " ")

$-100 \le x_3 \le 100$ (But here range is high from -100 to +100 i.e 200 types of value (ie -100 to 0, 0 to 100)) So Not acceptable, {Need to rescale}

(Too large)

Too large can

Too small can

$-0.001 \le x_4 \le +0.001$ (These values are too small) (Need to rescale)

(Too small)

$98.6 \le x_5 \le 105$ (values are around 100, i.e pretty large. compared to other scale features so it will cause gradient descent to run more slowly. So we need to rescale the feature to make fast the gradient descent)

With or without feature scaling, when you run gradient descent, how can you know, how can you check if gradient descent is really working? if it's finding you the global minum or something close to it.

Next {Checking gradient descent for Convergence}

Checking Gradient Descent for Convergence :-

When running gradient descent, how can you tell whether it is converging? That is whether it's helping you to find parameters close to the global minimum of cost function.

By learning to recognize what a well-running implementation of gradient descent looks like, we will also look how to choose better learning rate (α). Let's take a look.

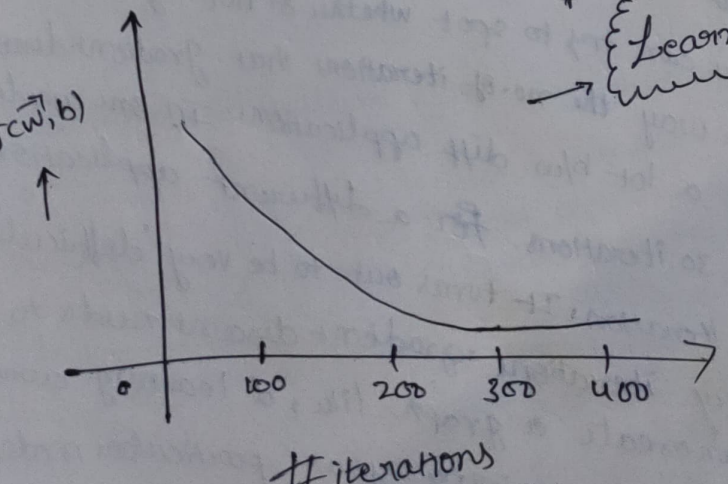
$$w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$$

$$b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$$

Recalling the objective of gradient descent : $\min_{\vec{w}, b} J(\vec{w}, b)$

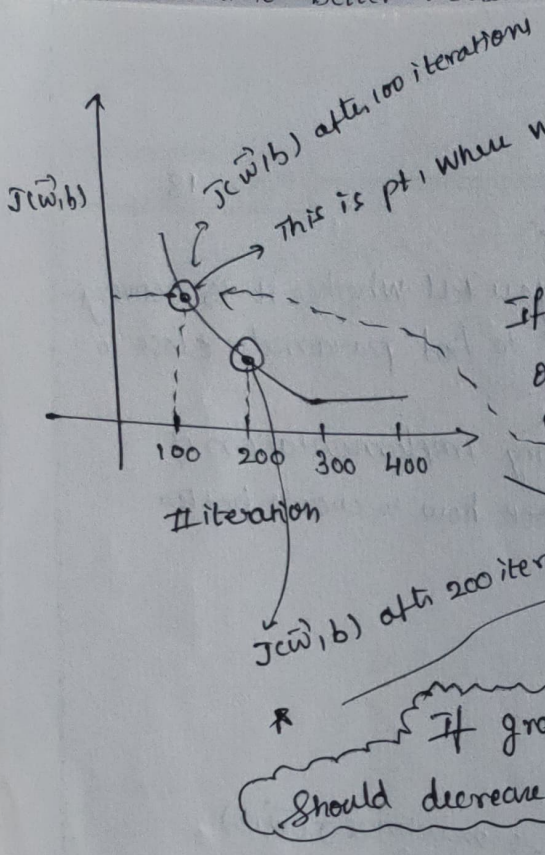
→ is finding parameters $\vec{w}, b$ that hopefully minimize cost function.

plot cost fⁿ J (which is calculated on training set) & I plot the J at each iteration. (each iteration means each simultaneous update of parameters w and b). In this plot the horizontal axis is no. of iterations of gradient descent you've run so far. then we get a curve like below & vertical axis is cost fⁿ J.



