#fun/ml/andrewng

Intro to ML

↳ Science of getting computers to learn w/o being specifically programmed.

2 main types of ML algo

- Supervised (course 1, 2)
- Unsupervised (course 3)
- Recommender systems
- Reinforcement learning

Supervised Learning

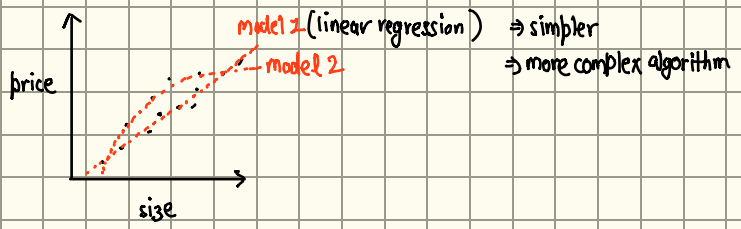
↳ Algorithms that learn input → output mapping

↳ look at a set of examples (labled) & learn to make a good guess

Eg: email spam → 0 or 1
english → spanish
house size → price

Regression Example : House size vs price

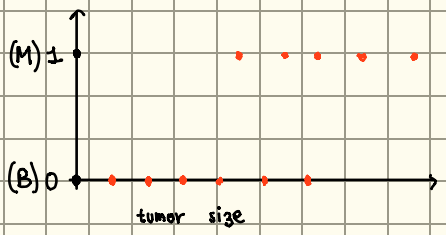
↳ try to predict from only many possible output



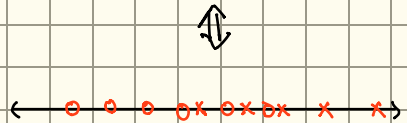
⇒ simpler
⇒ more complex algorithm

Classification Example : breast cancer → malignant (1), benign (0)

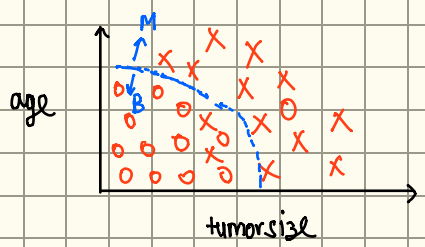
↳ classify into a fixed/small no of possible outcomes



algo will say more size ⇒ more likely to be M



can be done in 1d ∵ only 1 input

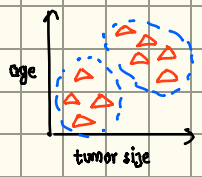


⇐ 2d example ∵ 2 input

Unsupervised learning

↳ No output labels unlike supervised, algorithm has to come up w patterns/structure

Eg:



model comes up w interesting patterns
→ in this case, clusters
↳ We find clusters using clustering algorithm

Eg: Google news use clustering algo to group similar news.

More Eg: anomaly detection : detect unusual data points
: dimentionality reduction : compressing data w/o much info loss
↳ find patterns & compress (remember primagen video)

Intro Over

Linear Regression Model

↳ Fitting a straight line to data

The model predicts
1250 sq ft ⇒ $220K

∞ly many possible outcomes ∴ regression

#Terminologies

1. Training set : dataset fed to model
2. Input variable : x : feature
3. Output variable : y : target
4. Total number of training examples (no of rows in dataset table) = m
5. Single training example : $(x^{(i)}, y^{(i)})$

Process Of Supervised Learning

features, targets → training set → fed to → learning algo → produces → hypotheses function "f"

x → hypotheses function "f" → $\hat{y}$

gives out prediction for y when some x is given

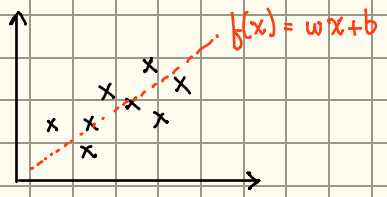
- $\hat{y}$ is estimate for y

The Function

Let's take a simple linear function ⇒ $f_{w,b}(x) = wx + b$

weight / slope — bias / intercept

values chosen for w & b determine value of $\hat{y}$



This is Linear Regression with one variable
(one feature)

C1 W1 Lab02

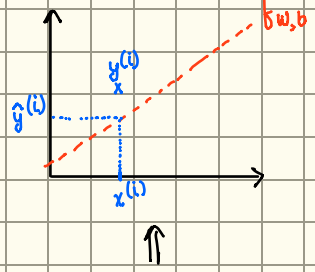
numpy array: `x_train = np.array([1.0, 2.0])`
`y_train = np.array([300.0, 500.0])`

no of training ex = size of array = shape of array [0] = `x_train.shape[0]` = `len(x_train)`

The cost func

↳ Tells us how well model is doing Model: $f(x) = wx + b$ (w & b are parameters)

we want to benchmark the chosen values w & b that the line fits the data well



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

here, error = $(\hat{y} - y)$ for $x^{(i)}$, cost func = errors for all $x^{(i)}$

