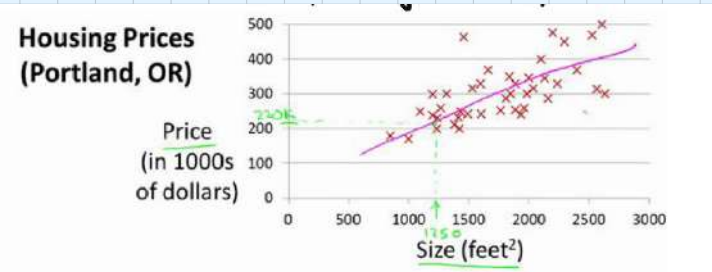
FIRST WEEK

Supervised Learning vs Unsupervised Learning

Supervised Learning model
- Regression model → predict number Infinity
- Classification model → small number of possible output

Linear Regression model



Regression model predict numbers

↳ or can use Data table

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
...	...

Got many Terminology 术语

Terminology in Linear Regression

- Training set : data used to train the model
-

Size in feet² (x)	Price ($) in 1000's (y)
2104	460
1416	232
1534	315
852	178
...	...

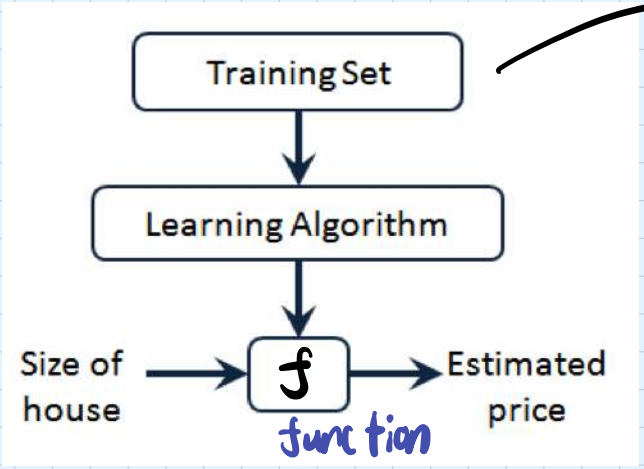
m=47

x: "input" variable feature
y: "output" variable feature
"target" variable
m: numbers of training examples

- $x^{(1)} = 2104$, $y^{(1)} = 460$, when $i = 1$
  i: i^{th} training example

$(x^{(1)}, y^{(1)}) = (2104, 460)$

↳ single training example



features
targets

x

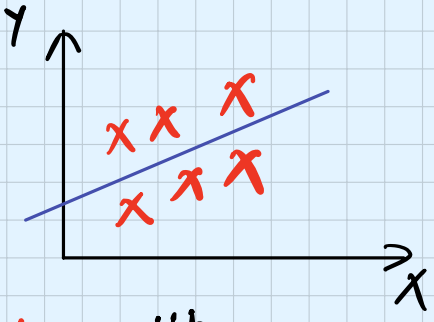
model

$\hat{y}$: "y-hat"
Prediction $\hat{y}$
(estimated y)

How to represent f ?

↳ $f_{w,b}(x) = wx + b$

simple : $f(x) = wx + b$



↳ This is Linear Regression with One variable

↳ also called Univariate linear Regression
(one)

Optional Lab

General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
Regression		
$\mathbf{x}$	Training Example feature values (in this lab - Size (1000 sqft))	`x_train`
$\mathbf{y}$	Training Example targets (in this lab Price (1000s of dollars)).	`y_train`
$x^{(i)}$, $y^{(i)}$	i_{th}Training Example	`x_i`, `y_i`
m	Number of training examples	`m`
w	parameter: weight,	`w`
b	parameter: bias	`b`
$f_{w,b}(x^{(i)})$	The result of the model evaluation at $x^{(i)}$ parameterized by w, b: $f_{w,b}(x^{(i)}) = wx^{(i)} + b$	`f_wb`

Cost Function

Training Set	features: Size in feet² (x)	targets: Price ($) in 1000's (y)
	2104	460
	1416	232
	1534	315
	852	178
	...	...

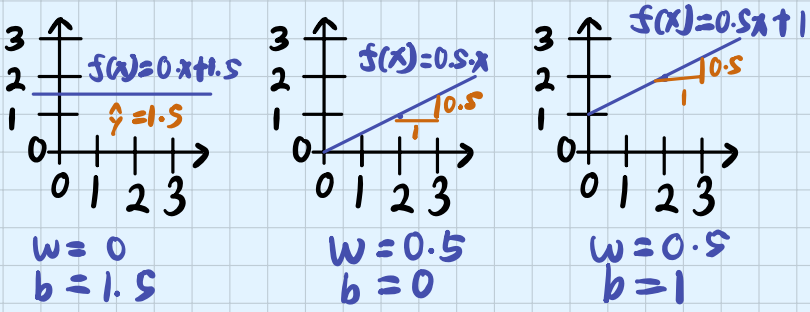
$m = 47$

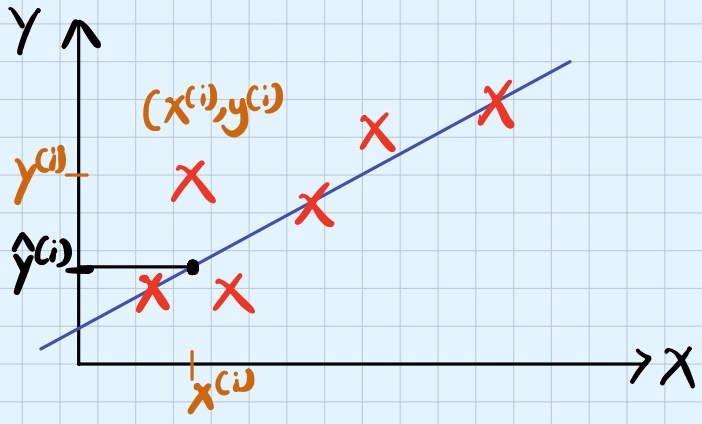
Model: $f_{w,b} = wx + b$

w, b : parameter / weights / coefficients

What w & b do?

↳ $f_{w,b}(x) = wx + b$





$\therefore f_{w,b}(x^{(i)}) = wx^{(i)} + b$

$\hat{y}^{(i)} = f_{w,b}(x^{(i)})$

$\therefore$ So w, b is for predict of $\hat{y}^{(i)}$ near the targets $y^{(i)}$

Find w, b: $\hat{y}^{(i)}$ is close to $y^{(i)}$ for all $(x^{(i)}, y^{(i)})$

Cost Function : Squared error cost function

↳ $J_{w,b} = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$

m for numbers of examples
2 for convenient

sum

error

We can write

↳ $$J_{w,b} = \frac{1}{2m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})^2$$

We hope the value of $J_{w,b}$ small

Examples : using cost Function Find parameter w, b

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

Parameters : w, b

Cost Function : $J_{w,b} = \frac{1}{2m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})^2$

Goal :

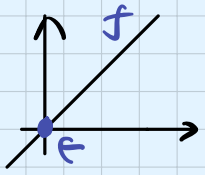
$$\underset{w,b}{\text{minimize}}\ J(w,b)$$

Simplified

↳ $f_w = wx$, $b = 0$

$J(w) = \frac{1}{2m} \sum_{i=1}^{m} (f_w(x^{(i)}) - y^{(i)})^2$

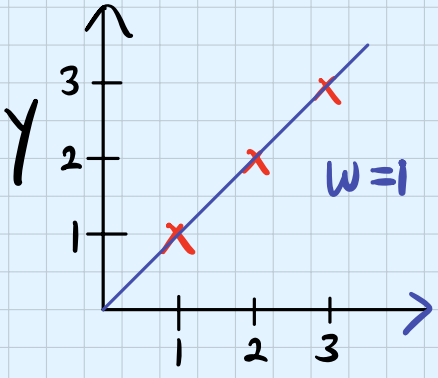
$$\underset{w}{\text{minimize}}\ J(w)$$



↖ $w(x^{(i)})$

$f_w(x)$

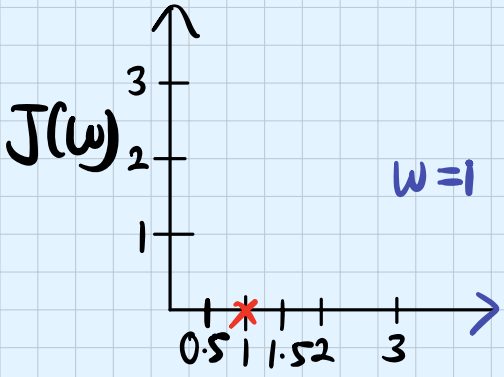
(for fixed w, function of x)



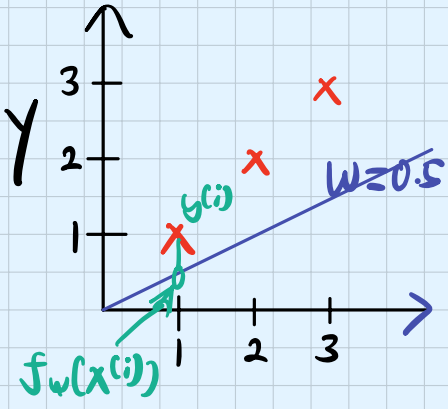
$J(w) = \frac{1}{2m}(0^2 + 0^2 + 0^2)$

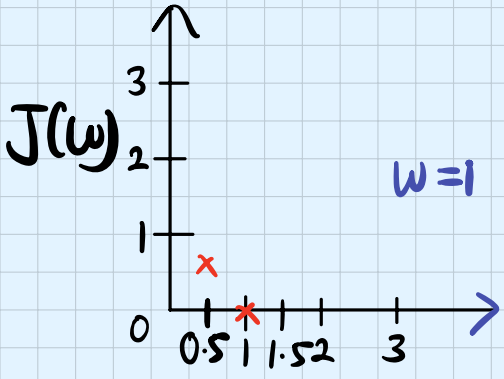
$= 0$

$J(w)$



When $w = 0.5$,



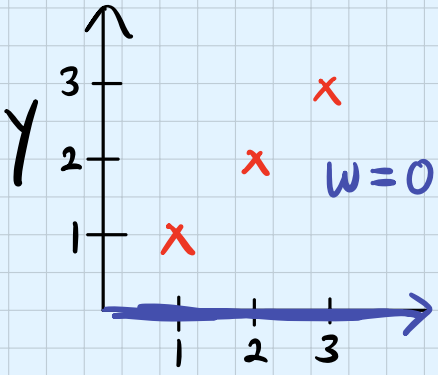


$J(0.5) = \frac{1}{2m}\left[(0.5-1)^2+(1-2)^2+(1.5-3)^2\right]$

$= \frac{1}{2\times 3}(3.5)$

≈ 0.58

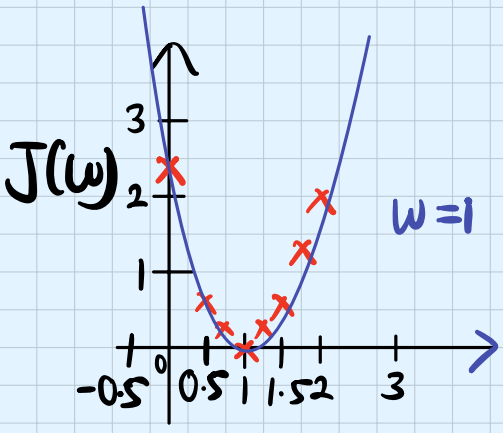
When $w=0$,



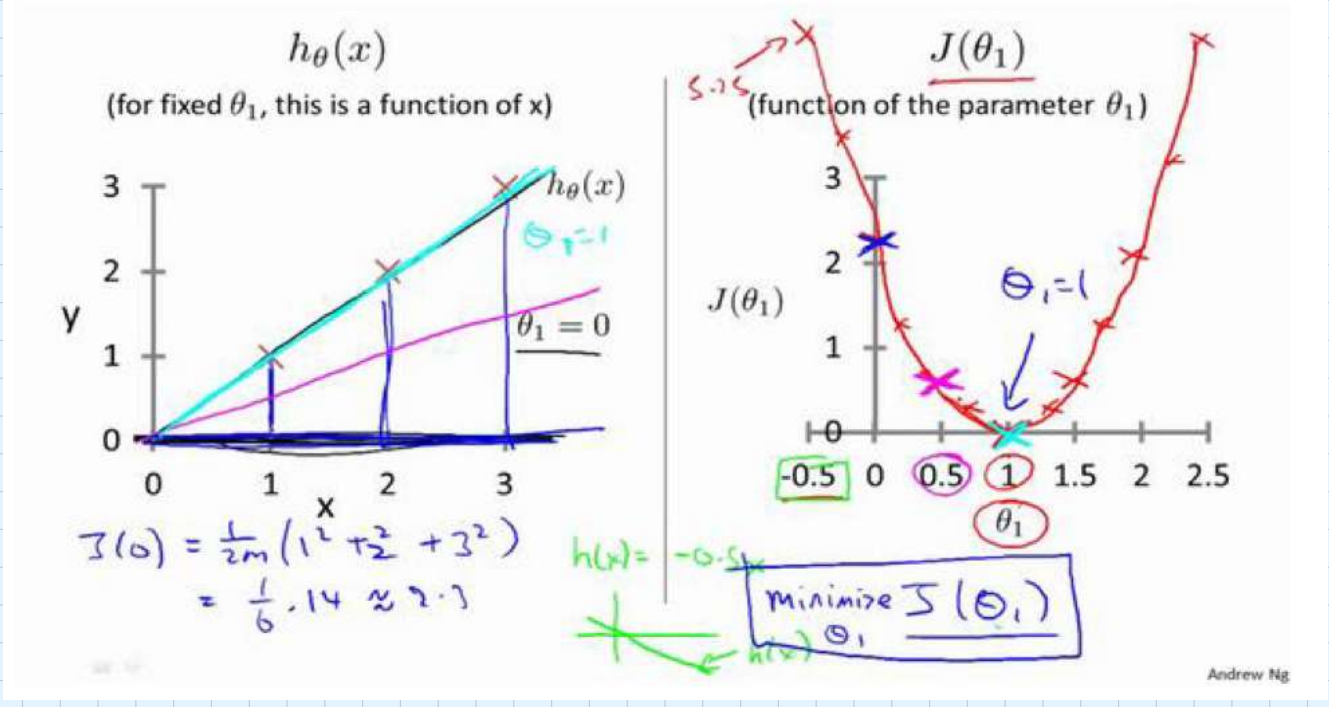
$J(0) = \frac{1}{6}\left[1^2+2^2+3^2\right]$