Learning Curve → cost fⁿ vs no. of iterations

There are some more types of learning curves in Machine learning. We will see later in the course.



meaning 100 simultaneous updates of parameters

we have some learned values for w and b

If you compute the cost J, w, b for those values of w and b the ones you got after 100 iterations we get this value of J.

From this graph we do see that cost J after each iteration of gradient descent.

how ~~graph~~ changes ~~according to No of~~

If gradient descent is working properly, then cost J should decrease after every single iteration

If J ever increases after one iteration that means either α is choosen poorly (and it usually means α is too large) or there could be a bug in the code.

Another useful thing that this part can tell you is that if you look at this curve by the time we reach 300 iterations also, the cost J is levelling off and no longer decreasing much. By 400 iterations, it looks like curve has flattened out. This means that gradient descent has more or less converged because the curve is no longer decreasing.

(tend to move toward lower point)

By looking into the learning curve, you can try to spot whether or not gradient descent is converging. By the way the no. of iterations that gradient descent take a convergence can vary a lot b/w diff applications. In one application, it may converge after 30 iterations. For a different application it could take 1,000 or 10,000 iterations. It turns out to be very difficult to tell in advance how many iterations gradient descent needs to converge, which is why we can create a graph like, a learning curve. Try to find out when you can start training your particular model.

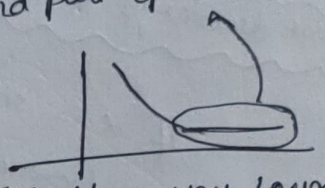
Another way to decide when your model is done training, is with an automatic convergence test.

Automatic convergence test :

Let 'ε' be 10^{-3} i.e 0.001 (a small number)

If $J(\vec{w}, b)$ decreases by $\leq \varepsilon$ in one iteration, declare "convergence"

i.e you're likely on flattend part of curve



Remember convergence, hopefully in the case that you found parameters w and b that are close to minimum ponible value of J. I usually find that choosing the right threshold 'ε' is difficult. I actually tend to look at graphs rather than rely on automatic convergence tests.

You've now seen what the learning curve should look like when gradient descent is running well.

CHOOSING LEARNING RATE:- (α) We need to choose best α.

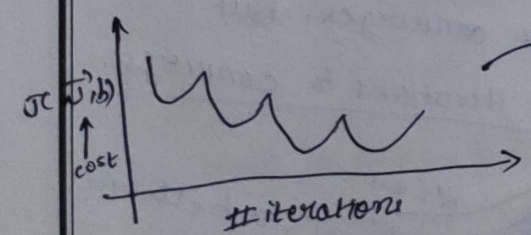
If α is small, it will run slowly (grad. desc) & if α is high (too large) it may not converge.

Let's take a look how can choose a good learning rate for your model?

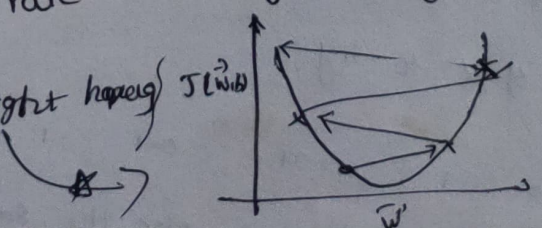
(10:17)

Identify the problem with gradient descent

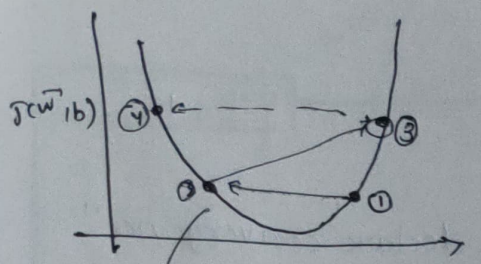
when we plot cost vs no. of iterations, we notice that cost sometimes up & sometimes down. you should take that clear sign that gradient descent is not working properly. This could mean that there a bug in code or sometimes it could mean that your learning rate is too large. (not converge)