COST FUNCTION (squared error): $J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} (\hat{y}^{(i)} - y^{(i)})^2$

squared error cost func is most commonly used one for regression

to make later work neater; so that cost func doesn't grow with training data size, $\frac{1}{2m}$ normalizes CF

$$\star \quad J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(\hat{y}^{(i)} - y^{(i)}\right)^2 = \frac{1}{2m} \sum_{i=1}^{m} \left(f(x^{(i)}) - y^{(i)}\right)^2$$

Intuition of $J(w,b)$

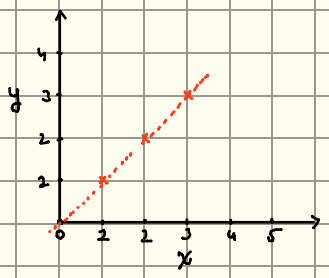
$J(w,b)$ is calculating total error in our model
∴ goal: minimize $J(w,b)$

Take simplified model: $f(x) = wx$

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

goal = find w, that minimizes $J(w)$

$f(x)$ as w varies & Corresponding $J(w)$ value

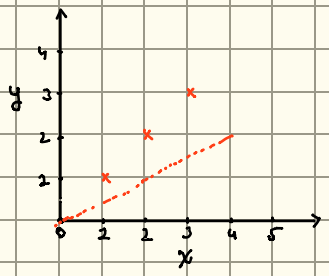


$w = 1$

$$J(w) = \frac{1}{2m}\sum_{i=1}^{m}\left(f_w(x^{(i)}) - y^{(i)}\right)^2 = \frac{1}{2m}\sum_{i=1}^{m}\left(\underbrace{wx^{(i)}}_{\hat{y}^{(i)}} - y^{(i)}\right)^2 = \frac{1}{2m}(0^2 + 0^2 + \cdots \text{ m times}) = 0$$

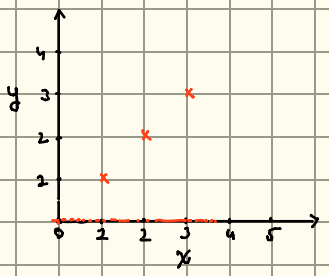
$\hat{y}^{(i)} = y^{(i)}$

0



$w = 0.5$

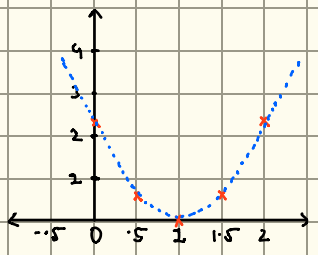
$$J(w) = \frac{1}{2m}\left((0.5-1)^2 + (1-2)^2 + (1.5-3)^2\right) = \frac{3.5}{6} = 0.58$$



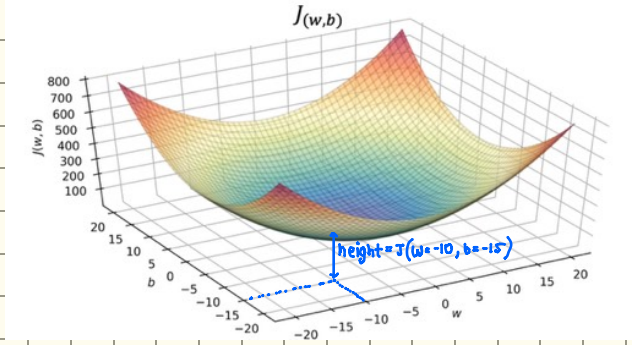
$w = 0$

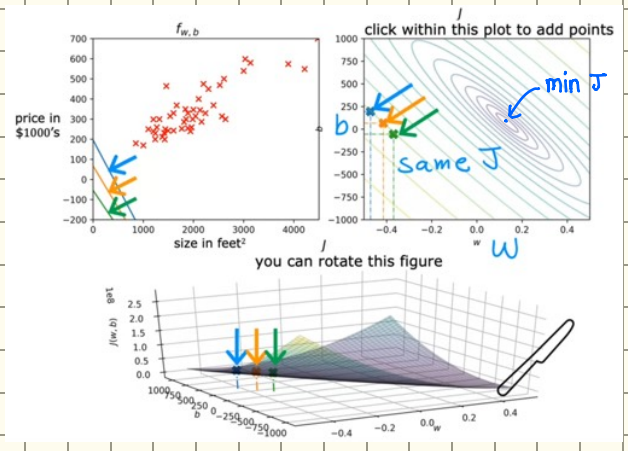
$$J(w) = \frac{1}{2m}(1^2 + 2^2 + 3^2) = 2.3$$