$= \frac{1}{6}(14)$

≈ 2.3



$J(w)$



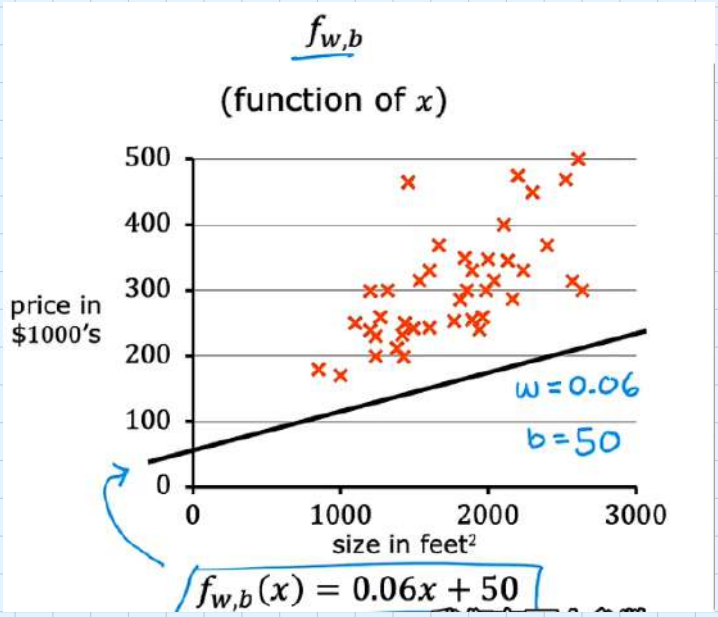
Choose w to minimize $J(w)$

In this model → $w=1$, $J\to 0$

Called Good model

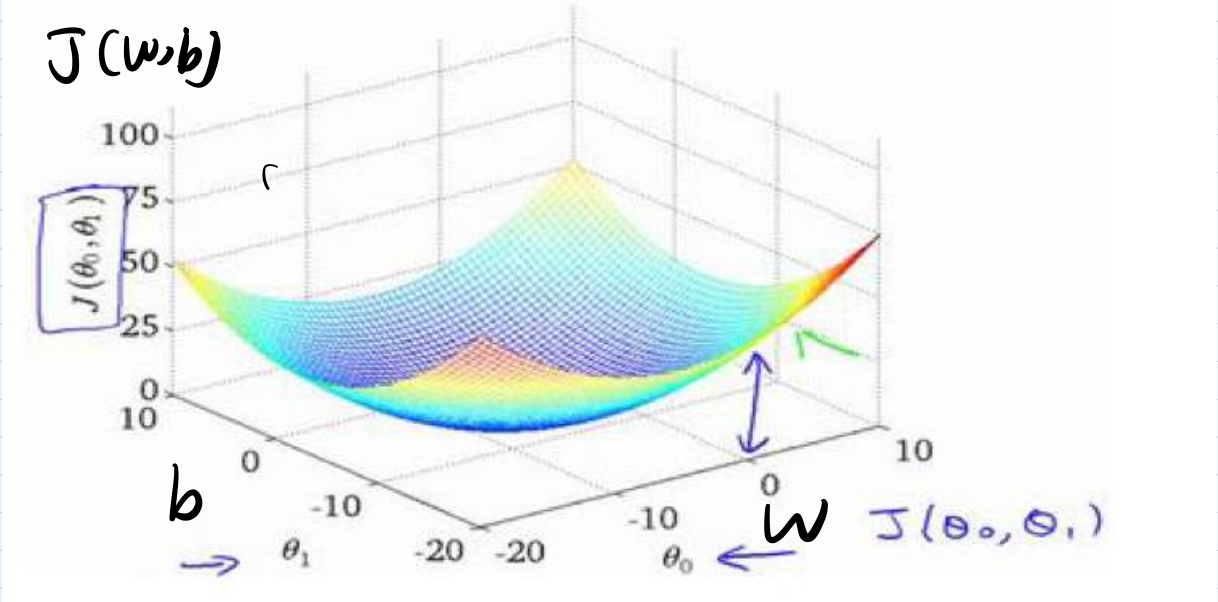
Visualization Cost Function

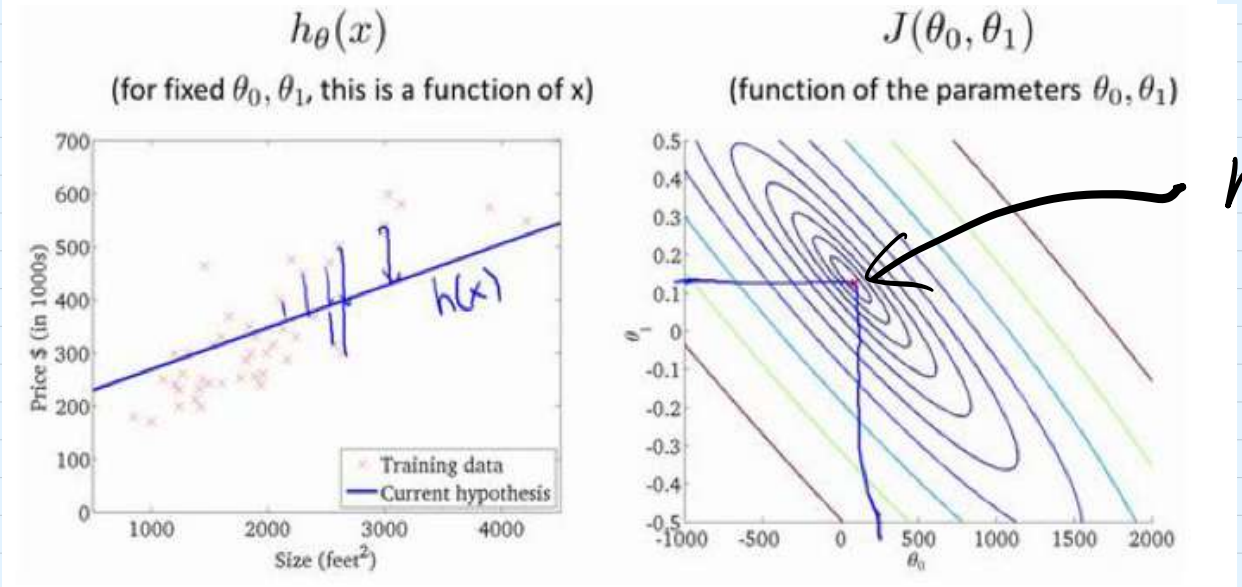
↳ with two parameter w, b



3D Graph ← 等高线

Got many Contour line

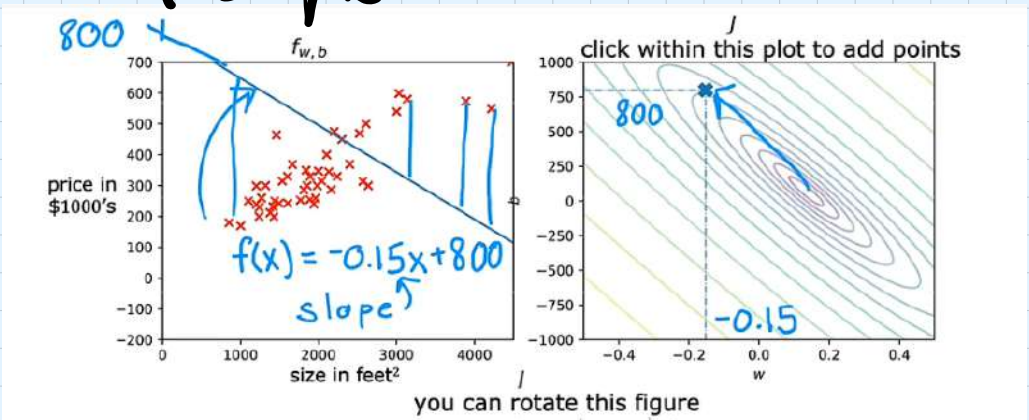




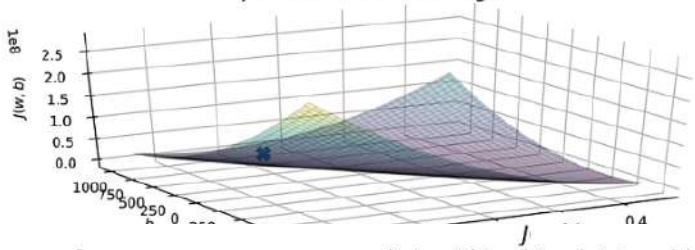
↳ center

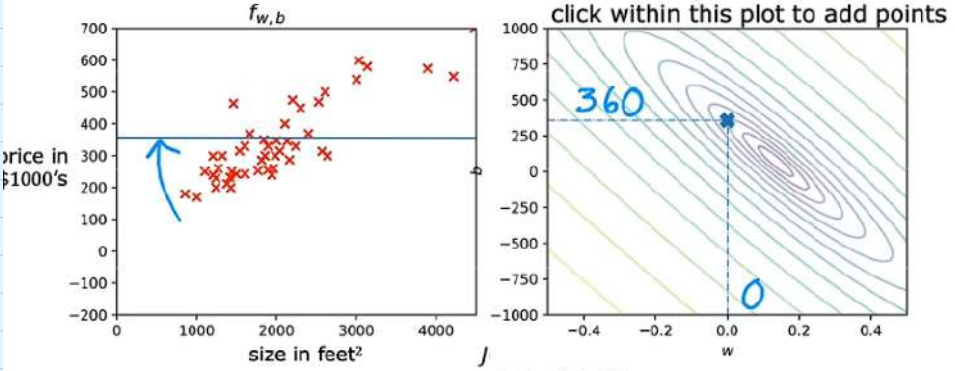
more Examples

Bad example

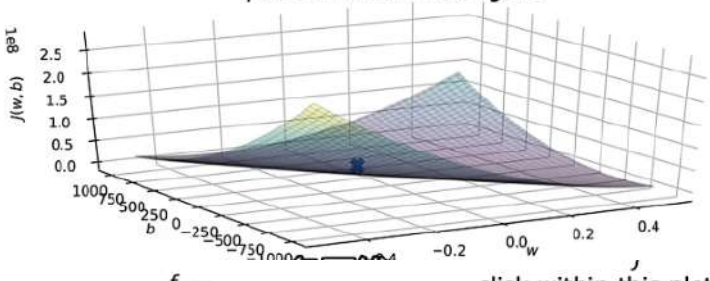


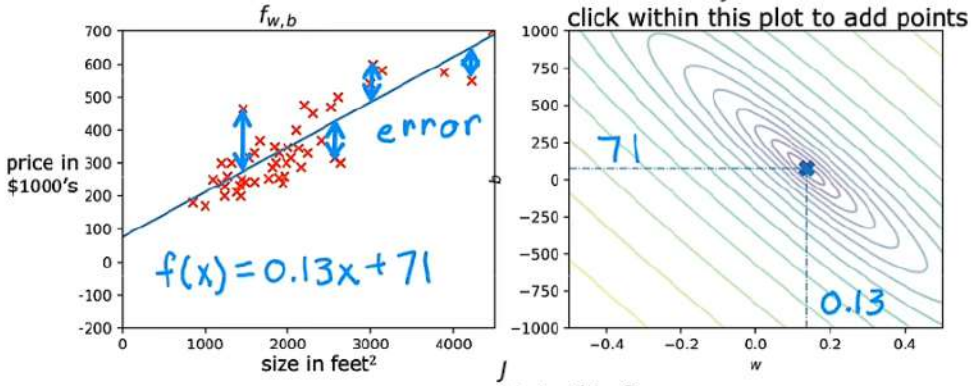
you can rotate this figure



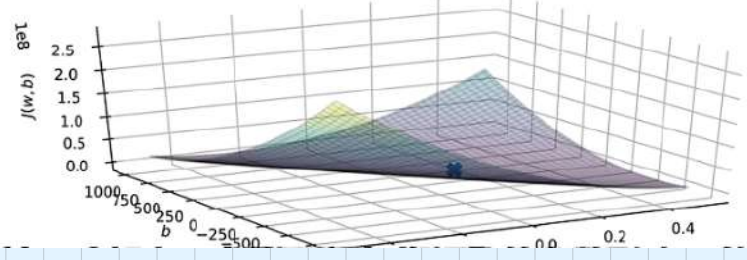


you can rotate this figure





you can rotate this figure



Got Optional Lab

∴ use code to find w,b can make J ↓

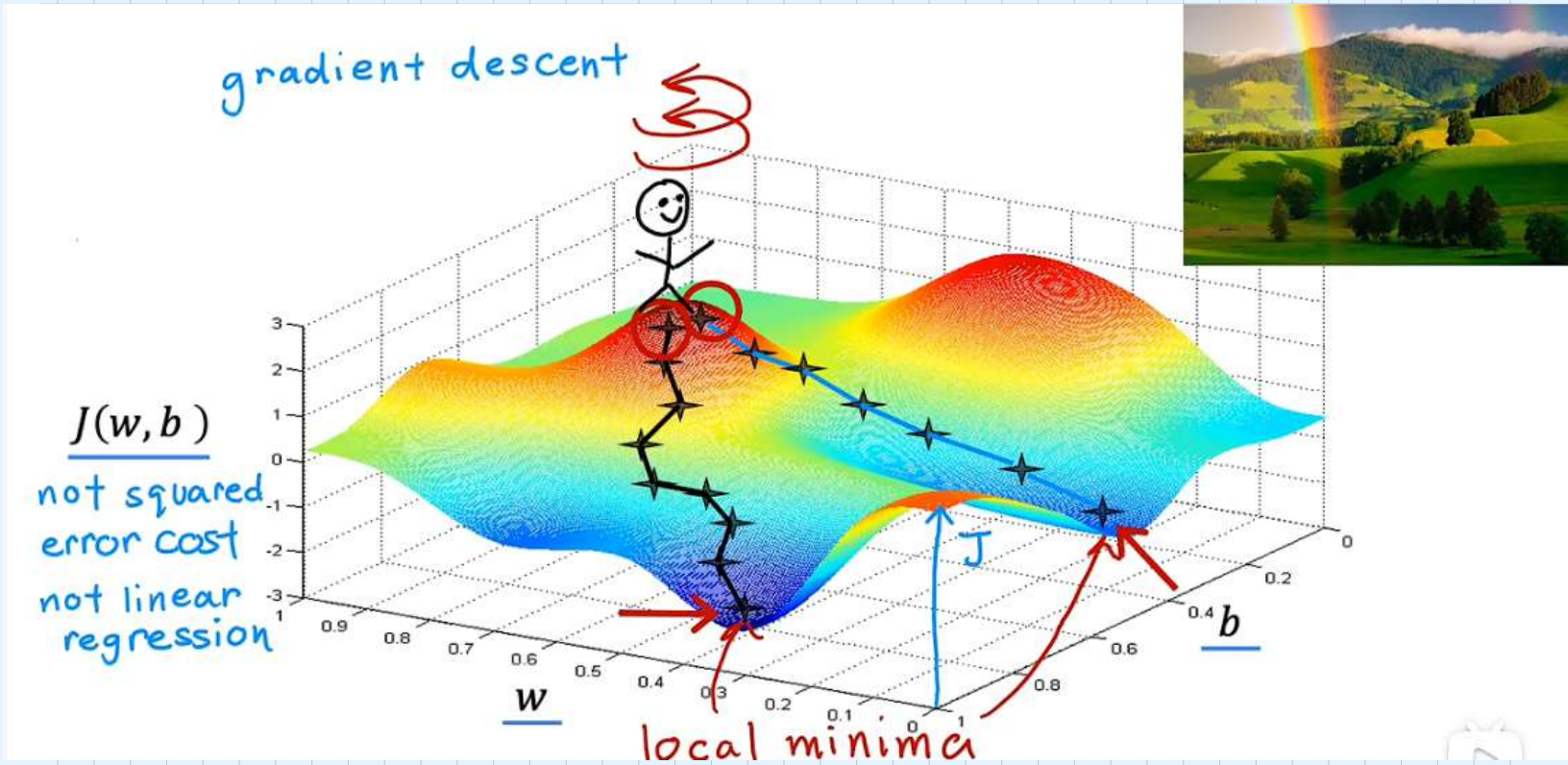
Gradient Descent 梯度下降

Have some function $J(w,b)$ → for linear regression or any function
Want $\min_{w,b} J(w,b)$

If Have

$$\min_{w_1,\dots w_n,b} J(w_1, w_2, \dots, w_n, b)$$

Start with some w,b (set $w=0$, $b=0$)
Keep changing w,b reduce $J(w,b)$
Until we settle at or near a minimum
But it got minimum >1



downhill quickly

property: ~if downhill into a minimum valley gradient descent cannot take u to another minimum valley
~ the first point of w and b can bring u downhill to diff m.v

Implement Gradient descent algorithm

$$W_{new} = W_{old} - \alpha \frac{\partial}{\partial w} J(w,b)$$

$=$: Assignment

α : Alpha, called **Learning rate**, control size of steps

$\frac{\partial}{\partial w} J(w,b)$: **Derivative**, control the direction

Remember got another parameter

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

Repeat this two equation until convergence 收斂

★ Simultaneously update w and b

同時更新

$$\left.\begin{aligned} tmp\_w &= w - \alpha \frac{\partial}{\partial w} J(w,b) \\ tmp\_b &= b - \alpha \frac{\partial}{\partial b} J(w,b) \\ w &= tmp\_w \\ b &= tmp\_b \end{aligned}\right\} \text{Correct}$$

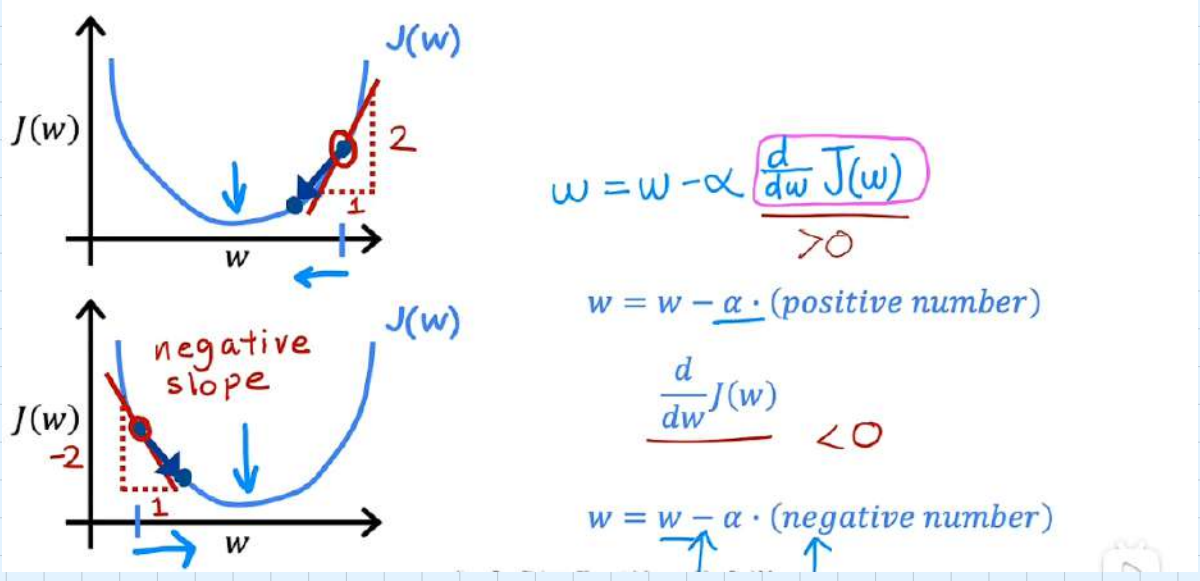
Example of derivative term

↳ If one variable when $b=0$

$J(w)$

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

$\min_w J(w)$



$w = w - \alpha \cdot \underbrace{\frac{d}{dw} J(w)}_{>0}$

$w = w - \alpha \cdot (\text{positive number})$

$\underbrace{\frac{d}{dw} J(w)}_{<0}$

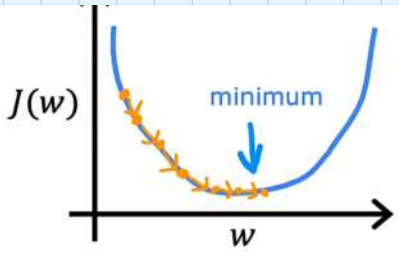
$w = w - \alpha \cdot (\text{negative number})$

Example of learning rates

If learning rate too small, downhill speed incredible slowly

learning rate too large, downhill speed incredible fastly, cost may be worst

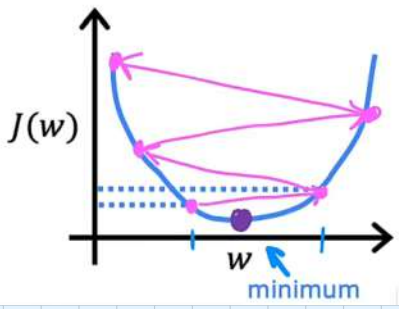
$$w = w - \alpha \frac{d}{dw} J(w)$$



If α is too small...
Gradient descent may be slow.

If α is too large...