So, here is illustration of what might happen



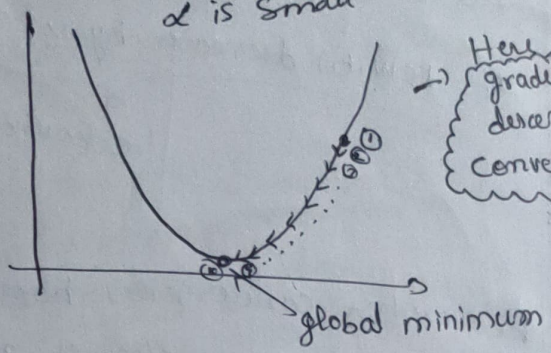
If α is too big



The cost will take increase instead of decreasing. To fix this so, we need to takes α value too small. to limit this. Then updates may starts smaller

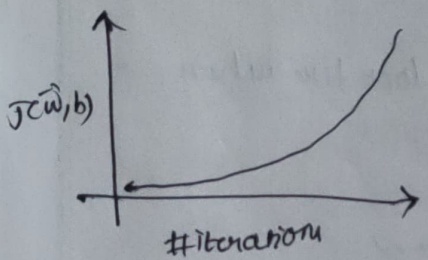
Here gradient descent is diverge

if α is small, the cost decrease until it reaches global minimum.



Here gradient descent is converge

Sometimes we may see that cost increases w.r.to iteration



→ This is also due to α is too large & sometimes due to bug in the code

So to fix this we need to take smaller learning rate(α)

→ like if we write code like

incorrect → $w_1 = w_1 + \alpha \frac{\partial}{\partial w_1} J(\vec{w}, b)$

Usually, here we need to put (-ve) sign but we place '+ve' there so this is called 'Bug in code'

correct → $w_1 = w_1 - \alpha \frac{\partial}{\partial w_1} J(\vec{w}, b)$

One debugging tip for correct implementation of gradient descent is

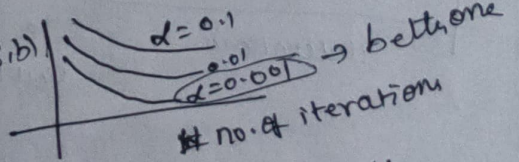
1) Using smaller 'α'
2) cost f^n should decrease on every iteration. (if we follow 1st one, 2nd one will automatically done)

Note: If α is too small, the gradient descent converges, But α is too small gradient descent takes a lot more iterations to converge.

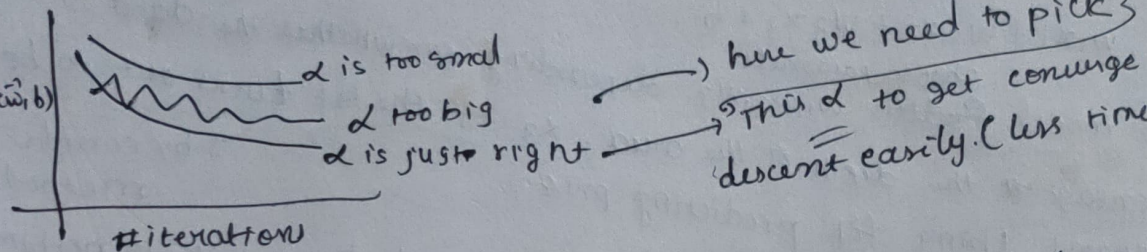
Values of α to try:

... 0.001 0.01 0.1 1 .....

Try range of values & pick the 'α' value which giving best (less time.



Suppose if we got curves like

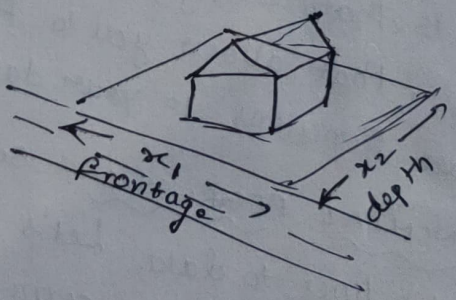


FEATURE ENGINEERING: The choice of features can have huge impact on your learning algorithm's performance. In fact, for many practical applications choosing or entering right features is a critical step to making the algorithm work well.

Let's see how you can choose or engineer the most appropriate features for the learning algorithm

Let's take example of predicting price of house.

Let's say we have two features, x_1 → width of the lot side of the plots of land that house is build on (In realestate, we called frontage of the lot) x_2 → (second feature) depth of the lot size (Assume the house built on rectangular plot)



By using two feature we built a Model to predict price is like.