Plot of $J(w)$ vs w



when $b \neq 0$, varying both w & b change J





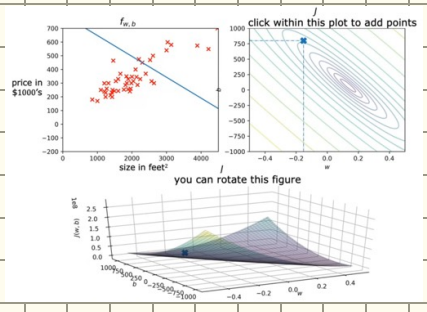
3d

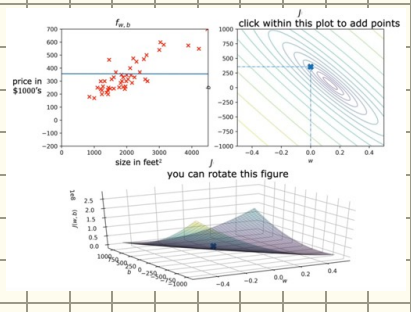
↓

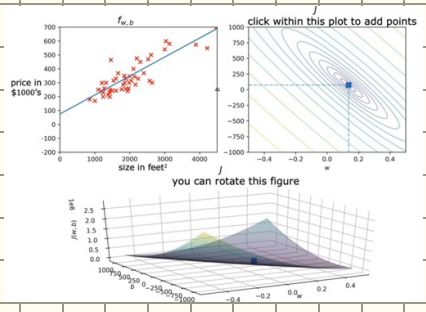
Contour 2d

flatten down 3d

- Each ellipses show the points on the 3d surface which are at the same height i.e. same J







C1. W1. Lab 03

```
x_train = np.array([1.0, 2.0])   #size
y_train = np.array([300.0, 500.0])   #price
```

```
def compute_cost(x, y, w, b):
    """
    Computes the cost function for linear regression.

    Args:
      x (ndarray (m,)): Data, m examples
      y (ndarray (m,)): target values
      w,b (scalar)    : model parameters

    Returns
        total_cost (float): The cost of using w,b as the parameters for linear regression
               to fit the data points in x and y
    """
    # number of training examples
    m = x.shape[0]

    cost_sum = 0
    for i in range(m):
        f_wb = w * x[i] + b
        cost = (f_wb - y[i]) ** 2
        cost_sum = cost_sum + cost
    total_cost = (1 / (2 * m)) * cost_sum

    return total_cost
```

takes i training data & w,b from model

m = no of data points

Loop

Find $\hat{y}$

squared error

add for total sq. error

normalize

Back to the Goal: how to choose w s.t. $J(w)$ is min

Gradient descent

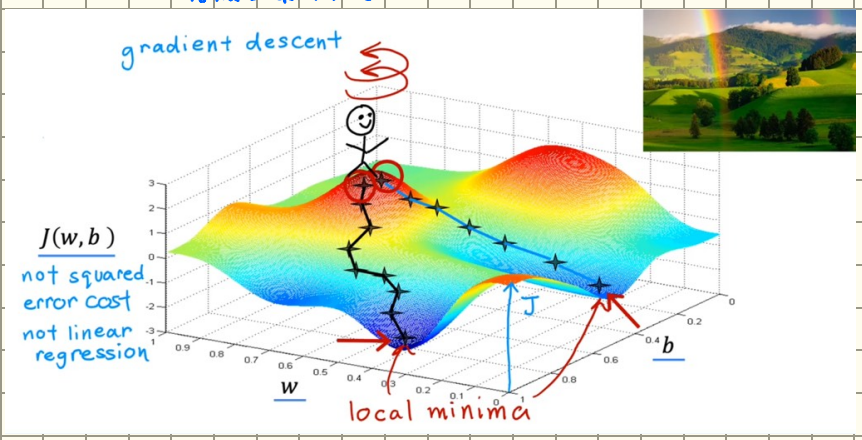
Goal: $\min_{w,b} J(w,b)$ using GD

GD can be used w any function/model w many parameters. It can minimize $J(w_1, w_2, w_3 \ldots\ldots w_n, b)$

Process

1] Start w some w, b — in linear regression starting values won't affect much, ∴ let $w=0$ & $b=0$

2] Keep changing w, b to reduce $J(w,b)$ until we settle at / near a min
↳ to the steepest side



(J) For complex functions, its possible to have >1 min

we could also land at local minima & not global min

↳ To counter this, we should run with different w, b again

#Changing w, b

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

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

α = learning rate = how big step is

$\frac{\partial}{\partial b} J(w,b)$ ↳ direction we want to take step in (direc of steepest slope)

Algo in code

↳ from current w, b

↳ $w_{new} = w - \alpha \frac{\partial}{\partial w} J(w,b)$; $b_{new} = b - \alpha \frac{\partial}{\partial b} J(w,b)$

↳ $w = w_{new}$; $b = b_{new}$

cannot do $w = w - \alpha \frac{\partial}{\partial w} J(w,b)$ ∵ then b will be calculated from a new w (w_{new}, b)