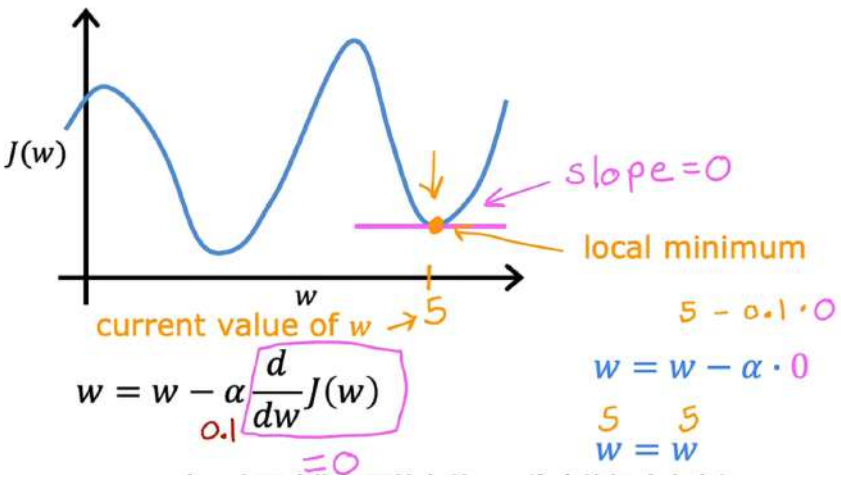
Gradient descent may:
- Overshoot, never reach minimum
- Fail to converge, diverge



If already at local minimum

$w = w - \alpha(0)$

$\therefore w = w$



$\therefore$ So can reach local m with fixed Learning rate

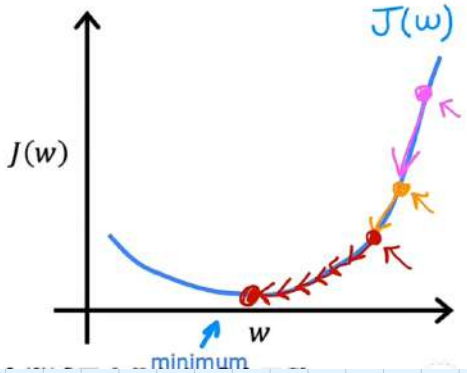
Can reach local minimum with fixed learning rate α

$$w = w - \alpha \frac{d}{dw} J(w)$$

smaller
not as large
large

Near a local minimum,
- Derivative becomes smaller
- Update steps become smaller

Can reach minimum without decreasing learning rate α



Linear Regression model with Gradient Descent

Linear Regression model : $f_{w,b} = wx + b$

Cost function : $J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)})^2$

Gradient descent algorithm: $w = w - \alpha \frac{\partial}{\partial w} J(w,b)$

repeat until convergence $b = b - \alpha \frac{\partial}{\partial b} J(w,b)$

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

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

推导: (Optional)

$$\frac{\partial}{\partial w} J(w,b) = \frac{\partial}{\partial w} \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2 = \frac{\partial}{\partial w} \frac{1}{2m} \sum_{i=1}^{m} \left(wx^{(i)} + b - y^{(i)}\right)^2$$

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

$$\frac{\partial}{\partial b} J(w,b) = \frac{\partial}{\partial b} \frac{1}{2m} \sum_{i=1}^{m} \left(f_{w,b}(x^{(i)}) - y^{(i)}\right)^2 = \frac{\partial}{\partial b} \frac{1}{2m} \sum_{i=1}^{m} \left(wx^{(i)} + b - y^{(i)}\right)^2$$

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

现在你有了这两个导数表达式。

Stanford Now you have these two expressions for the derivatives. Andrew Ng

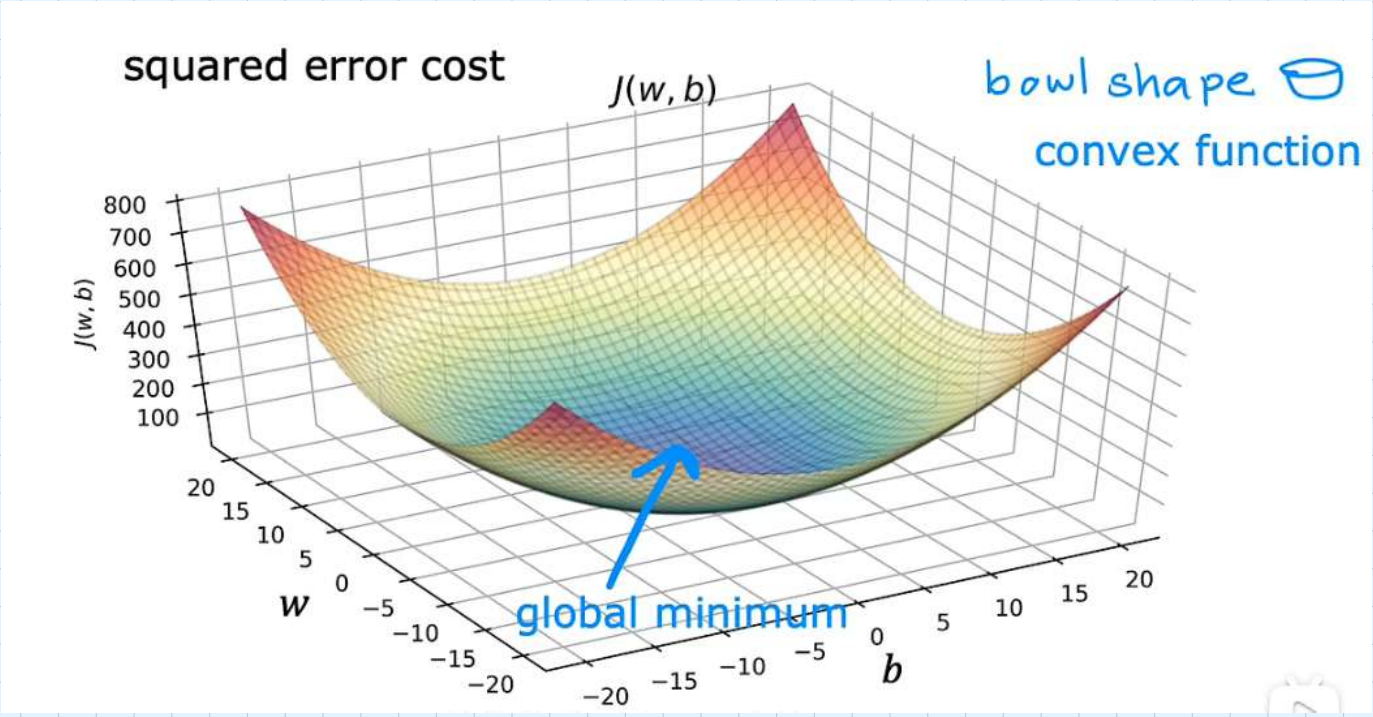
$\therefore w = w - \alpha \frac{1}{m} \sum_{i=1}^{m} (f_{w,b}(x^{(i)}) - y^{(i)}) x^{(i)}$

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

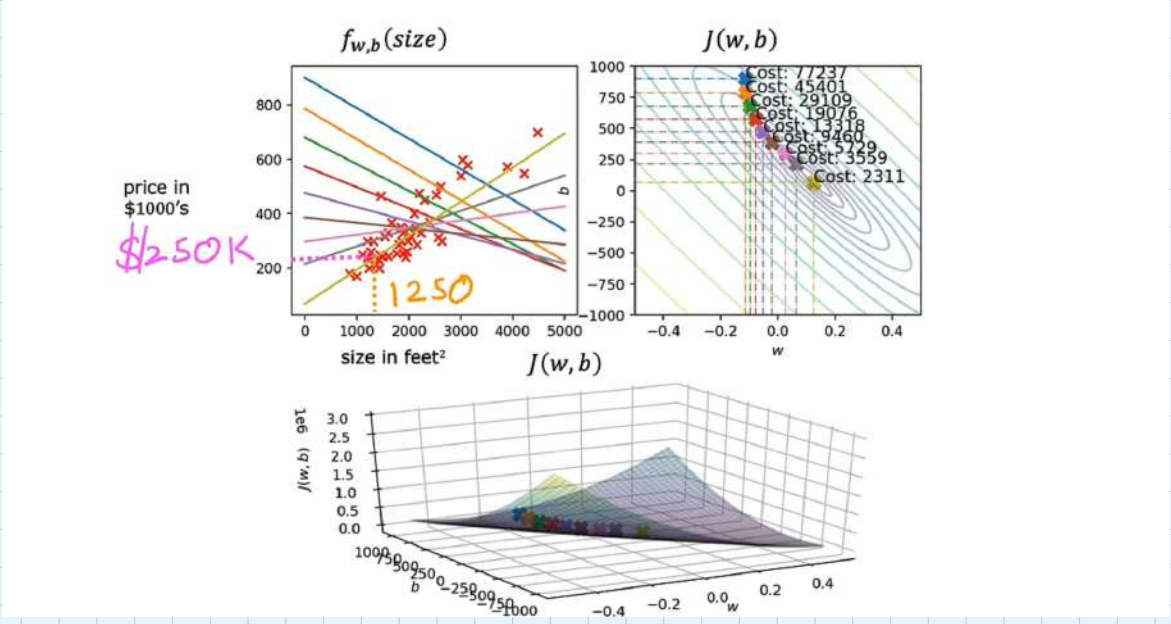
If using squared error Cost Function, it never have multiple local minima! has single global minimum

↳ bowl shape

↳ convex function



Running Gradient Descent



This Gradient Descent Called "Batch" Gradient Descent

"Batch" gradient descent

DeepLearning.AI THE BATCH

"Batch": Each step of gradient descent uses all the training examples.

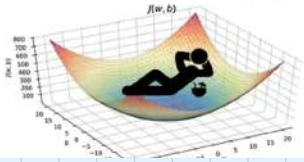
other gradient descent: subsets

OPTional LAB

	x size in feet²	y price in $1000's
(1)	2104	400
(2)	1416	232
(3)	1534	315
(4)	852	178
...	...	...
(47)	3210	870

$m = 47$

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



Use Jupyter implement Gradient Descent

Implement Gradient Descent

You will implement gradient descent algorithm for one feature. You will need three functions.

- `compute_gradient` implementing equation (4) and (5) above
- `compute_cost` implementing equation (2) above (code from previous lab)
- `gradient_descent`, utilizing compute_gradient and compute_cost

Compute_Cost

This was developed in the last lab. We'll need it again here.

```
#Function to calculate the cost
def compute_cost(x, y, w, b):

    m = x.shape[0]
    cost = 0

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

    return total_cost
```

← use shape get m

Use Linear Regression model + cost function

$$f_{w,b} = wx^{(i)} + b$$

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

compute_gradient

`compute_gradient` implements (4) and (5) above and returns $\frac{\partial J(w,b)}{\partial w}, \frac{\partial J(w,b)}{\partial b}$. The embedded comments describe the operations.

```
def compute_gradient(x, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters
    Returns
      dj_dw (scalar): The gradient of the cost w.r.t. the parameters w
      dj_db (scalar): The gradient of the cost w.r.t. the parameter b
     """

    # Number of training examples
    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
```

Find derivative

整合：Gradient Descent

```python
def gradient_descent(x, y, w_in, b_in, alpha, num_iters, cost_function, gradient_function):
    """
    Performs gradient descent to fit w,b. Updates w,b by taking
    num_iters gradient steps with learning rate alpha

    Args:
      x (ndarray (m,))  : Data, m examples
      y (ndarray (m,))  : target values
      w_in,b_in (scalar): initial values of model parameters
      alpha (float):     Learning rate
      num_iters (int):   number of iterations to run gradient descent
      cost_function:     function to call to produce cost
      gradient_function: function to call to produce gradient

    Returns:
      w (scalar): Updated value of parameter after running gradient descent
      b (scalar): Updated value of parameter after running gradient descent
      J_history (List): History of cost values
      p_history (list): History of parameters [w,b]
      """

    w = copy.deepcopy(w_in) # avoid modifying global w_in
    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    p_history = []
    b = b_in
    w = w_in

    for i in range(num_iters):
        # Calculate the gradient and update the parameters using gradient_function
        dj_dw, dj_db = gradient_function(x, y, w , b)

        # Update Parameters using equation (3) above
        b = b - alpha * dj_db
        w = w - alpha * dj_dw

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( cost_function(x, y, w , b))
            p_history.append([w,b])
        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters/10) == 0:
            print(f"Iteration {i:4}: Cost {J_history[-1]:0.2e} ",
                  f"dj_dw: {dj_dw: 0.3e}, dj_db: {dj_db: 0.3e}  ",
                  f"w: {w: 0.3e}, b:{b: 0.5e}")

    return w, b, J_history, p_history #return w and J,w history for graphing
```

```python
# initialize parameters
w_init = 0
b_init = 0
# some gradient descent settings
iterations = 10000
tmp_alpha = 1.0e-2
# run gradient descent
w_final, b_final, J_hist, p_hist = gradient_descent(x_train ,y_train, w_init, b_init, tmp_alpha,
                                                    iterations, compute_cost, compute_gradient)
print(f"(w,b) found by gradient descent: ({w_final:8.4f},{b_final:8.4f})")
```

↳ Test

Second Week → make Linear Regression more powerful

Multiple features

↳ x_j = j^{th} feature

n = number of features

$\vec{x}^{(i)}$ = feature of i^{th} training example

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

Size in feet²	Number of bedrooms	Number of floors	Age of home in years	Price ($) in $1000's
x_1	x_2	x_3	x_4	
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

$i=2$

$j=1 \dots 4$

$n=4$

Eg: $\vec{x}_3^{(2)} = 2$, $\vec{x}^{(1)} = [2104 \;\; 5 \;\; 1 \;\; 45]$

↳ row vector

Model change:

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

Example: $f_{wb}(x) = 0.1x_1 + 4x_2 + 10x_3 + -2x_4 + 80$

- x_1: size
- x_2: bedrooms
- x_3: floors
- x_4: years
- b: base price

Mutiple Linear Regression

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

$\vec{w} = [w_1 \; w_2 \; w_3 \; \cdots \; w_n]$ } parameters of the model
b is a number

$\vec{x} = [x_1 \; x_2 \; x_3 \; \cdots \; x_n]$ } features

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

↑ Dot product

Vectorization 矢量化

↳ use in algorithm, can make code shortly and efficiently

Parameters and features

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

b is a number

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

linear algebra: count from 1

NumPy

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

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

$x[0]$ $x[1]$ $x[2]$

code: count from 0

Without vectorization

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

$\sum_{j=1}^{n} \rightarrow j = 1 \ldots n$ (1, 2, 3)

range(0,n) → $j = 0 \ldots n-1$

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

Without vectorization $n = 100,000$

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

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

Vectorization

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

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

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

Without vectorization

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

t_0
f + w[0] * x[0]

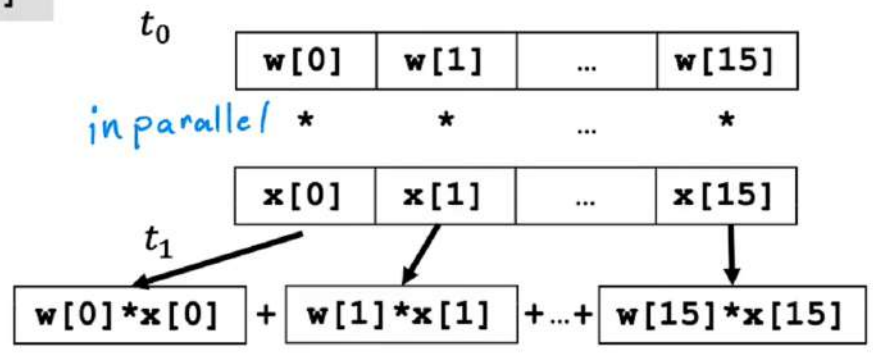
t_1
f + w[1] * x[1]

...

t_{15}
f + w[15] * x[15]

Vectorization

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



efficient → scale to large datasets

Gradient descent

$\vec{w} = (w_1 \quad w_2 \quad \cdots \quad w_{16})$ parameters

derivatives $\vec{d} = (d_1 \quad d_2 \quad \cdots \quad d_{16})$

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

compute $w_j = w_j - 0.1 d_j$ for $j = 1 \ldots 16$ (0.1: learning rate α)

Without vectorization

$w_1 = w_1 - 0.1 d_1$

$w_2 = w_2 - 0.1 d_2$

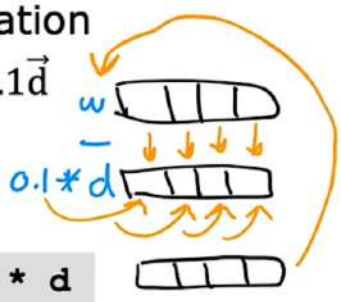
$\vdots$

$w_{16} = w_{16} - 0.1 d_{16}$

```
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
```

Optional Lab: Numpy

Gradient Descent With multiple features (Vetorization)

	Previous	Vector
Parameters	$w_1, \cdots, w_n$	$\vec{w} = [w_1 \ \cdots \ w_n]$ b still number (n: vector of length)
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \cdots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ (dot product)
Cost function	$J(w_1, \cdots, w_n, b)$	$J(\vec{w}, b)$
Gradient Descent	repeat { $w_j = w_j - \alpha \frac{\partial}{\partial w} J(w_1 \cdots w_n, b)$ $b = b - \alpha \frac{\partial}{\partial b} J(w_1, \cdots, w_n, b)$ }	repeat { $w_j = w_j - \alpha \frac{\partial}{\partial w_j} J(\vec{w}, b)$ $b = b - \alpha \frac{\partial}{\partial b} J(\vec{w}, b)$ }

Gradient descent

One feature

repeat {

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

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

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

simultaneously update w, b

}

n features ($n \geq 2$)

repeat {

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

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

$\vdots$

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

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

simultaneously update w_j (for $j = 1, \cdots, n$) and b

}

An alternative to gradient descent

→ Normal equation
- Only for linear regression
- Solve for w, b without iterations