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

w_1 → Frontage, w_2 → Depth

This Model is okay, but here's another option for how you might choose a different way to use these features in the model that could be more effective. You might notice that area of land calculated by frontage & width time the depth

$$\text{area} = \text{frontage} \times \text{depth}$$

We have a intution that area of land is more predictive of the price, than the frontage & depth as seperate features. So you might define a new feature, x_3 as $x_1 * x_2$, $x_3 \rightarrow$ area of plot of land

With the feature x_3 we have a model, f_w

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

choose parameters

so that now model ~~f~~ w_1, w_2, w_3 depending on whether the data shows that the frontage & the depth or the area.. x_3 of the lot turns out to be most important thing for predicting price

by correlation method.

We have created New feature for predicting price of house, it is nothing but "feature engineering". → Using intution to design new features by transforming or combining original features. to make easier for learning algorithm to make accurate predictions.

Depending on what insights you may have into the application, rather than just taking the features that you happen to have started off with sometimes by defining new features, you might be able to get a much better model ♡

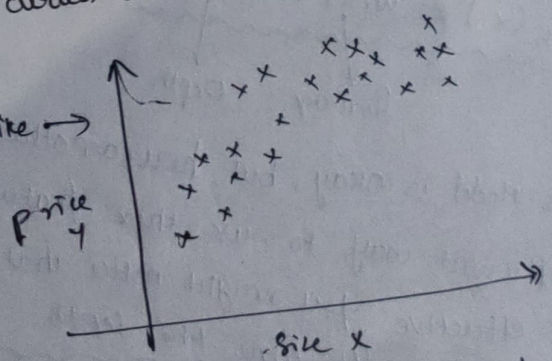
This is one flavour of feature engg, It turns out that this one flavor of feature engg, that allow you to fit not just straight lines, but curves, non-linear functions to your data. Let's see that

POLYNOMIAL REGRESSION: So far we have seen that, we are fitting straight lines to data. Let's take the ideas of multiple Linear Regression and feature engineering to come up with a new algorithm called "polynomial regression". which you let you fit curves with non-linear functions to your data

Let's say you have housing data like →

feature x → size in feet

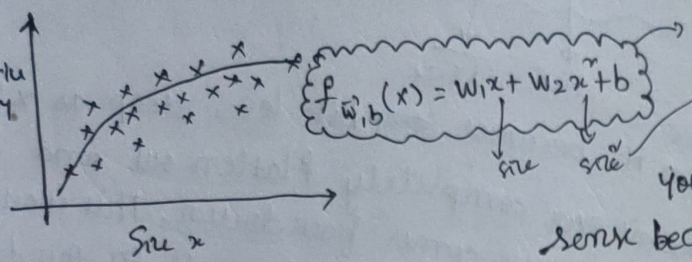
feature y → price



It looks like not a straight line fit this dataset very well. Maybe you want to fit a curve may be a quadratic function to the data like

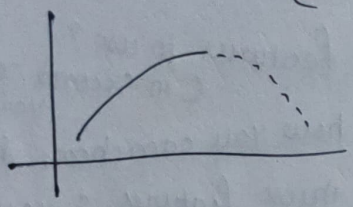
$$f_{\vec{w},b}(x) = w_1x + w_2x^2 + b$$

size, size²



May be this give better fit to data.

But then you may decide that your quadratic model doesn't really makes sense because a quadratic function, eventually comes back down. well we wouldn't really expect housing prices to go down when size ↑'es

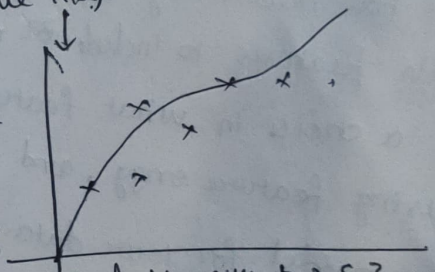


Big houses seems like they should usually cost more. Then you may choose cubic function where we now have not only x squared, but x cubed.

$$f_{\vec{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + b$$

size, size², size³

May be this Model produces this curve like,



Which is somewhat better fit to data because the size does eventually come back up as the size increases.