Gradient descent Intuition

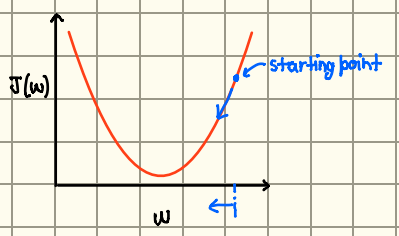
α = how big of a step

$\frac{\partial}{\partial w} J(w,b)$ = in what direction

Simplify (F for now, ignore b ⇒ $J(w)$

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

makes w go down the slope



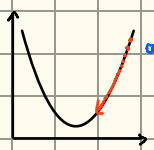
slope +ve ∴ w - +ve number ⇒ $w\downarrow$ (case at the starting point)

slope -ve ∴ w - -ve number ⇒ $w\uparrow$

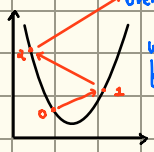
Choosing α

↳ has huge effect on efficiency

If α ↓



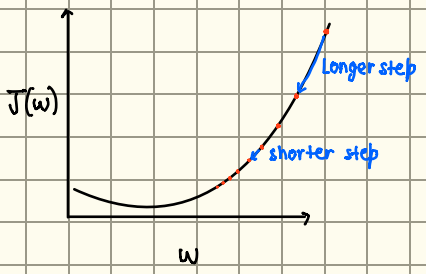
If α ↑



as gradient descent goes on, the length of steps keep getting smaller ∵ slope $\left(\frac{d}{dw} J(w)\right)$ keeps getting smaller

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

smaller number gets subtracted each iteration



Intuition over

Gradient descent for Linear Regression

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

Cost Function : $J(w,b) = \frac{1}{2m} \sum_{i=1}^{m} \left(f(x^{(i)}) - y^{(i)} \right)^2$

Gradient descent :

repeat until convergence {

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

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

$w = w_{new}$; $b = b_{new}$;

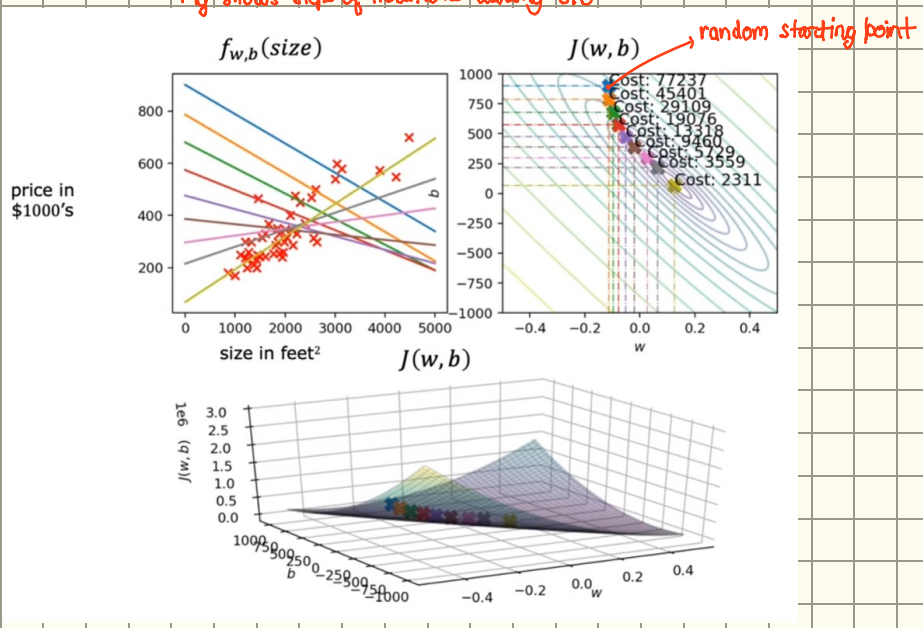
}

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

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

$\because f(x^{(i)}) = wx^{(i)} + b$

Fig shows steps of iterations during GD



This GD is batch GD ∵ we look at all points of training set during each iteration here
There are also GD algos that look at smaller subset

C1 W1 Lab04 GD for LR

3 functions needed

- compute_gradient
- compute_cost
- gradient_descent

$\frac{\partial}{\partial w} J(w,b) = \frac{1}{m} \sum_{i=0}^{m-1} \left(f(x^{(i)}) - y^{(i)}\right) x^{(i)}$; $\frac{\partial}{\partial b} J(w,b) = \frac{1}{m} \sum_{i=0}^{m-1} \left(f(x^{(i)}) - y^{(i)}\right)$

$J(w,b) = \frac{1}{2m} \sum_{i=0}^{m-1} \left(f(x^{(i)}) - y^{(i)}\right)^2$

utilize the above two

both called for each iteration of GD func

```
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
```