Disadvantages
- Doesn't generalize to other learning algorithms.
- Slow when number of features is large (> 10,000)

What you need to know
- Normal equation method may be used in machine learning libraries that implement linear regression.
- Gradient descent is the recommended method for finding parameters w,b

Another algorithm Solve Linear Regression

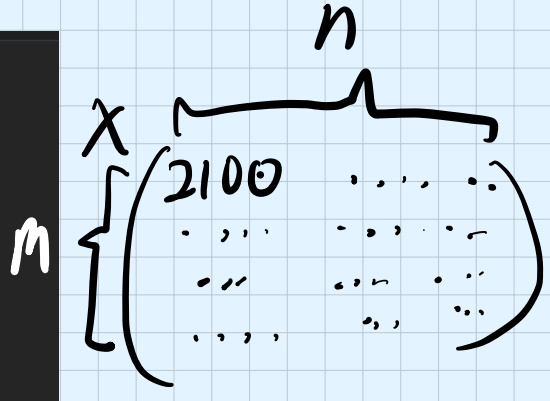
Optional Lab

```python
def compute_cost(X, y, w, b):
    """
    compute cost
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """
    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        f_wb_i = np.dot(X[i], w) + b           #(n,)(n,) = scalar (see np.dot)
        cost = cost + (f_wb_i - y[i])**2       #scalar
    cost = cost / (2 * m)                      #scalar
    return cost
```

一个example完了跑2

↳依然需for，but 一行一行(

```python
def compute_gradient(X, y, w, b):
    """
    Computes the gradient for linear regression
    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      dj_dw (ndarray (n,)): The gradient of the cost w.r.t. the parameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the parameter b.
    """
    m,n = X.shape           #(number of examples, number of features)
    dj_dw = np.zeros((n,))
    dj_db = 0.

    for i in range(m):
        err = (np.dot(X[i], w) + b) - y[i]
        for j in range(n):
            dj_dw[j] = dj_dw[j] + err * X[i, j]    ← 特殊
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```



$X_1^{(1)} = 2100$

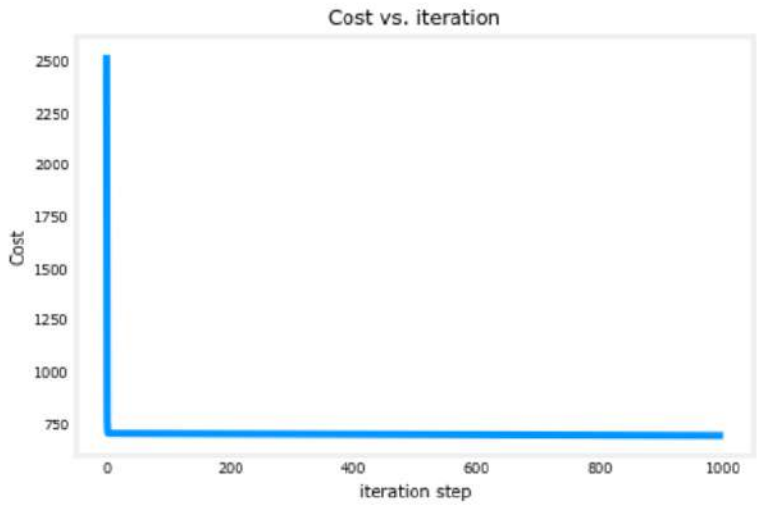
np.dot是灵魂

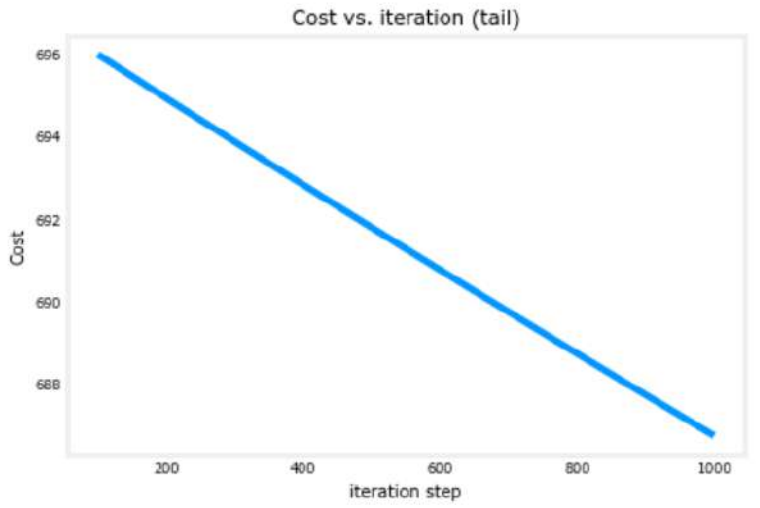
整合：

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):
    """
    Performs batch gradient descent to learn theta. Updates theta by taking
    num_iters gradient steps with learning rate alpha

    Args:
      X (ndarray (m,n))   : Data, m examples with n features
      y (ndarray (m,))    : target values
      w_in (ndarray (n,)) : initial model parameters
      b_in (scalar)       : initial model parameter
      cost_function       : function to compute cost
      gradient_function   : function to compute the gradient
      alpha (float)       : Learning rate
      num_iters (int)     : number of iterations to run gradient descent

    Returns:
      w (ndarray (n,)) : Updated values of parameters
      b (scalar)       : Updated value of parameter
      """

    # An array to store cost J and w's at each iteration primarily for graphing later
    J_history = []
    w = copy.deepcopy(w_in)  #avoid modifying global w within function
    b = b_in

    for i in range(num_iters):

        # Calculate the gradient and update the parameters
        dj_db,dj_dw = gradient_function(X, y, w, b)   ##None

        # Update Parameters using w, b, alpha and gradient
        w = w - alpha * dj_dw               ##None
        b = b - alpha * dj_db               ##None

        # Save cost J at each iteration
        if i<100000:      # prevent resource exhaustion
            J_history.append( cost_function(X, y, w, b))

        # Print cost every at intervals 10 times or as many iterations if < 10
        if i% math.ceil(num_iters / 10) == 0:
            print(f"Iteration {i:4d}: Cost {J_history[-1]:8.2f}   ")

    return w, b, J_history #return final w,b and J history for graphing
```

```
# initialize parameters
initial_w = np.zeros_like(w_init)
initial_b = 0.
# some gradient descent settings
iterations = 1000
alpha = 5.0e-7
# run gradient descent
w_final, b_final, J_hist = gradient_descent(X_train, y_train, initial_w, initial_b,
                                            compute_cost, compute_gradient,
                                            alpha, iterations)
print(f"b,w found by gradient descent: {b_final:0.2f},{w_final} ")
m,_ = X_train.shape
for i in range(m):
    print(f"prediction: {np.dot(X_train[i], w_final) + b_final:0.2f}, target value: {y_train[i]}")
```

Result are not inspiring!
Cost still declining and our prediction not very accurate






```
b,w found by gradient descent: -0.00,[ 0.2    0.    -0.01 -0.07]
prediction: 426.19, target value: 460
prediction: 286.17, target value: 232
prediction: 171.47, target value: 178
```

Feature Scaling 特征缩放 → Solve derivative Part

Feature and parameter values

$\widehat{price} = w_1 x_1 + w_2 x_2 + b$

x_1: size (feet^2) range: 300 – 2,000 large

x_2: # bedrooms range: 0 – 5 small

House: $x_1 = 2000$, $x_2 = 5$, $price = \$500k$ one training example

size of the parameters w_1, w_2?

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

$\widehat{price} = \underbrace{50 * 2000}_{100{,}000k} + \underbrace{0.1 * 5}_{0.5k} + \underbrace{50}_{50k}$

$\widehat{price} = \$100{,}050.5k = \$100{,}050{,}500$

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

small large

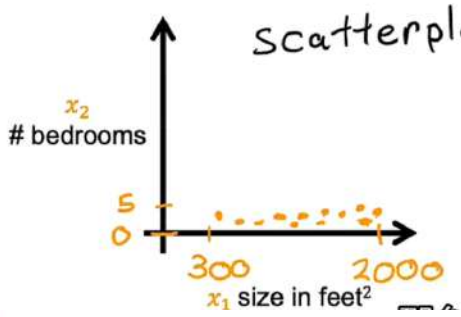
$\widehat{price} = \underbrace{0.1 * 2000k}_{200k} + \underbrace{50 * 5}_{250k} + \underbrace{50}_{50k}$

$\widehat{price} = \$500k$ more reasonable

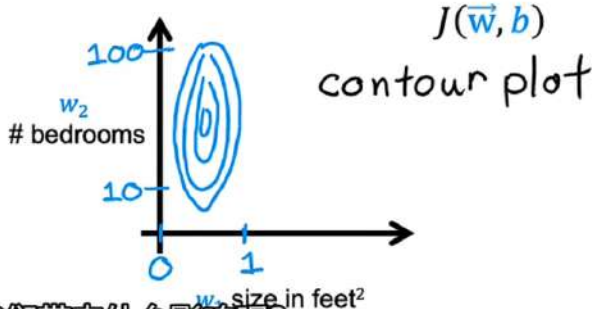
Feature size and parameter size

	size of feature x_j	size of parameter w_j
size in feet^2	⟷	⟷
#bedrooms	⟷	⟷

Features

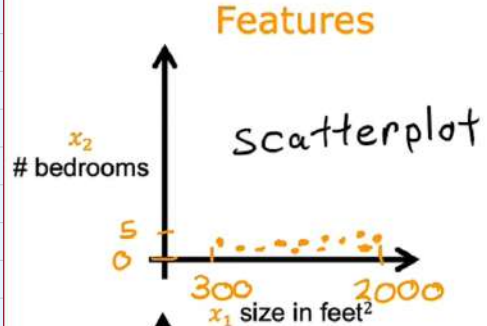


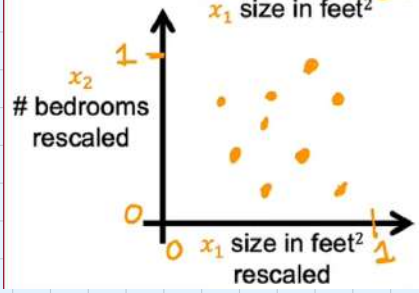
Parameters



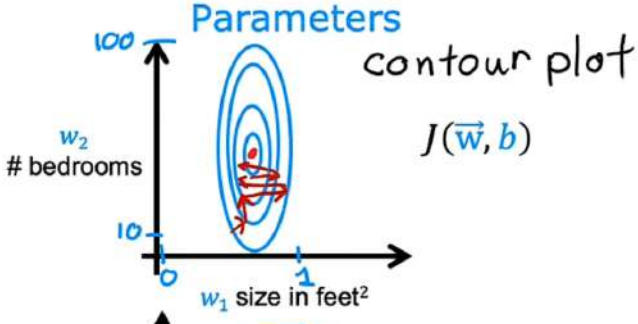
Feature size and gradient descent

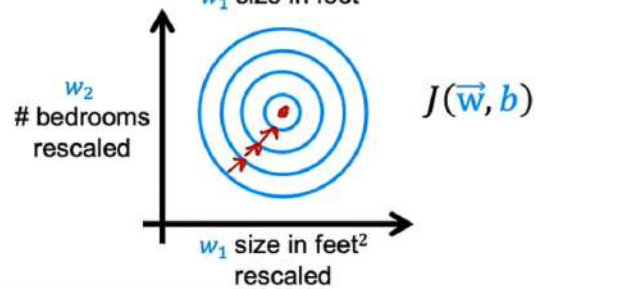
Features





Parameters





Go skill difference Range to have comparable range

① Dividing by max

$300 \le x_1 \le 2000$

$\downarrow$

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

$\downarrow$

$0.15 \le x_{1,scaled} \le 1$

$0 \le x_2 \le 5$

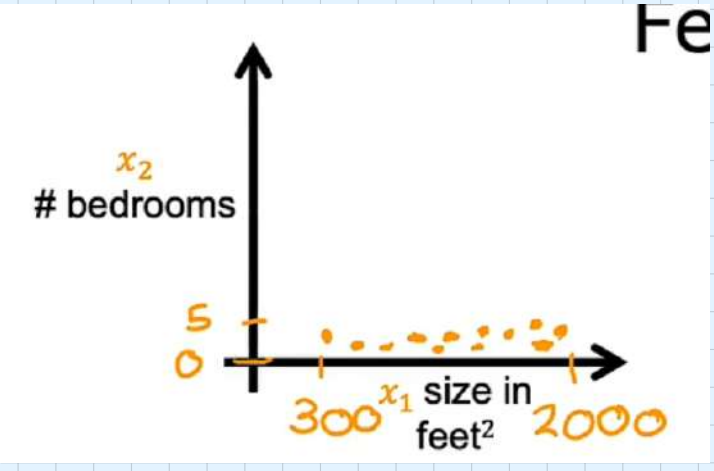
$\downarrow$

$x_{2,scaled} = \frac{x_2}{5}$

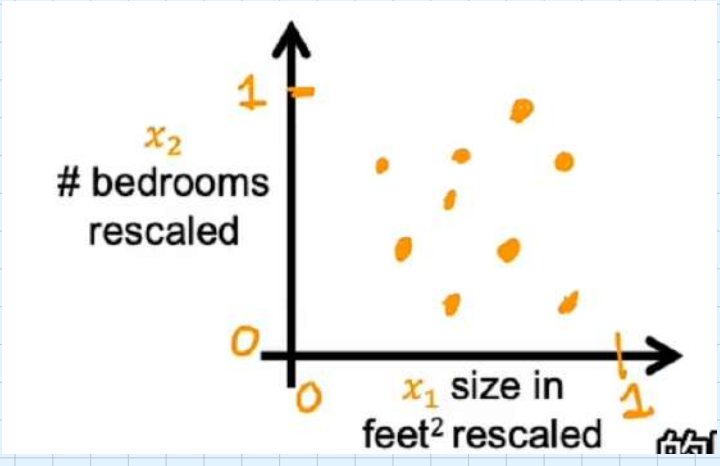
$\downarrow$

$0 \le x_2 \le 1$

Graph will from Left to Right



Fe



② Mean normalization 均一化

Find average μ_1

$\mu_1 = 600$

$300 \le x_1 \le 2000$

$\downarrow$

$x_1 = \frac{x_1 - \mu_1}{2000 - 300}$

$\downarrow$

$-0.18 \le x_1 \le 0.82$

$\mu_2 = 2.3$

$0 \le x_2 \le 5$

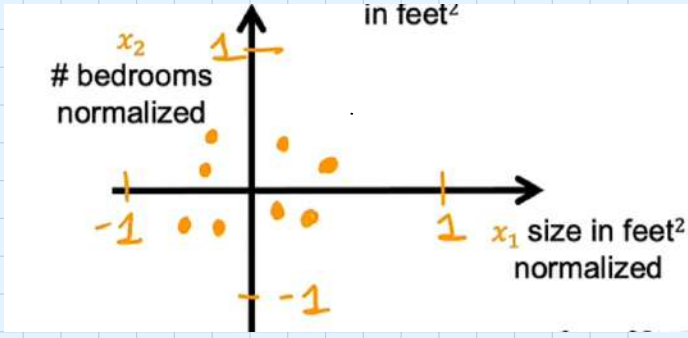
$\downarrow$

$x_2 = \frac{x_2 - \mu_1}{5 - 0}$

$\downarrow$

$-0.46 \le x_2 \le 0.54$

Graph became



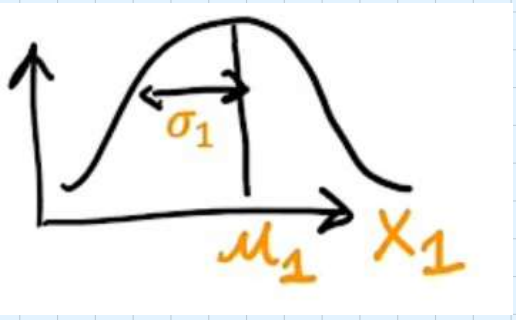
③ Z-score normalization

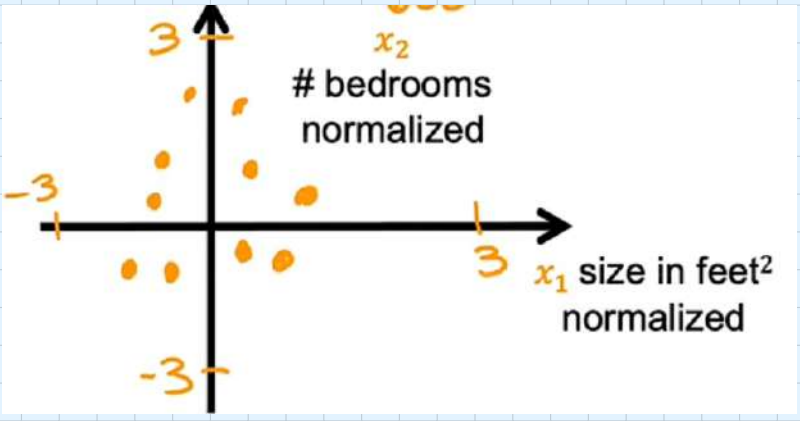
→ Find standard deviation σ

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

$x_1 = \frac{x_1 - \mu_1}{\sigma_1}$ $\quad$ $x_2 = \frac{x_2 - \mu_2}{\sigma_2}$