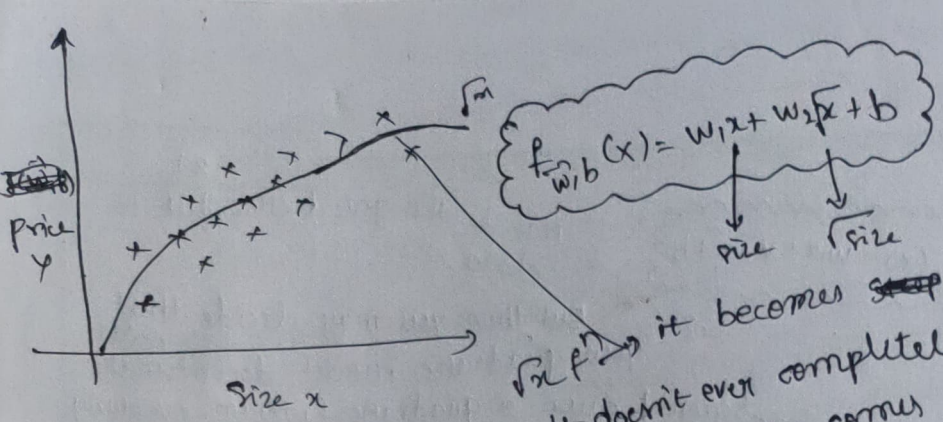
These are both examples of polynomial Regression. Because you took optional feature x & raised it power to 2 & 3...

I just want to point out one more thing, which is that if you create features that are these powers like square of original feature like this, then feature scaling becomes increasingly important.

if size of house (x) ranges $1 - 10^3$ then $x^2 \rightarrow (1)^2 - (10^3)^2 \Rightarrow 1 - 10^6 \leftarrow$ 1 millia

then $x^3 \rightarrow 1^3 - 10^9 \rightarrow$ 1 billion

These two features (x^2, x^3) take on very different ranges of values compared to the original feature x. If you are using gradient descent, its important to apply feature scaling to get your features into comparable range of values. Finally, just one last example of how you really have a wide range of choice of features to use. Another reasonable alternative to taking the size² & size³ is to say use square root of x then Model look like



$\sqrt{x}$ f^n → it becomes bit less steep as x ↑es but it doesn't ever completely flatten out and it certainly never ever comes back down. This would be another choice of features that might work well for this data set as well

You may ask yourself, how do I decide what features to use? (In second course (advance learning algo^m) you see how you can choose diff features and diff models that include & don't include these features. (in second course you can know that) & you have a process for measuring how well these diff models perform to help you to decide which features to include & not include. For now, you need to be aware that you have a choice in what features you have.

By using feature engg, and polynomial f^n, you can potentially get a much better model for your data.

LABS:

Computing cost: 1st Lab.

$$\text{cost} = J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

(squared diff b/w target value & prediction)

$f_{w,b}(x^{(i)}) = wx^{(i)} + b$ → it is the prediction ($\hat{y}$)

```
>> import numpy as np
>> from matplotlib.pyplot as plt
>> x_train = np.array([1.0, 2.0]) → size in feet²
   y_train = np.array([300.0, 500.0]) → price in $
```

compute cost method: (x_train, y_train, w & b value)

```
def compute_cost(x, y, w, b):
    m = x.shape[0] → no. of training ex's (len(x)) also works
    cost_sum = 0
    for i in range(m):
        f_wb = w * x[i] + b
```

```
for i in range(m):
        f_wb = w * x[i] + b  ⟶ test
        cost = (f_wb - y[i]) **2
        cost_sum = cost_sum + cost
total_cost = (1/(2*m)) * cost_sum
return total_cost
```

LAB 2: Implementing gradient descent.

```
def compute_gradient(x, y, w, b):   (x-train, y-train)
    m = x.shape[0]
    dj_dw = 0
    dj_db = 0
    for i in range(m):
        f_wb = w * x[i] + b
        dj_dw_i = (f_wb - y[i]) * x[i]
        dj_db_i = f_wb - y[i]
        dj_db += dj_db_i
        dj_dw += dj_dw_i
    dj_dw = dj_dw/m
    dj_db = dj_db/m
return dj_dw, dj_db
```

Refer page NO:10

(Just to know concept).