$\frac{\partial J}{\partial w}$ & $\frac{\partial J}{\partial b}$ finding

```
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
```

GD

training set initial w & b learning rate when to stop helper functions

```
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]
      """

    # 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
```

find J_{min} w & b history of parameters [w,b]

AKA: multiple LR
not multivariate LR

Linear Reg w multiple input variables / features

Many features like size (x_1), no. of bedrooms (x_2), no. of floors (x_3), age of home (x_4) to determine price (y)

x_j = j^{th} feature
n = total number of features (4 here)
$\vec{x}^{(i)}$ = features of i^{th} training example, a row vector with 4 elements, $\vec{x}^{(2)} = [\dots, \dots, \dots, \dots]$
$x_j^{(i)}$ = feature j of i^{th} example, a number

LR model : $f_{\vec{w},b}(\vec{x}) = w_1x_1 + w_2x_2 + w_3x_3 + w_4x_4 + b$

Eg: $f_{\vec{w},b}(\vec{x}) = 0.1x_1 + 4x_2 + 10x_3 + -2x_4 + 80$
80 kinda like base price of house

Let $\vec{w} = [w_1, w_2, w_3 \dots w_n]$ } parameters of model
b = single number
$\vec{x} = [x_1, x_2, x_3 \dots x_n]$

Short model : $f_{\vec{w},b} = \vec{w} \cdot \vec{x} + b$
dot product

Vectorization : code shorter & more efficient

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

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

runs in $||^{el}$ on computer

C1 W2 Lab01 : Python, numpy, Vectorization

Vector (numpy array) creation :
- `a = np.zeroes(4)`] same `[0. 0. 0. 0.]`
- `a = np.zeroes((4,1))`
- `a = np.random.random_sample(4)` rand b/w 0 to 1
- `a = np.arange(4.)` `a = [0. 1. 2. 3.]`
- `a = np.array([5,4,3,2])` `a = [5 4 3 2]` int
- `a = np.array([5.,4,3,2])` `a = [5. 4. 3. 2.]` float

} by default all float ∴ "." at the end

there is indexing, slicing, vk that stuff

single vector ops :
```
b = np.sum(a)
b = np.mean(a)
b = -a
b = a**2   #will do **2 on every element of a
```

Vector Vector ops

`c = a+b` , `c = a*b` # a & b have to be same shape vectors

dot product : $a \cdot b = \begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ a_3 \end{bmatrix} \begin{bmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{bmatrix} = a_0b_0 + a_1b_1 + a_2b_2 + a_3b_3 =$ scalar value

`c = np.dot(a,b)`

#delete arrays from memory : `del(a)`, `del(b)`

```
np.random.seed(1)
a = np.random.rand(10000000)  # very large arrays
b = np.random.rand(10000000)

tic = time.time()  # capture start time
c = np.dot(a, b)
toc = time.time()  # capture end time

print(f"np.dot(a, b) =  {c:.4f}")
print(f"Vectorized version duration: {1000*(toc-tic):.4f} ms ")

tic = time.time()  # capture start time
c = my_dot(a,b)
toc = time.time()  # capture end time

print(f"my_dot(a, b) =  {c:.4f}")
print(f"loop version duration: {1000*(toc-tic):.4f} ms ")

del(a);del(b)  #remove these big arrays from memory
```

```
np.dot(a, b) =  2501072.5817
Vectorized version duration: 170.3112 ms
my_dot(a, b) =  2501072.5817
loop version duration: 9538.0843 ms
```

Matrices : m x n

m: training examples, n: features

: creation :
- `a = np.zeroes((1,5))` `[[0. 0. 0. 0. 0.]]`
- `a = np.zeroes((2,1))` `[[0.]` `[0.]]` 2x1
- `a = np.array([[5],[4],[3]])` `[[5]` `[4]` `[3]]` 3x1 = `np.array([[5], [4], [3]])`
- `a = np.arange(6).reshape(-1, 2)`

↑ dont matter ↑ 2 columns I want — could've done `.reshape(3,2)` ∵ 6 elements there

There is indexing & slicing also go look at ipynb if needed

4.4.2 Slicing

Slicing creates an array of indices using a set of three values (`start:stop:step`). A subset of values is also valid. Its use is best explained by example:

```
#vector 2-D slicing operations
a = np.arange(20).reshape(-1, 10)
print(f"a = \n{a}")

#access 5 consecutive elements (start:stop:step)
print("a[0, 2:7:1] = ", a[0, 2:7:1], ",  a[0, 2:7:1].shape =", a[0, 2:7:1].shape, "a 1-D

#access 5 consecutive elements (start:stop:step) in two rows
print("a[:, 2:7:1] = \n", a[:, 2:7:1], ",  a[:, 2:7:1].shape =", a[:, 2:7:1].shape, "a 2-

# access all elements
print("a[:,:] = \n", a[:,:], ",  a[:,:].shape =", a[:,:].shape)

# access all elements in one row (very common usage)
print("a[1,:] = ", a[1,:], ",  a[1,:].shape =", a[1,:].shape, "a 1-D array")
# same as
print("a[1]   = ", a[1],   ",  a[1].shape   =", a[1].shape, "a 1-D array")
```