$-0.67 \le x_1 \le 3.1$ $\quad$ $-1.6 \le x_2 \le 1.9$





Feature scaling

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

$0 \le x_1 \le 3$ — okay, no rescaling

$-2 \le x_2 \le 0.5$ — okay, no rescaling

$-100 \le x_3 \le 100$ — too large → rescale

$-0.001 \le x_4 \le 0.001$ — too small → rescale

$98.6 \le x_5 \le 105$ — too large → rescale

↳ temperature

Checking Gradient Descent for Convergence

→ check Convergence & Solve iteration

Gradient descent

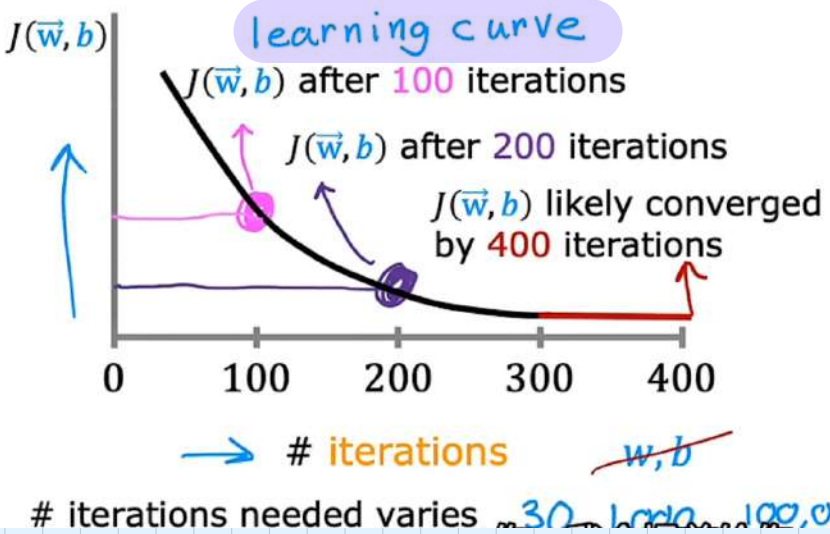
$$\begin{cases} w_j = w_j - \alpha \dfrac{\partial}{\partial w_j} J(\vec{w}, b) \\ b = b - \alpha \dfrac{\partial}{\partial b} J(\vec{w}, b) \end{cases}$$

most used

Make sure gradient descent is working correctly

objective: $\min_{\vec{w},b} J(\vec{w}, b)$

$J(\vec{w}, b)$ should decrease after every iteration



~~w, b~~

iterations needed varies 30 ... 100,000

Automatic convergence test

Let ε "epsilon" be 10^{-3}. 0.001

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

(found parameters $\vec{w}, b$ to get close to global minimum)

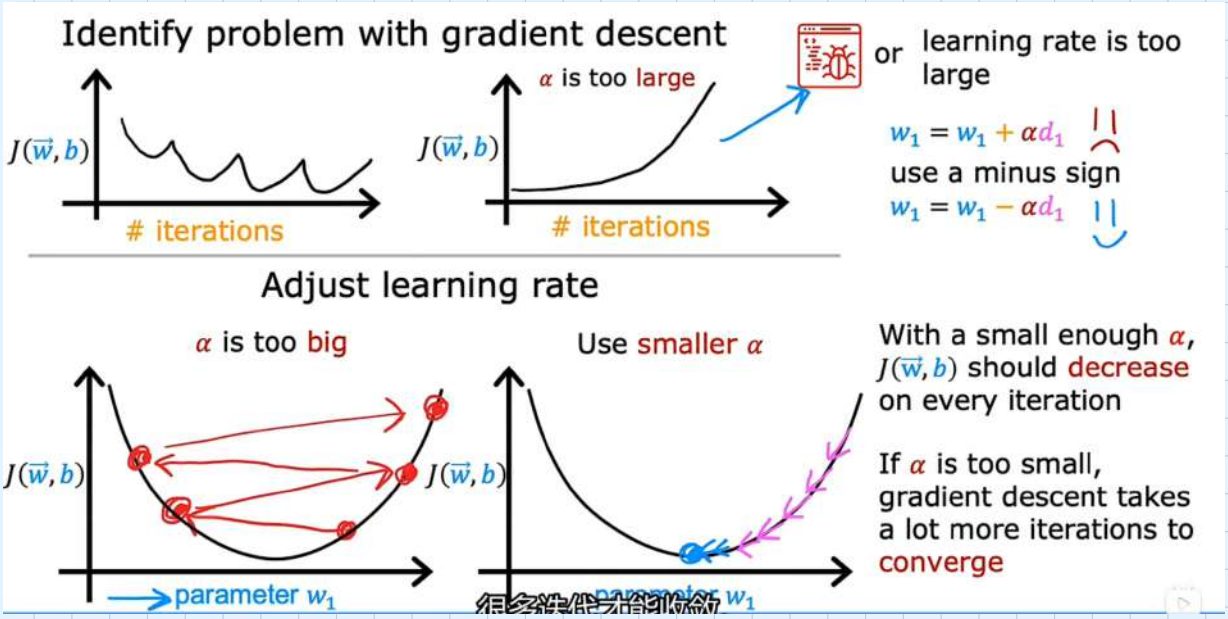
↳ different app got diff iterations

→ If divergence, may be learning rate too large

→ we will see diff learning curve in diff model

∝ decide iterations

Choosing the learning rate → Solve Alpha part



① with a small α

② broken code

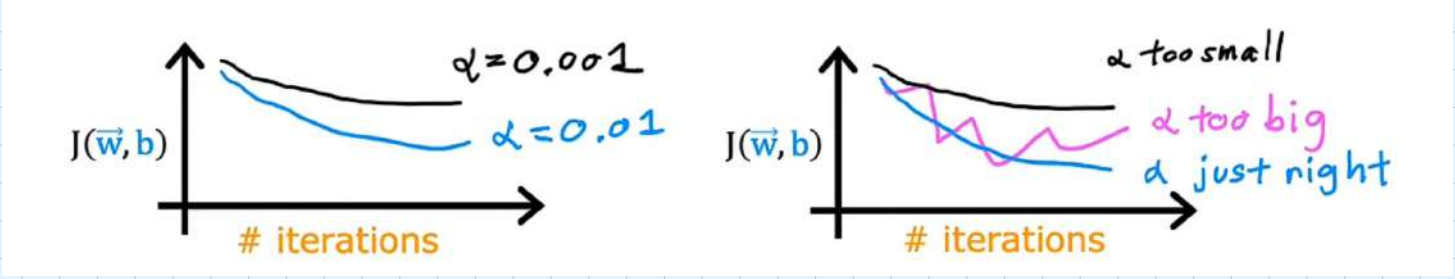
Values of α to try

: ... 0.001 → 0.01 → 0.1 → 1...

Cansee on graph

try 3x

0.003 → 0.03 → 0.3 or 1



Optional LAB

→ Features Scale

→ Try α

α = 9.9e-7

```
In [ ]: #set alpha to 9.9e-7
        _, _, hist = run_gradient_descent(X_train, y_train, 10, alpha = 9.9e-7)
```

It appears the learning rate is too high. The solution does not converge. Cost is *increasing* rather than decreasing. Let's plot the result:

```
In [ ]: plot_cost_i_w(X_train, y_train, hist)
```

The plot on the right shows the value of one of the parameters, w_0. At each iteration, it is overshooting the optimal value and as a result, cost ends up *increasing* rather than approaching the minimum. Note that this is not a completely accurate picture as there are 4 parameters being modified each pass rather than just one. This plot is only showing w_0 with the other parameters fixed at benign values. In this and later plots you may notice the blue and orange lines being slightly off.

α = 9e-7

Let's try a bit smaller value and see what happens.

```
In [ ]: #set alpha to 9e-7
        _, _, hist = run_gradient_descent(X_train, y_train, 10, alpha = 9e-7)
```

Cost is decreasing throughout the run showing that alpha is not too large.

```
In [ ]: plot_cost_i_w(X_train, y_train, hist)
```

On the left, you see that cost is decreasing as it should. On the right, you can see that w_0 is still oscillating around the minimum, but it is decreasing each iteration rather than increasing. Note above that `dj_dw[0]` changes sign with each iteration as `w[0]` jumps over the optimal value. This alpha value will converge. You can vary the number of iterations to see how it behaves.

α = 1e-7

Let's try a bit smaller value for α and see what happens.

```
In [ ]: #set alpha to 1e-7
        _, _, hist = run_gradient_descent(X_train, y_train, 10, alpha = 1e-7)
```

Cost is decreasing throughout the run showing that α is not too large.

try learning rate

→ Best Only didn't scale

Scaling

z-score normalization

After z-score normalization, all features will have a mean of 0 and a standard deviation of 1.

To implement z-score normalization, adjust your input values as shown in this formula:

$$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j} \tag{4}$$

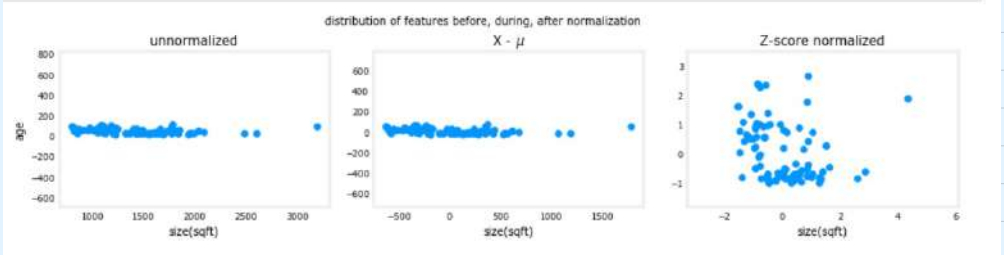
where j selects a feature or a column in the X matrix. μ_j is the mean of all the values for feature (j) and σ_j is the standard deviation of feature (j).

$$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} x_j^{(i)} \tag{5}$$

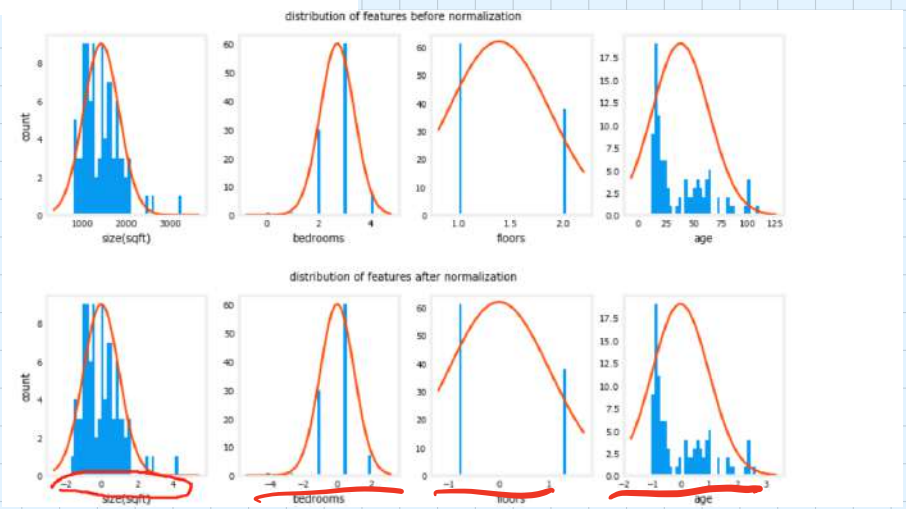
$$\sigma_j^2 = \frac{1}{m}\sum_{i=0}^{m-1} (x_j^{(i)} - \mu_j)^2 \tag{6}$$

```
def zscore_normalize_features(X):
    """
    computes  X, zcore normalized by column

    Args:
      X (ndarray): Shape (m,n) input data, m examples, n features

    Returns:
      X_norm (ndarray): Shape (m,n)  input normalized by column
      mu (ndarray):     Shape (n,)   mean of each feature
      sigma (ndarray):  Shape (n,)   standard deviation of each feature
    """
    # find the mean of each column/feature
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)

#check our work
#from sklearn.preprocessing import scale
#scale(X_orig, axis=0, with_mean=True, with_std=True, copy=True)
```

use numpy



→ Comparable



become small

make lines become Cost Contour

Features engineering ↝ use intuition
↳ Custom Linear Regression, can define new features

Feature engineering

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

x_1: frontage, x_2: depth

$$area = frontage \times depth$$

$$x_3 = x_1 x_2$$

new feature

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

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

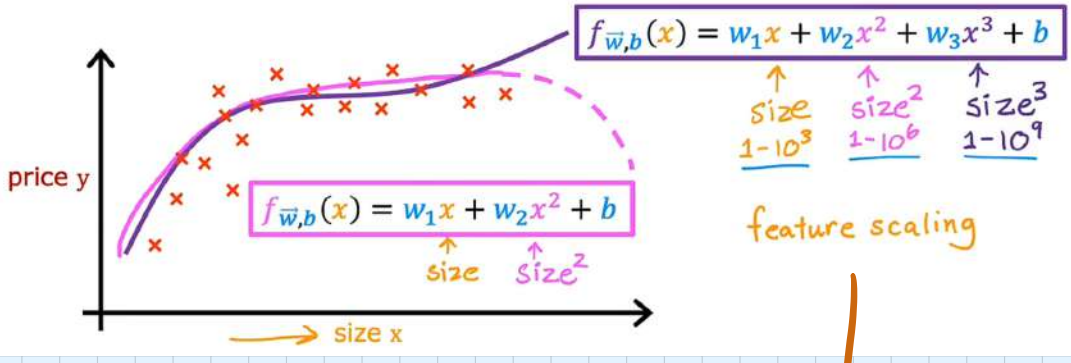
Polynomial Regression ↝ let fit curve, non-linear function to data

↳ Caused by Features engineering

Choose x_3 not only x_2

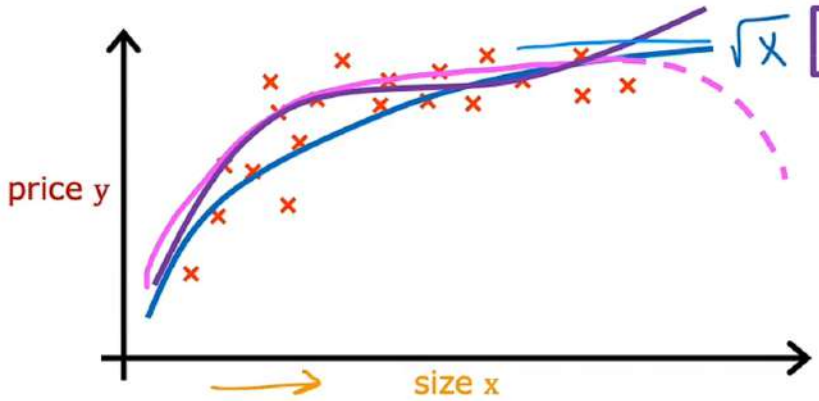
↓ size cube ↳ sizeSquare

Polynomial regression



↳ become important

Choice of features



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

size √size

what features to use?
↳ course 2

Optional lab

→ Scikit-learn (tools)

→ features & Poly

→ features eng + Poly

np.c_ emphasize at X^2

```
In [11]: # create target data
x = np.arange(0, 20, 1)
X = np.c_[x, x**2, x**3]
print(f"Peak to Peak range by column in Raw        X:{np.ptp(X,axis=0)}")

# add mean_normalization
X = zscore_normalize_features(X)
print(f"Peak to Peak range by column in Normalized X:{np.ptp(X,axis=0)}")
```

```
Peak to Peak range by column in Raw        X:[  19  361 6859]
Peak to Peak range by column in Normalized X:[3.3  3.18 3.28]
```

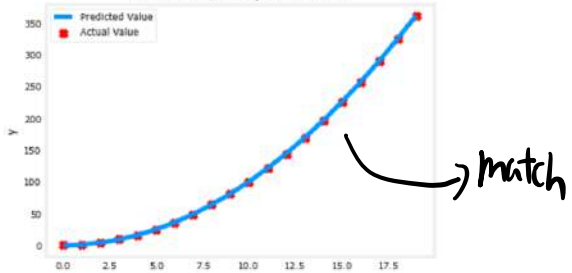
```
In [13]: x= np.arange(0, 20, 1)
y = x**2

X = np.c_[x, x**2, x**3]
X = zscore_normalize_features(X)

model_w, model_b = run_gradient_descent_feng(X, y, iterations=100000, alpha=1e-1)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized x x**2, x**3 feature")
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.ylabel("y"); plt.legend(); plt.show
```

target

→ After Scaling features

```
Iteration         0, Cost: 9.42147e+03
Iteration     10000, Cost: 3.90938e-01
Iteration     20000, Cost: 2.78389e-02
Iteration     30000, Cost: 1.98242e-03
Iteration     40000, Cost: 1.41169e-04
Iteration     50000, Cost: 1.00527e-05
Iteration     60000, Cost: 7.15855e-07
Iteration     70000, Cost: 5.09763e-08
Iteration     80000, Cost: 3.63004e-09
Iteration     90000, Cost: 2.58497e-10
w,b found by gradient descent: w: [5.27e-05 1.13e+02 8.43e-05], b: 123.5000
```

emphasize

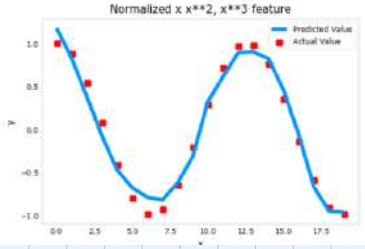


match

Complex model also can be modified

```
X = np.c_[x, x**2, x**3,x**4, x**5, x**6, x**7, x**8, x**9, x**10, x**11, x**12, x**13]
X = zscore_normalize_features(X)

model_w,model_b = run_gradient_descent_feng(X, y, iterations=1000000, alpha = 1e-1)

plt.scatter(x, y, marker='x', c='r', label="Actual Value"); plt.title("Normalized x x**2, x**3 feature")
plt.plot(x,X@model_w + model_b, label="Predicted Value"); plt.xlabel("x"); plt.ylabel("y"); plt.legend(); plt.show()
```

```
Iteration         0, Cost: 2.24887e-01
Iteration    100000, Cost: 2.31061e-02
Iteration    200000, Cost: 1.83619e-02
Iteration    300000, Cost: 1.47950e-02
Iteration    400000, Cost: 1.21114e-02
Iteration    500000, Cost: 1.00914e-02
Iteration    600000, Cost: 8.57025e-03
Iteration    700000, Cost: 7.42385e-03
Iteration    800000, Cost: 6.55908e-03
Iteration    900000, Cost: 5.90594e-03
w,b found by gradient descent: w: [-1.61e+00 -1.01e+01  3.00e+01 -6.92e-01 -2.37e+01 -1.51e+01  2.09e+01
 -2.29e-03 -4.69e-03  5.51e-02  1.07e-01 -2.53e-02  6.49e-02], b: -0.0073
```



Scikit-Learn

↳ toolkit of open-source
→ contains implementation of many algorithms

Use sklearn implement Linear Regression

```
In [1]: import numpy as np
        np.set_printoptions(precision=2)
        from sklearn.linear_model import LinearRegression, SGDRegressor
        from sklearn.preprocessing import StandardScaler
        from lab_utils_multi import  load_house_data
        import matplotlib.pyplot as plt
        dlblue = '#0096ff'; dlorange = '#FF9300'; dldarkred='#C00000'; dlmagenta='#FF40FF'; dlpurple='#7030A0';
        plt.style.use('./deeplearning.mplstyle')
```

```
In [2]: X_train, y_train = load_house_data()
        X_features = ['size(sqft)','bedrooms','floors','age']
```

```
In [3]: #Scale/normalize the training data
```

```
In [5]: scaler = StandardScaler()
        X_norm = scaler.fit_transform(X_train)
        print(f"Peak to Peak range by column in Raw        X:{np.ptp(X_train,axis=0)}")
        print(f"Peak to Peak range by column in Normalized X:{np.ptp(X_norm,axis=0)}")
```

→ call scaler f
→ scale

```
Peak to Peak range by column in Raw        X:[2.41e+03 4.00e+00 1.00e+00 9.50e+01]
Peak to Peak range by column in Normalized X:[5.85 6.14 2.06 3.69]
```

```
In [6]: #Create and fit the regression model
```

```
In [7]: sgdr = SGDRegressor(max_iter=1000)
        sgdr.fit(X_norm, y_train)
        print(sgdr)
        print(f"number of iterations completed: {sgdr.n_iter_}, number of weight updates: {sgdr.t_}")
```

```
SGDRegressor()
number of iterations completed: 117, number of weight updates: 11584.0
```

sgdr.fit(X_train)
sgd .n_iter_
sgdr.t

```
In [8]: #View parameters
```

```
In [9]: b_norm = sgdr.intercept_
        w_norm = sgdr.coef_
        print(f"model parameters:                   w: {w_norm}, b:{b_norm}")
        print(f"model parameters from previous lab: w: [110.56 -21.27 -32.71 -37.97], b: 363.16")
```

```
model parameters:                   w: [110.03 -20.99 -32.41 -38.08], b:[363.18]
model parameters from previous lab: w: [110.56 -21.27 -32.71 -37.97], b: 363.16
```

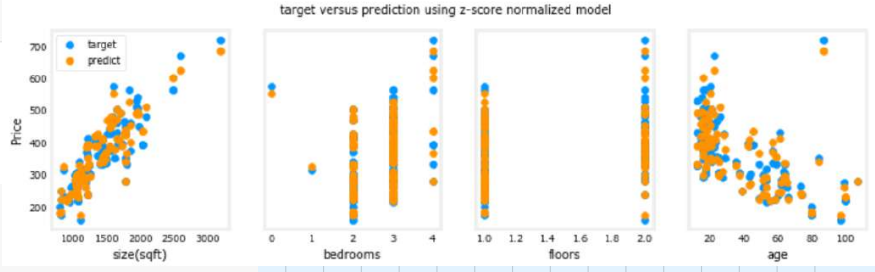
w_norm = sgdr.coef_
b_norm = sgdr.intercept_

```
In [10]: #Make predictions
```

```
In [11]: # make a prediction using sgdr.predict()
         y_pred_sgd = sgdr.predict(X_norm)
         # make a prediction using w,b.
         y_pred = np.dot(X_norm, w_norm) + b_norm
         print(f"prediction using np.dot() and sgdr.predict match: {(y_pred == y_pred_sgd).all()}")

         print(f"Prediction on training set:\n{y_pred[:4]}" )
         print(f"Target values \n{y_train[:4]}")
```

```
prediction using np.dot() and sgdr.predict match: True
Prediction on training set:
[295.21 485.91 389.64 492.07]
Target values
[300.   509.8 394.   540. ]
```

sgdr.predict(X_norm)




```
# plot predictions and targets vs original features
fig,ax=plt.subplots(1,4,figsize=(12,3),sharey=True)
for i in range(len(ax)):
    ax[i].scatter(X_train[:,i],y_train, label = 'target')
    ax[i].set_xlabel(X_features[i])
    ax[i].scatter(X_train[:,i],y_pred,color=dlorange, label = 'predict')
ax[0].set_ylabel("Price"); ax[0].legend();
fig.suptitle("target versus prediction using z-score normalized model")
plt.show()
```

By Linear Regression in Sklearn

In [1]:
```
import numpy as np
np.set_printoptions(precision=2)
from sklearn.linear_model import LinearRegression, SGDRegressor
from sklearn.preprocessing import StandardScaler
from lab_utils_multi import  load_house_data
import matplotlib.pyplot as plt
dlblue = '#0096ff'; dlorange = '#FF9300'; dldarkred='#C00000'; dlmagenta='#FF40FF'; dlpurple='#7030A0';
plt.style.use('./deeplearning.mplstyle')
```

In [2]:
```
X_train = np.array([1.0, 2.0])   #features
y_train = np.array([300, 500])   #target value
```

In [3]:
```
linear_model = LinearRegression()
#X must be a 2-D Matrix
linear_model.fit(X_train.reshape(-1, 1), y_train)
```

Out[3]:
```
LinearRegression()
```

In [4]:
```
b = linear_model.intercept_
w = linear_model.coef_
print(f"w = {w:}, b = {b:0.2f}")
print(f"'manual' prediction: f_wb = wx+b : {1200*w + b}")
```

```
w = [200.], b = 100.00
'manual' prediction: f_wb = wx+b : [240100.]
```

In [5]:
```
y_pred = linear_model.predict(X_train.reshape(-1, 1))

print("Prediction on training set:", y_pred)

X_test = np.array([[1200]])
print(f"Prediction for 1200 sqft house: ${linear_model.predict(X_test)[0]:0.2f}")
```

```
Prediction on training set: [300. 500.]
Prediction for 1200 sqft house: $240100.00
```

In [6]:
```
#second example
```

In [7]:
```
# load the dataset
X_train, y_train = load_house_data()
X_features = ['size(sqft)','bedrooms','floors','age']
```

In [9]:
```
linear_model=LinearRegression()
linear_model.fit(X_train,y_train)
```

Out[9]:
```
LinearRegression()
```

In [11]:
```
b=linear_model.intercept_
w=linear_model.coef_
print(f"w = {w:}, b = {b:0.2f}")
```

```
w = [  0.27 -32.62 -67.25  -1.47], b = 220.42
```

In [12]:
```
print(f"Prediction on training set:\n {linear_model.predict(X_train)[:4]}" )
print(f"prediction using w,b:\n {(X_train @ w + b)[:4]}")
print(f"Target values \n {y_train[:4]}")

x_house = np.array([1200, 3,1, 40]).reshape(-1,4)
x_house_predict = linear_model.predict(x_house)[0]
print(f" predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old = ${x_house_predict*1000:0.2f}")
```

```
Prediction on training set:
 [295.18 485.98 389.52 492.15]
prediction using w,b:
 [295.18 485.98 389.52 492.15]
Target values
 [300.  509.8 394.  540. ]
 predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old = $318709.09
```

Third Week ⇝ Logistics Regression 逻辑回归

Motivation

Classification

Question	Answer "y"
Is this email spam?	no yes
Is the transaction fraudulent?	no yes
Is the tumor malignant?	no yes

y can only be one of two values

"binary classification"

class = category

false true

0 1

useful for classification

"negative class" "positive class"

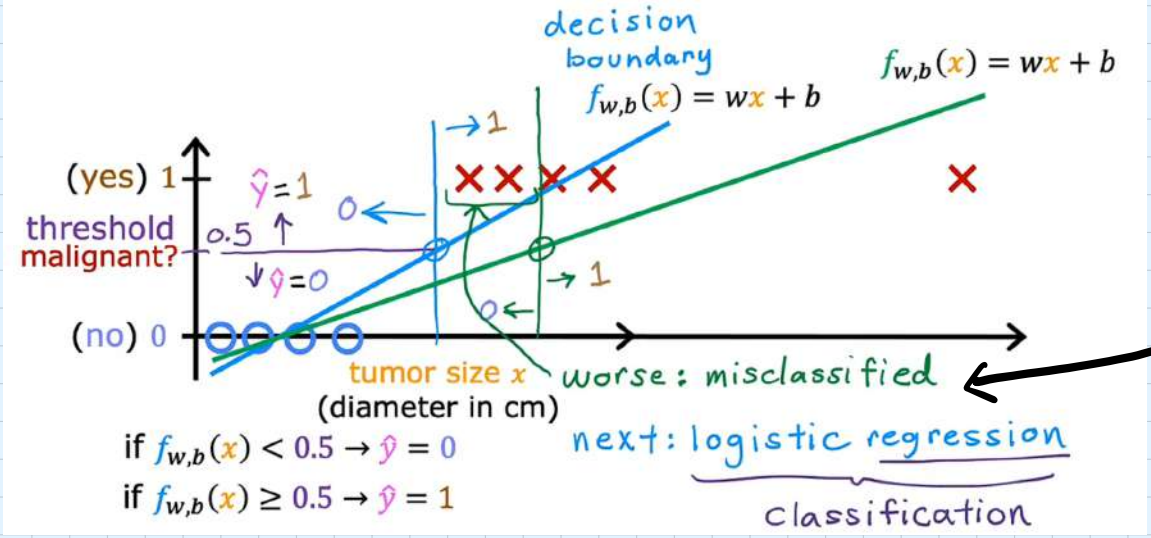
≠ bad, absence

≠ Good, presence

only Two values

↓

called binary Classification



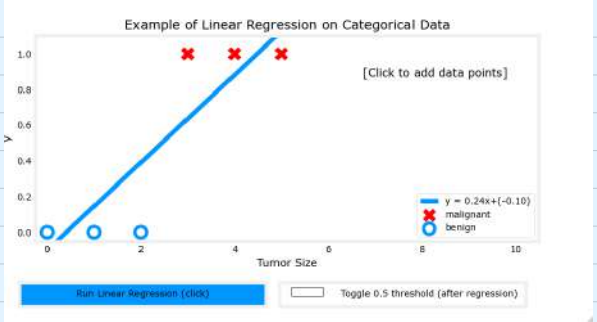
Example of Linear Regression used in Classification

If added new train, decision boundary swift to right

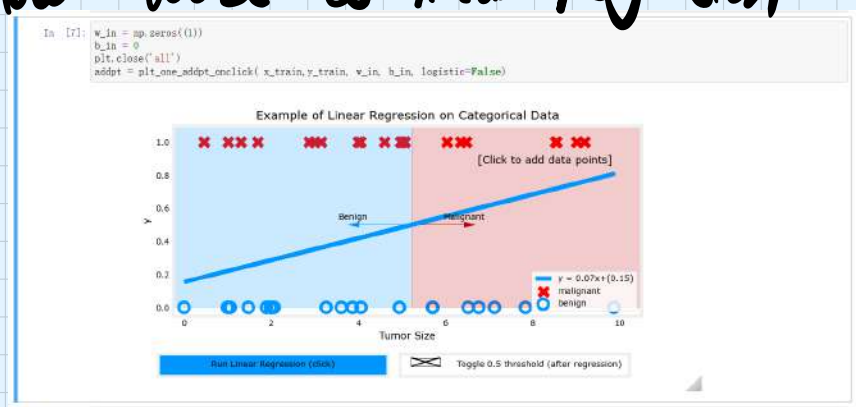
use threshold to implement

↳ still call regression but used in Class...

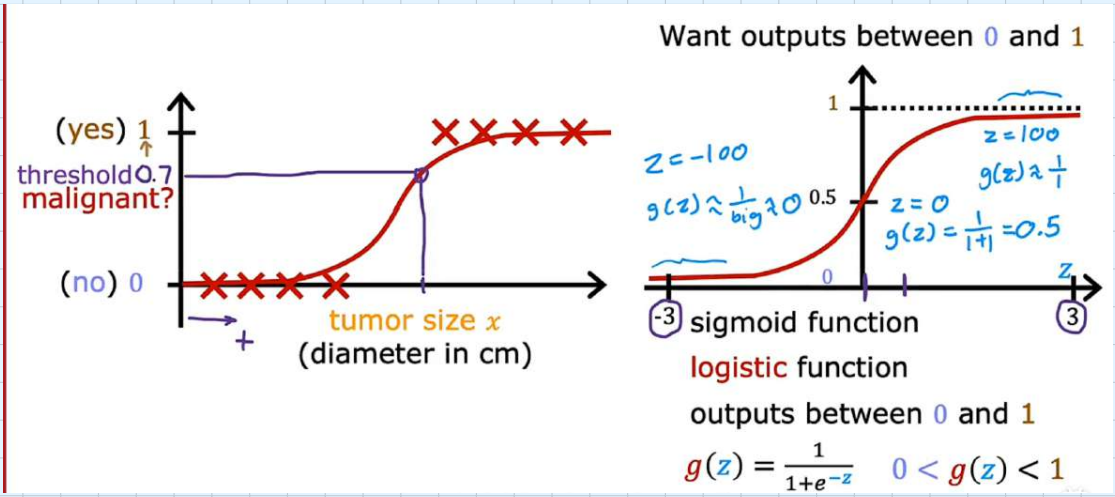
⇝ Optional Lab → show worse of linear Regression



to



Logistic Regression



Sigmoid Function

$$g(z) = \frac{1}{1+e^{-z}}$$

Turn $g(z)$ mathematic model to Logistic Regression

two steps:

$f_{\vec{w},b}(\vec{x})$ 1) $z = \vec{w} \cdot \vec{x} + b$

↓

2) $g(z) = \frac{1}{1+e^{-z}}$

outputs between 0 and 1

$g(z) = \frac{1}{1+e^{-z}}$ $0 < g(z) < 1$

Logistic Regression model

$$\Rightarrow f_{\vec{w},b}(\vec{x}) = g(\overbrace{\vec{w} \cdot \vec{x} + b}^{z}) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x} + b)}}$$

Interpretation of logistic regression output

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w} \cdot \vec{x} + b)}}$$

"probability" that class is 1

P(1)

$f_{\vec{w},b}(\vec{x}) = P(y = 1 | \vec{x}; \vec{w}, b)$

Probability that y is 1, given input $\vec{x}$, parameters $\vec{w}, b$

Example:
x is "tumor size"
y is 0 (not malignant) or 1 (malignant)

$f_{\vec{w},b}(\vec{x}) = 0.7$
70% chance that y is 1

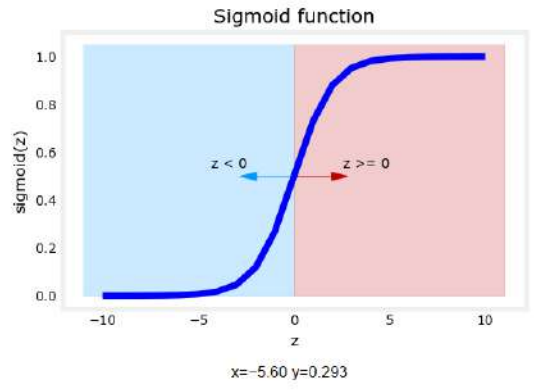
$P(y = 0) + P(y = 1) = 1$

→ Probability

→ Optional Lab
→ Sigmoid function

```
In [6]: # Plot z vs sigmoid(z)
fig, ax = plt.subplots(1, 1, figsize=(5, 3))
ax.plot(z_tmp, y, c="b")

ax.set_title("Sigmoid function")
ax.set_ylabel('sigmoid(z)')
ax.set_xlabel('z')
draw_vthresh(ax, 0)
```

Figure 1

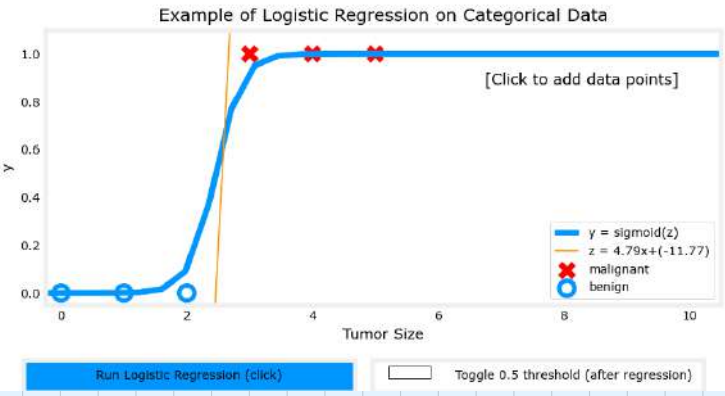


x=-5.60 y=0.293

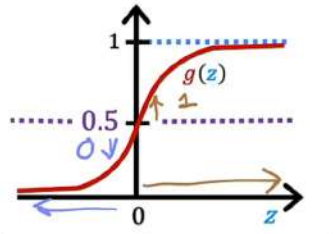
```
In [8]: #Logistic Regression
```

```
In [9]: x_train = np.array([0., 1, 2, 3, 4, 5])
y_train = np.array([0,  0, 0, 1, 1, 1])

w_in = np.zeros((1))
b_in = 0
```

```
In [12]: plt.close('all')
addpt = plt_one_addpt_onclick( x_train,y_train, w_in, b_in, logistic=True)
```



Decision Boundary



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

$z = \vec{w} \cdot \vec{x} + b$

z

$g(z) = \frac{1}{1+e^{-z}}$

$$f_{\vec{w},b}(\vec{x}) = g(\underbrace{\vec{w} \cdot \vec{x} + b}_{z}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

$= P(y = 1 | x; \vec{w}, b)$ 0.7 0.3

0 or 1? threshold

Is $f_{\vec{w},b}(\vec{x}) \geq 0.5$?