Gradient Descent (multiple variable) for the model $f_{\vec{w},b}(\vec{x}) = \vec{w}\vec{x} + b$ to minimize func $J(\vec{w}, b)$

Notation

General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
$\mathbf{A}$	matrix, bold capital	
Regression	↳ m (no of points) x n (no of features)	
$\mathbf{X}$	training example matrix	`X_train`
$\mathbf{y}$	training example targets vector ↳ length m	`y_train`
$\mathbf{x}^{(i)}, y^{(i)}$	i_{th} Training Example	`X[i]`, `y[i]`
m	number of training examples	`m`
n	number of features in each example	`n`
$\mathbf{w}$	parameter: weight, vector ↳ length n	`w`
b	parameter: bias number	`b`
$f_{\mathbf{w},b}(\mathbf{x}^{(i)})$	The result of the model evaluation at $\mathbf{x}^{(i)}$ parameterized by $\mathbf{w}, b$: $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b$	`f_wb`

$\vec{w} = (w_1, w_2 \dots w_{16})$

$\vec{d} = (d_1, d_2 \dots d_{16})$ d_j = derivative of CF wrt w_j $d_j = \frac{\partial}{\partial w_j} J(\vec{w})$

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

Gradient descent: repeat till convergence {

for range n { ← n = no of features j

$w_{new}[j] = w_j - \alpha \frac{\partial}{\partial w[j]} J(w_1, \dots w_n, b)$

} } update all w

$b_{new} = b - \alpha \frac{\partial}{\partial b} J(w_1, w_2 .. w_n, b)$ } update b

$\vec{w} = \vec{w}_{new}$

$b = b_{new}$

}

] can be done in vector form without use of for loop

$\vec{w}_{new} = \vec{w} - \alpha \vec{d}$

C1 W2 Lab02

Optional Lab: Multiple Variable Linear Regression

In this lab, you will extend the data structures and previously developed routines to support multiple features. Several routines are updated making the lab appear lengthy, but it makes minor adjustments to previous routines making it quick to review.

Outline

1.1 Goals

- Extend our regression model routines to support multiple features
  - Extend data structures to support multiple features
  - Rewrite prediction, cost and gradient routines to support multiple features
  - Utilize NumPy `np.dot` to vectorize their implementations for speed and simplicity

1.2 Tools

In this lab, we will make use of:

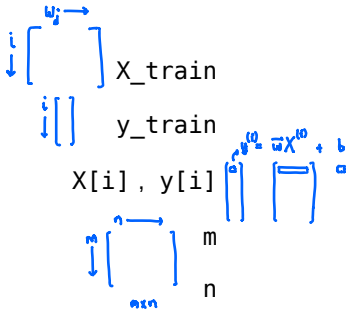
- NumPy, a popular library for scientific computing
- Matplotlib, a popular library for plotting data

In [34]:
```
import copy, math
import numpy as np
import matplotlib.pyplot as plt
plt.style.use('./deeplearning.mplstyle')
np.set_printoptions(precision=2)  # reduced display precision on nu
mpy arrays
```

1.3 Notation

Here is a summary of some of the notation you will encounter, updated for multiple features.

General Notation	Description	Python (if applicable)
a	scalar, non bold	
$\mathbf{a}$	vector, bold	
$\mathbf{A}$	matrix, bold capital	
Regression		
$\mathbf{X}$	training example matrix	X_train
$\mathbf{y}$	training example targets	y_train
$\mathbf{x}^{(i)}$, $y^{(i)}$	i_{th} Training Example	X[i], y[i]
m	number of training examples	m
n	number of features in each example	n
$\mathbf{w}$	parameter: weight,	w
b	parameter: bias	b
$f_{\mathbf{w},b}(\mathbf{x}^{(i)})$	The result of the model evaluation at $\mathbf{x}^{(i)}$ parameterized by $\mathbf{w}, b$: $f_{\mathbf{w},b}(\mathbf{x}^{(i)}) = \mathbf{w} \cdot \mathbf{x}^{(i)} + b$	f_wb



2 Problem Statement

You will use the motivating example of housing price prediction. The training dataset contains three examples with four features (size, bedrooms, floors and, age) shown in the table below. Note that, unlike the earlier labs, size is in sqft rather than 1000 sqft. This causes an issue, which you will solve in the next lab!

Size (sqft)	Number of Bedrooms	Number of floors	Age of Home	Price (1000s dollars)
2104	5	1	45	460
1416	3	2	40	232
852	2	1	35	178

You will build a linear regression model using these values so you can then predict the price for other houses. For example, a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old.