Yes: $\hat{y} = 1$ No: $\hat{y} = 0$

When is $f_{\vec{w},b}(\vec{x}) \geq 0.5$?

$g(z) \geq 0.5$

$z \geq 0$

$\vec{w} \cdot \vec{x} + b \geq 0$ $\vec{w} \cdot \vec{x} + b < 0$

$\hat{y} = 1$ $\hat{y} = 0$

By

$f_{\vec{w},b}(\vec{x}) = g(z)$

$= g(w_1 x_1 + w_2 x_2 + b)$

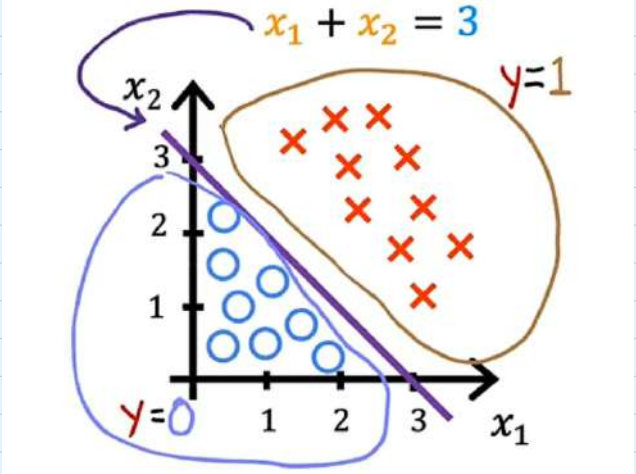
1 1 -3

Decision boundary

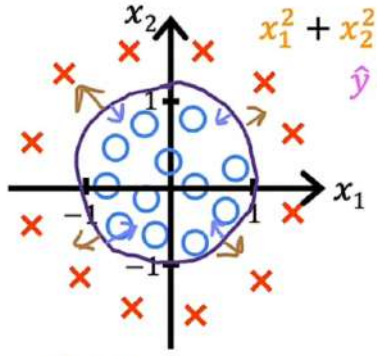
$z = \vec{w} \cdot \vec{x} + b = 0$

$z = x_1 + x_2 - 3 = 0$

$x_1 + x_2 = 3$



Non-linear decision boundaries



$x_1^2 + x_2^2 \geq 1$

$\hat{y} = 1$

$f_{\vec{w},b}(\vec{x}) = g(z) = g(\overbrace{w_1 x_1^2 + w_2 x_2^2 + b}^{z})$

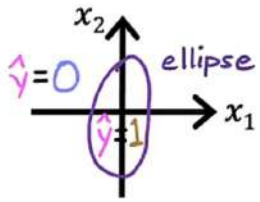
$w_1 = 1,\ w_2 = 1,\ b = -1$

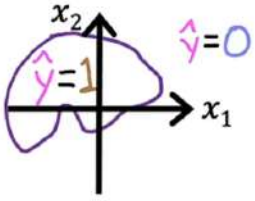
decision boundary $z = x_1^2 + x_2^2 - 1 = 0$

$x_1^2 + x_2^2 = 1$

$x_1^2 + x_2^2 < 1$

$\hat{y} = 0$





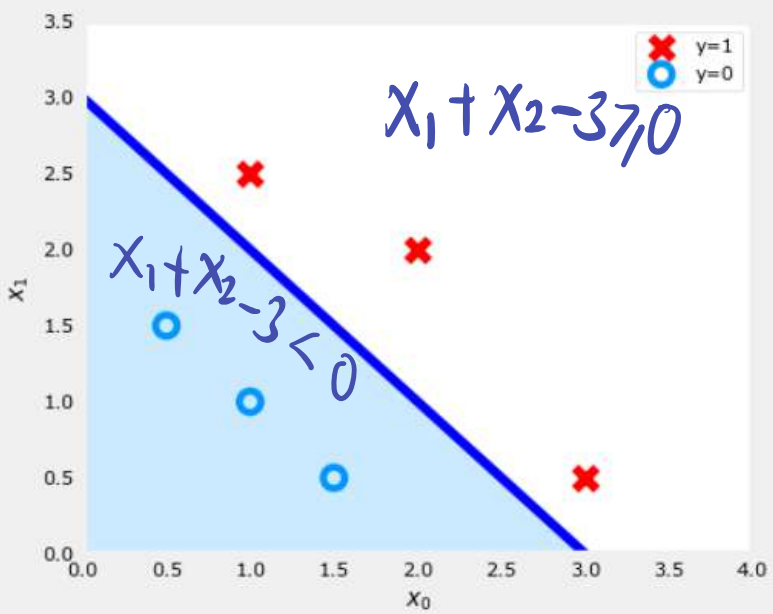
$f_{\vec{w},b}(\vec{x}) = g(z) = g(w_1 x_1 + w_2 x_2 + w_3 x_1^2 + w_4 x_1 x_2 + w_5 x_2^2 + w_6 x_1^3 + \cdots + b)$

Can be complex to fit data

Optional lab

↳ decision boundary

```
# Choose values between 0 and 6
x0 = np.arange(0,6)

x1 = 3 - x0
fig,ax = plt.subplots(1,1,figsize=(5,4))
# Plot the decision boundary
ax.plot(x0,x1, c="b")
ax.axis([0, 4, 0, 3.5])

# Fill the region below the line
ax.fill_between(x0,x1, alpha=0.2)

# Plot the original data
plot_data(X,y,ax)
ax.set_ylabel(r'$x_1$')
ax.set_xlabel(r'$x_0$')
plt.show()
```



Cost Function

target y is 0 or 1

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\vec{x}+b)}}$$

	tumor size (cm) x_1	...	patient's age x_n	malignant? y
$i=1$	10		52	1
	2		73	0
	5		55	0
	12		49	1
$i=m$	...		...	...

$i = 1, \dots, m$ ← training examples

$j = 1, \dots, n$ ← features

target y is 0 or 1

$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

How to choose $\vec{w} = [w_1 \quad w_2 \quad \cdots \quad w_n]$ and b?

Squared error cost

cost

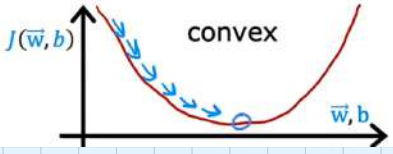
$$J(\vec{w},b) = \frac{1}{m}\sum_{i=1}^{m}\frac{1}{2}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2$$

loss $L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right)$

→ only Change $\frac{1}{2}$ into sum

linear regression

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



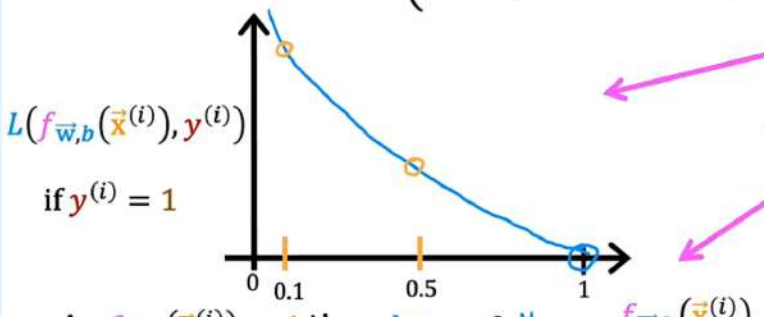
logistic regression

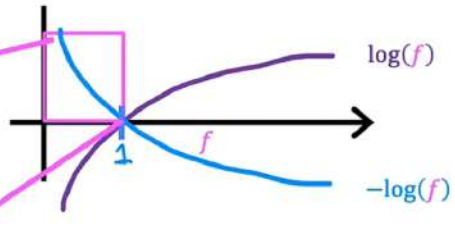
$$f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$$

→ Can make sure convergence to minimum global

Logistic loss function

$$L\left(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$





As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 1$ then loss $\to 0$

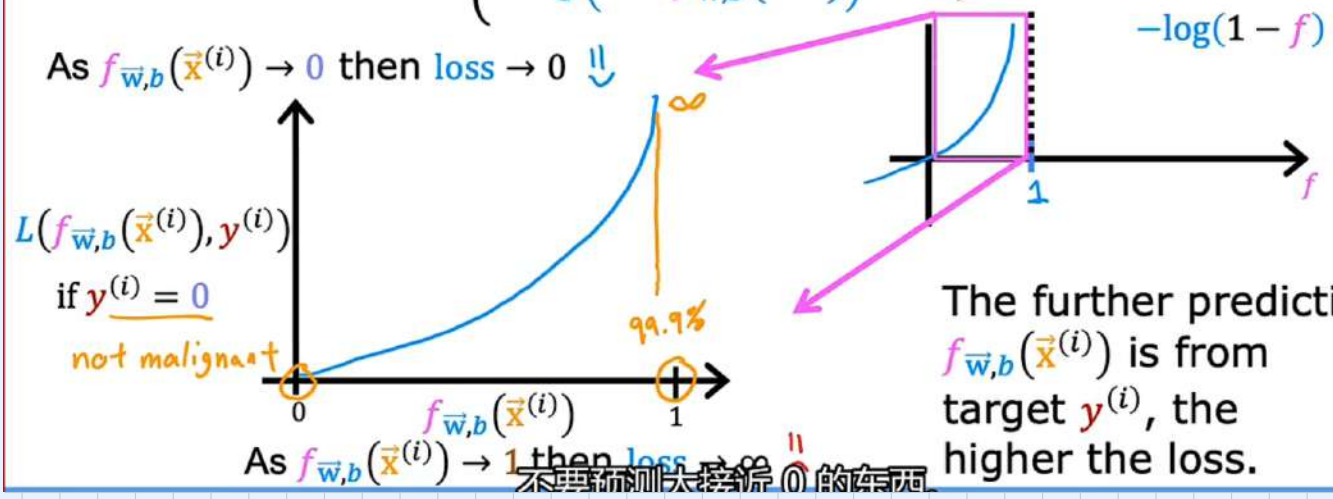
As $f_{\vec{w},b}(\vec{x}^{(i)}) \to 0$ then loss $\to \infty$

Loss is lowest when $f_{\vec{w},b}(\vec{x}^{(i)})$ predicts close to true label $y^{(i)}$.

$y^{(i)} = 1$

$$L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right) = \begin{cases} -\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 0 \end{cases}$$

As $f_{\vec{w},b}\left(\vec{x}^{(i)}\right) \to 0$ then loss $\to 0$



As $f_{\vec{w},b}\left(\vec{x}^{(i)}\right) \to 1$ then loss $\to \infty$

The further prediction $f_{\vec{w},b}\left(\vec{x}^{(i)}\right)$ is from target $y^{(i)}$, the higher the loss.

Cost

$$J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} L\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right), y^{(i)}\right)$$

↳ loss

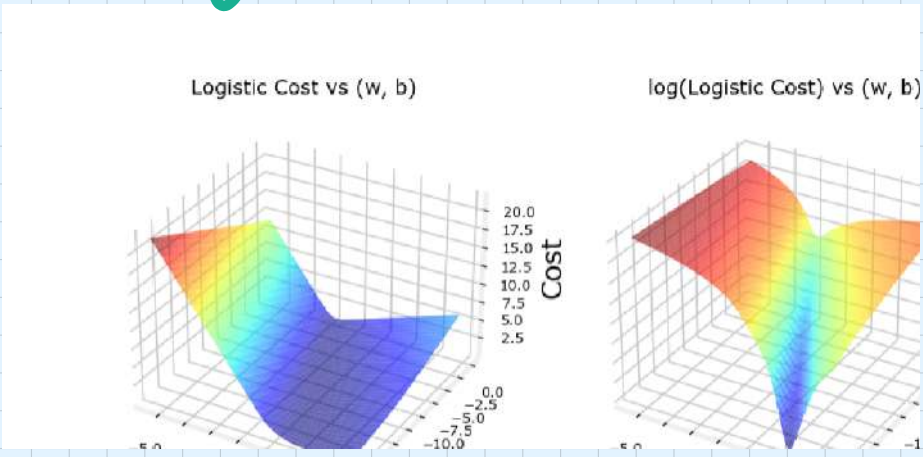
Convex ⇝ can reach a global minimum

$$= \begin{cases} -\log\left(f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}\left(\vec{x}^{(i)}\right)\right) & \text{if } y^{(i)} = 0 \end{cases}$$

⟶ Find w, b that minimize cost J

Optional Lab → Logistic Loss

- determined a squared error loss function is not suitable for classification tasks
- developed and examined the logistic loss function which **is** suitable for classification tasks.



Simplified loss function → write w/ hart

From above to

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -y^{(i)} \log(f_{\vec{w},b}(\vec{x}^{(i)})) - (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(\vec{x}^{(i)}))$$

Simplified loss function

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = \begin{cases} -\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 1 \\ -\log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) & \text{if } y^{(i)} = 0 \end{cases}$$

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)$$

($y^{(i)} = 0$ → first term $\to 0$; ($1 - y^{(i)}$) → ($1 - 0$))

if $y^{(i)} = 1$:

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -1 \log(f(\vec{x}))$$

if $y^{(i)} = 0$:

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -\log(1 - f_{\vec{w},b}(\vec{x}^{(i)})) - (1-0)\log(1 - f(\vec{x}))$$

loss

$$L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) = -y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) - (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right)$$

convex (single global minimum)

cost

$$J(\vec{w}, b) = \frac{1}{m} \sum_{i=1}^{m} \left[L(f_{\vec{w},b}(\vec{x}^{(i)}), y^{(i)}) \right]$$

$$= -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1 - y^{(i)}) \log\left(1 - f_{\vec{w},b}(\vec{x}^{(i)})\right) \right]$$

$$\therefore J(\vec{w},b) = -\frac{1}{m} \sum_{i=1}^{m} \left[y^{(i)} \log(f_{\vec{w},b}(\vec{x}^{(i)}) + (1 - y^{(i)}) \log(1 - f_{\vec{w},b}(x^{(i)})) \right]$$

Optional Lab ~ implement Loss Function

```
def compute_cost_logistic(X, y, w, b):
    """
    Computes cost

    Args:
      X (ndarray (m,n)): Data, m examples with n features
      y (ndarray (m,)) : target values
      w (ndarray (n,)) : model parameters
      b (scalar)       : model parameter

    Returns:
      cost (scalar): cost
    """

    m = X.shape[0]
    cost = 0.0
    for i in range(m):
        z_i = np.dot(X[i],w) + b
        f_wb_i = sigmoid(z_i)
        cost +=  -y[i]*np.log(f_wb_i) - (1-y[i])*np.log(1-f_wb_i)

    cost = cost / m
    return cost
```

$g(z_i)$

$z_i = wx + b$

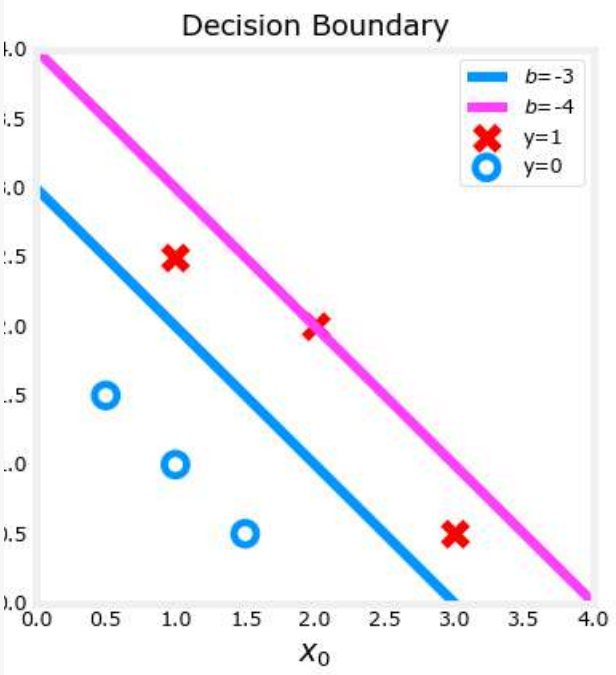
$f_{wb,i} = \frac{1}{1+e^{-z_i}}$

$$\therefore J(\vec{w},b) = -\frac{1}{m}\sum\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(x^{(i)})\right)\right]$$

```
w_tmp = np.array([1,1])
b_tmp = -3
print(compute_cost_logistic(X_train, y_train, w_tmp, b_tmp))
```

```
0.36686678640551745
```

```
import matplotlib.pyplot as plt

# Choose values between 0 and 6
x0 = np.arange(0,6)

# Plot the two decision boundaries
x1 = 3 - x0
x1_other = 4 - x0

fig,ax = plt.subplots(1, 1, figsize=(4,4))
# Plot the decision boundary
ax.plot(x0,x1, c=dlc["dlblue"], label="$b$=-3")
ax.plot(x0,x1_other, c=dlc["dlmagenta"], label="$b$=-4")
ax.axis([0, 4, 0, 4])

# Plot the original data
plot_data(X_train,y_train,ax)
ax.axis([0, 4, 0, 4])
ax.set_ylabel('$x_1$', fontsize=12)
ax.set_xlabel('$x_0$', fontsize=12)
plt.legend(loc="upper right")
plt.title("Decision Boundary")
plt.show()
```



Gradient Descent Implementation

Training logistic regression

Find $\vec{w}, b$

Given new $\vec{x}$, output $f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{-(\vec{w}\cdot\vec{x}+b)}}$

$$P(y = 1|\vec{x}; \vec{w}, b)$$

Gradient descent

cost

$$J(\vec{w}, b) = -\frac{1}{m}\sum_{i=1}^{m}\left[y^{(i)}\log\left(f_{\vec{w},b}(\vec{x}^{(i)})\right) + (1-y^{(i)})\log\left(1-f_{\vec{w},b}(\vec{x}^{(i)})\right)\right]$$

repeat {

$j = 1 \ldots n$

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

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

} simultaneous updates

Gradient descent for logistic regression

repeat { looks like linear regression!

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})x_j^{(i)}\right]$$

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

} simultaneous updates

↗ can use same

Same concepts:

- Monitor gradient descent (learning curve)
- Vectorized implementation
- Feature scaling

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

Logistic regression $f_{\vec{w},b}(\vec{x}) = \frac{1}{1+e^{(-\vec{w}\cdot\vec{x}+b)}}$

Looks similiar but different in $f_{\vec{w},b}(\vec{x}^{(i)})$

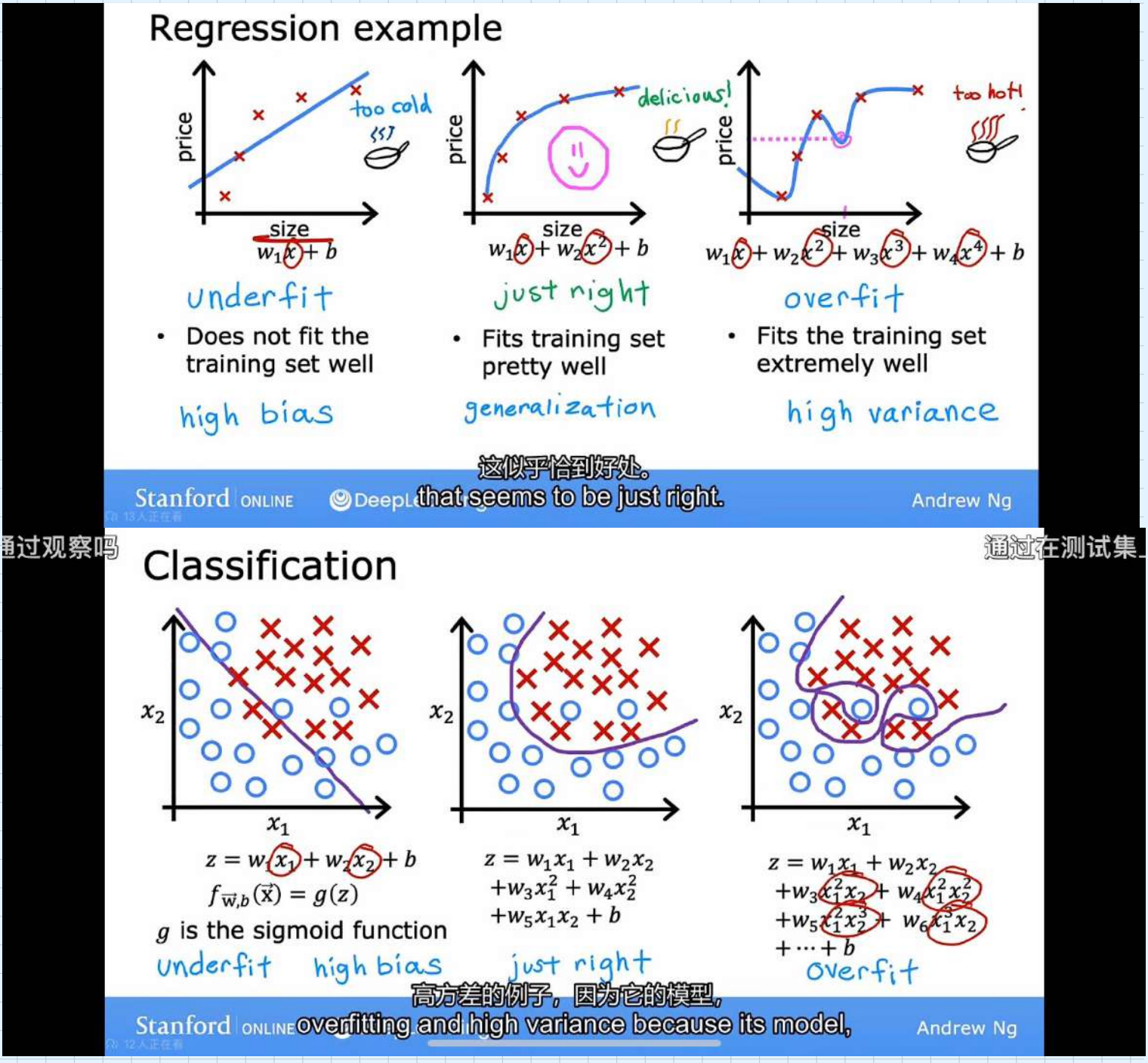
Optional Lab → Logistic Regression Gradient Descent of Implementation

找不到

Optional Lab : use sklearn implement

看Jupyter

The problem of Overfitting



通过观察吗

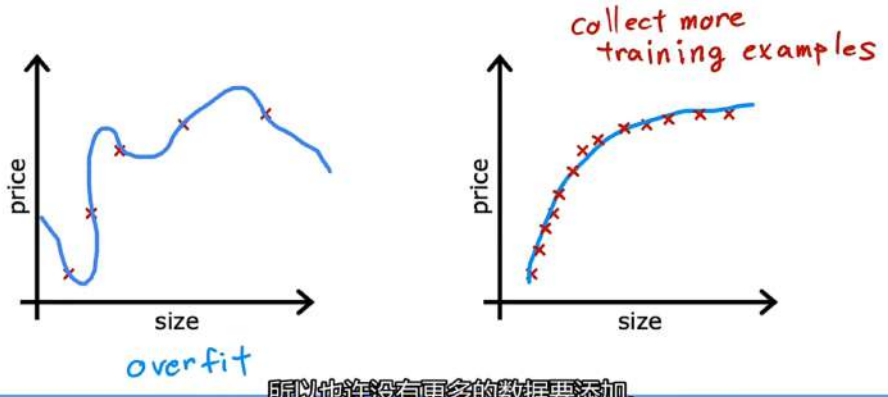
通过在测试集

We need Just Right

Solve overfitting

(1)

Collect more training examples



所以也许没有更多的数据要添加。
so maybe there just isn't more data to be add.

Stanford ONLINE Andrew Ng

(2)

Select features to include/exclude

size	bedrooms	floors	age	avg income	...	distance to coffee shop	price
x_1	x_2	x_3	x_4	x_5		x_{100}	y

all features
+
insufficient data
↓
overfit

selected features
size
bedrooms
age
just right
feature selection

disadvantage
↓
useful features could be lost

稍后在课程 2
Later in Course 2,

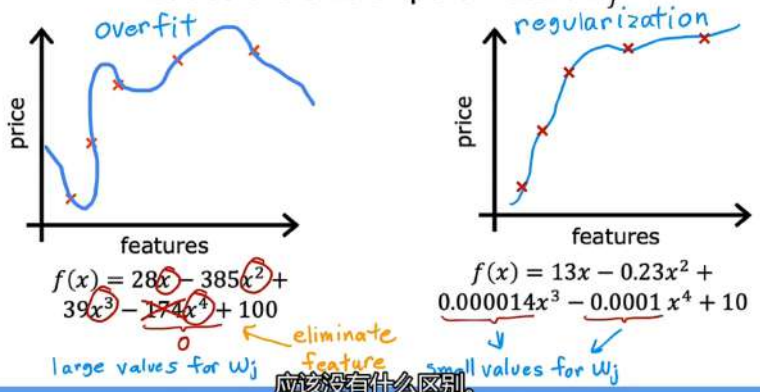
Stanford ONLINE DeepLearni Andrew Ng

(3) Regularization 正则化 (降低 W

→ gently reduce

Regularization

Reduce the size of parameters w_j



应该没有什么区别。
whether you also regularize b or not.

Stanford ONLINE Andrew Ng

Addressing overfitting

Options

1. Collect more data

2. Select features
   - Feature selection in course 2

3. Reduce size of parameters
   - "Regularization" next videos!

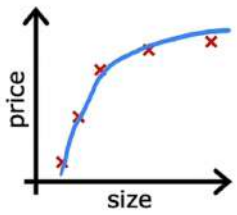
个非常有用的技术，
for training learning algorithms,

Stanford ONLINE Andrew Ng

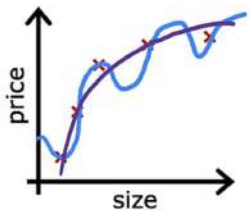
Optional Lab ~> Overfitting

Regularization

Intuition



$w_1x + w_2x^2 + b$



$w_1x + w_2x^2 + \underbrace{w_3x^3}_{\approx 0} + \underbrace{w_4x^4}_{\approx 0} + b$

make w_3, w_4 really small (≈ 0)

$$\min_{\vec{w},b} \frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + 1000\,w_3^2 + 1000\,w_4^2$$

0.002

得到这个每周二次函数相比，它更适合数据
the parameters could be large and you
end up with this weekly quadratic function

Stanford ONLINE Andrew Ng

Regularization

small values $w_1, w_2, \cdots, w_n, b$

simpler model
less likely to overfit

$w_3 \approx 0$
$w_4 \approx 0$

size	bedrooms	floors	age	avg income	...	distance to coffee shop	price
x_1	x_2	x_3	x_4	x_5		x_{100}	y

$n = 100$

$w_1, w_1, w_2, \cdots, w_{100}, b$

n features

regularization term

$$J(\vec{w},b) = \frac{1}{2m}\left[\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2 + \frac{\lambda}{2m}b^2\right]$$

"lambda"

regularization parameter

$\lambda > 0$

can include or exclude b

只有参数 w 而不是参数 b。
only the parameters w rather than the parameter b.

Stanford ONLINE Andrew Ng

Regularization

$$\min_{\vec{w},b} J(\vec{w},b) = \min_{\vec{w},b}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2\right]$$

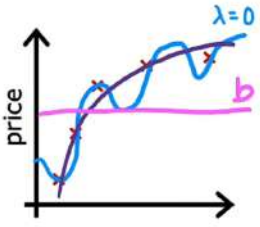
mean squared error

regularization term

fit data

Keep w_j small

λ balances both goals



choose $\lambda = 10^{10}$

$$f_{\vec{w},b}(\vec{x}) = \underset{\approx 0}{w_1 x} + \underset{\approx 0}{w_2 x^2} + \underset{\approx 0}{w_3 x^3} + \underset{\approx 0}{w_4 x^4} + b$$

$$f(x) = b$$

choose λ

保留所有这些功能，但功能看起来像这样。

Stan keeping all of these features, but with a function that looks like this. ew Ng

$$\min_{\vec{w},b} J(\vec{w},b) = \frac{1}{2n}\left[\sum_{i=1}^{n}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \lambda\sum_{j=1}^{n} w_j + \lambda b^2\right]$$

$\min_{\vec{w},b}$

Regularization in Linear Regression

Regularized linear regression

$$\min_{\vec{w},b} J(\vec{w},b) = \min_{\vec{w},b}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2\right]$$

Gradient descent

repeat {

$$w_j = w_j - \alpha\frac{\partial}{\partial w_j}J(\vec{w},b) \qquad j = 1,\ldots,n$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)} + \frac{\lambda}{m}w_j$$

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

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

don't have to regularize b

} simultaneous update

左边的表达式写出梯

Stanford ONLINE the expression on the left to write out Andrew Ng

Implementing gradient descent

repeat {

$$w_j = w_j - \alpha\left[\frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m}w_j\right]$$

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

} simultaneous update

How regularization work

$\frac{\lambda}{m}$

$0.01 \frac{1}{50} = 0.0002$

$w_j(1-0.0002)$

0.9998

$$w_j = \underbrace{1w_j - \alpha\frac{\lambda}{m} w_j}_{w_j\left(1-\alpha\frac{\lambda}{m}\right)} - \underbrace{\alpha\frac{1}{m}\sum_{i=1}^{m}\left(f_{w,b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}}_{\text{usual update}}$$

simultaneous update $j = 1 \dots n$

shrink w_j

↳ shrink by times number less than 1

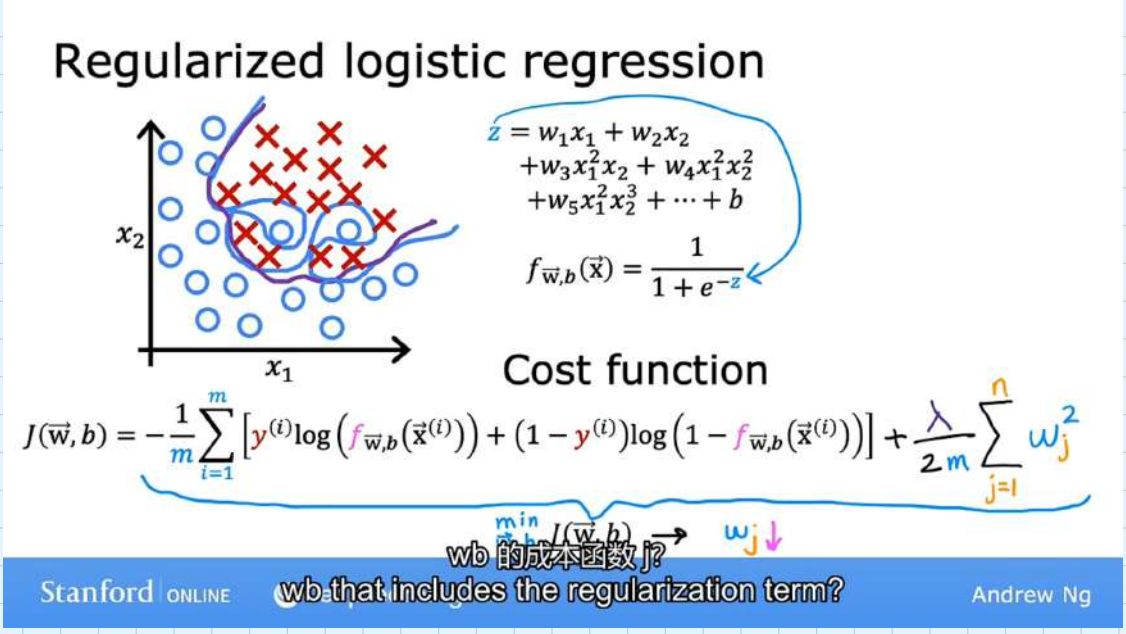
推导 ~

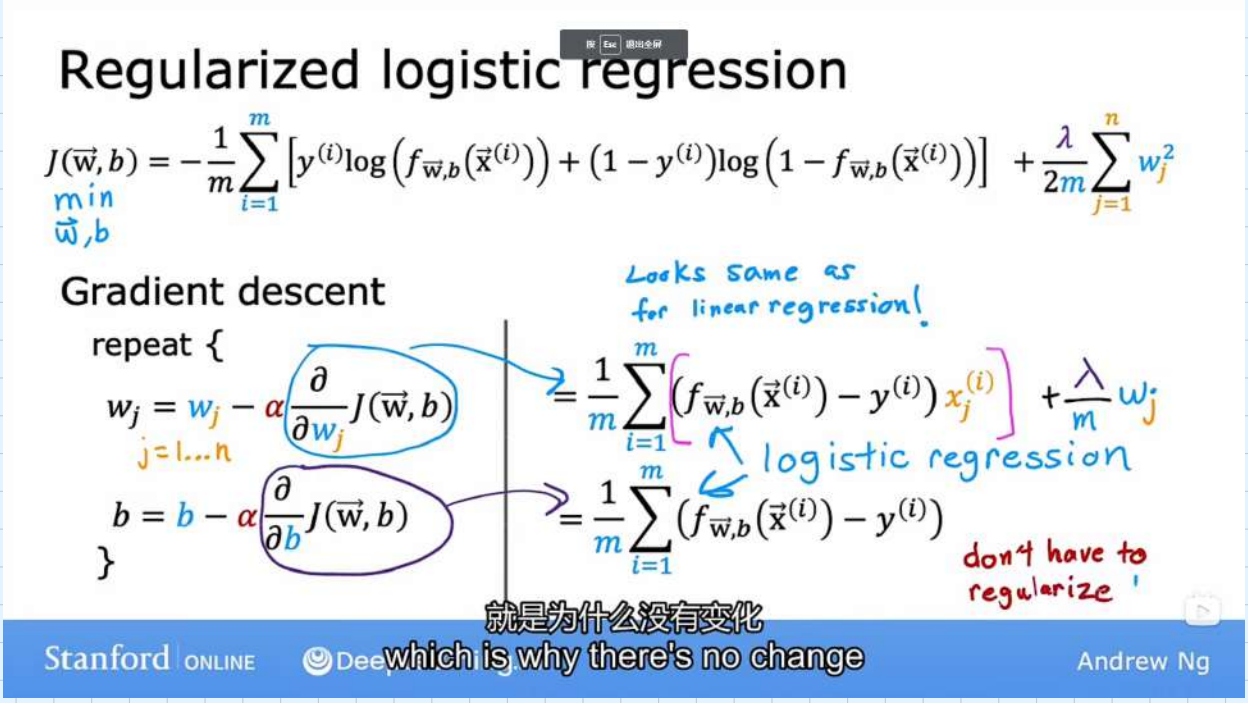
$$\frac{\partial}{\partial w_j} J(\vec{w},b) = \frac{\partial}{\partial w_j}\left[\frac{1}{2m}\sum_{i=1}^{m}\left(f(\vec{x}^{(i)}) - y^{(i)}\right)^2 + \frac{\lambda}{2m}\sum_{j=1}^{n} w_j^2\right]$$

$$= \frac{1}{2m}\sum_{i=1}^{m}\left[\left(\vec{w}\cdot\vec{x}^{(i)} + b - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{2m} 2w_j \quad \text{No } \sum_{j=1}$$

$$= \frac{1}{m}\sum_{i=1}^{m}\left[\left(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)}\right)x_j^{(i)}\right] + \frac{\lambda}{m} w_j$$

Regularization in Logistic Regression





Optional Lab ~> Regularization

See Jupyter