Please run the following code cell to create your `X_train` and `y_train` variables.

In [35]:
```
X_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1, 35]])
y_train = np.array([460, 232, 178])
```

2.1 Matrix X containing our examples

Similar to the table above, examples are stored in a NumPy matrix `X_train` . Each row of the matrix represents one example. When you have m training examples (m is three in our example), and there are n features (four in our example), $\mathbf{X}$ is a matrix with dimensions (m, n) (m rows, n columns).

$$\text{X\_train} = \mathbf{X} = \begin{pmatrix} x_0^{(0)} & x_1^{(0)} & \cdots & x_{n-1}^{(0)} \\ x_0^{(1)} & x_1^{(1)} & \cdots & x_{n-1}^{(1)} \\ \cdots & & & \\ x_0^{(m-1)} & x_1^{(m-1)} & \cdots & x_{n-1}^{(m-1)} \end{pmatrix} \begin{matrix} \\ \\ \\ \downarrow \text{ m examples} \end{matrix}$$

$$\underrightarrow{\text{n features}}$$

notation:

- $\mathbf{x}^{(i)}$ is vector containing example i. $\mathbf{x}^{(i)} = (x_0^{(i)}, x_1^{(i)}, \cdots, x_{n-1}^{(i)})$
- $x_j^{(i)}$ is element j in example i. The superscript in parenthesis indicates the example number while the subscript represents an element.

Display the input data.

In [36]:
```
# data is stored in numpy array/matrix
print(f"X Shape: {X_train.shape}, X Type:{type(X_train)})")
print(X_train)
print(f"y Shape: {y_train.shape}, y Type:{type(y_train)})")
print(y_train)
```

```
X Shape: (3, 4), X Type:<class 'numpy.ndarray'>)
[[2104    5    1   45]
 [1416    3    2   40]
 [ 852    2    1   35]]
y Shape: (3,), y Type:<class 'numpy.ndarray'>)
[460 232 178]
```

2.2 Parameter vector w, b

- $\mathbf{w}$ is a vector with n elements.
  - Each element contains the parameter associated with one feature.
  - in our dataset, n is 4.
  - notionally, we draw this as a column vector

$$\mathbf{w} = \begin{pmatrix} w_0 \\ w_1 \\ \cdots \\ w_{n-1} \end{pmatrix} \text{ n ws for n features}$$

- b is a scalar parameter.

For demonstration, $\mathbf{w}$ and b will be loaded with some initial selected values that are near the optimal. $\mathbf{w}$ is a 1-D NumPy vector.

random initial b & w

In [37]:
```
b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453, -26.421
31618])
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_ini
t)}")
```

```
w_init shape: (4,), b_init type: <class 'float'>
```

3 Model Prediction With Multiple Variables

The model's prediction with multiple variables is given by the linear model:

$$f_{\mathbf{w},b}(\mathbf{x}) = w_0 x_0 + w_1 x_1 + \ldots + w_{n-1} x_{n-1} + b \tag{1}$$

or in vector notation:

$$f_{\vec{\mathbf{w}},b}(\vec{\mathbf{x}}) = \vec{\mathbf{w}} \cdot \vec{\mathbf{x}} + b \tag{2}$$

where $\cdot$ is a vector `dot product`

dot product

To demonstrate the dot product, we will implement prediction using (1) and (2).

3.1 Single Prediction element by element

Our previous prediction multiplied one feature value by one parameter and added a bias parameter. A direct extension of our previous implementation of prediction to multiple features would be to implement (1) above using loop over each element, performing the multiply with its parameter and then adding the bias parameter at the end.

In [38]:
```
def predict_single_loop(x, w, b):
    """
    single predict using linear regression

    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):  model parameter

    Returns:
      p (scalar):  prediction
    """
    n = x.shape[0]
    p = 0
    for i in range(n):
        p_i = x[i] * w[i]
        p = p + p_i
    p = p + b
    return p
```

Prediction M1
non vector
Just calculating all terms $w_j x_j$ individually & adding

In [39]:
```
# get a row from our training data
x_vec = X_train[0,:]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict_single_loop(x_vec, w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

```
x_vec shape (4,), x_vec value: [2104    5    1   45]
f_wb shape (), prediction: 459.9999976194083
```

Note the shape of `x_vec` . It is a 1-D NumPy vector with 4 elements, (4,). The result, `f_wb` is a scalar.

3.2 Single Prediction, vector

Noting that equation (1) above can be implemented using the dot product as in (2) above. We can make use of vector operations to speed up predictions.

Recall from the Python/Numpy lab that NumPy `np.dot()` [link (https://numpy.org/doc/stable/reference/generated/numpy.dot.html)] can be used to perform a vector dot product.

In [40]:
```
def predict(x, w, b):
    """
    single predict using linear regression
    Args:
      x (ndarray): Shape (n,) example with multiple features
      w (ndarray): Shape (n,) model parameters
      b (scalar):             model parameter

    Returns:
      p (scalar):  prediction
    """
    p = np.dot(x, w) + b
    return p
```

Prediction M2

directly

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

In [41]:
```
# get a row from our training data
x_vec = X_train[0,:]
print(f"x_vec shape {x_vec.shape}, x_vec value: {x_vec}")

# make a prediction
f_wb = predict(x_vec,w_init, b_init)
print(f"f_wb shape {f_wb.shape}, prediction: {f_wb}")
```

```
x_vec shape (4,), x_vec value: [2104    5    1   45]
f_wb shape (), prediction: 459.99999761940825
```

The results and shapes are the same as the previous version which used looping. Going forward, `np.dot` will be used for these operations. The prediction is now a single statement. Most routines will implement it directly rather than calling a separate predict routine.

4 Compute Cost With Multiple Variables

The equation for the cost function with multiple variables $J(\mathbf{w}, b)$ is:

$$J(\vec{\mathbf{w}}, b) = \frac{1}{2m} \sum_{i=0}^{m-1} (f_{\vec{\mathbf{w}},b}(\vec{\mathbf{x}}^{(i)}) - y^{(i)})^2 \tag{3}$$

where:

$$f_{\vec{\mathbf{w}},b}(\vec{\mathbf{x}}^{(i)}) = \vec{\mathbf{w}} \cdot \vec{\mathbf{x}}^{(i)} + b \tag{4}$$

In contrast to previous labs, $\mathbf{w}$ and $\mathbf{x}^{(i)}$ are vectors rather than scalars supporting multiple features.

Below is an implementation of equations (3) and (4). Note that this uses a *standard pattern for this course* where a for loop over all `m` examples is used.

In [42]:
```
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
```

In [43]:
```
# Compute and display cost using our pre-chosen optimal parameters.
cost = compute_cost(X_train, y_train, w_init, b_init)
print(f'Cost at optimal w : {cost}')
```

```
Cost at optimal w : 1.5578904880036537e-12
```

Expected Result: Cost at optimal w : 1.5578904045996674e-12

5 Gradient Descent With Multiple Variables

Gradient descent for multiple variables:

$$
\begin{aligned}
&\text{repeat until convergence: } \{ \\
&\quad w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \qquad \text{for j} = 0..\text{n-1} \\
&\quad b \;\; = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \\
&\}
\end{aligned} \tag{5}
$$

update all w_s & one b

where, n is the number of features, parameters w_j, b, are updated simultaneously and where

$$\frac{\partial J(\mathbf{w},b)}{\partial w_j} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_j^{(i)} \tag{6}$$

for a particular w happens

$$\frac{\partial J(\mathbf{w},b)}{\partial b} = \frac{1}{m}\sum_{i=0}^{m-1}(f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \tag{7}$$

- m is the number of training examples in the data set
- $f_{\mathbf{w},b}(\mathbf{x}^{(i)})$ is the model's prediction, while $y^{(i)}$ is the target value

5.1 Compute Gradient with Multiple Variables

An implementation for calculating the equations (6) and (7) is below. There are many ways to implement this. In this version, there is an

- outer loop over all m examples.
  - $\frac{\partial J(\mathbf{w},b)}{\partial b}$ for the example can be computed directly and accumulated
  - in a second loop over all n features:
    - $\frac{\partial J(\mathbf{w},b)}{\partial w_j}$ is computed for each w_j.

To compute gradients

In [44]:
```
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 par
ameters w.
      dj_db (scalar):       The gradient of the cost w.r.t. the par
ameter b.
    """
    m,n = X.shape           #(number of examples, number of feature
s)
    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]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```

In [45]:
```
#Compute and display gradient
tmp_dj_db, tmp_dj_dw = compute_gradient(X_train, y_train, w_init,
b_init)
print(f'dj_db at initial w,b: {tmp_dj_db}')
print(f'dj_dw at initial w,b: \n {tmp_dj_dw}')
```

```
dj_db at initial w,b: -1.673925169143331e-06
dj_dw at initial w,b:
 [-2.73e-03 -6.27e-06 -2.22e-06 -6.92e-05]
```

Expected Result:
dj_db at initial w,b: -1.6739251122999121e-06
dj_dw at initial w,b:
[-2.73e-03 -6.27e-06 -2.22e-06 -6.92e-05]

5.2 Gradient Descent With Multiple Variables

The routine below implements equation (5) above.

In [46]:
```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_fun
ction, alpha, num_iters):
    """
    Performs batch gradient descent to learn w and b. Updates w and
b 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 de
scent

    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 funct
ion
    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 iterati
ons 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 grap
hing
```

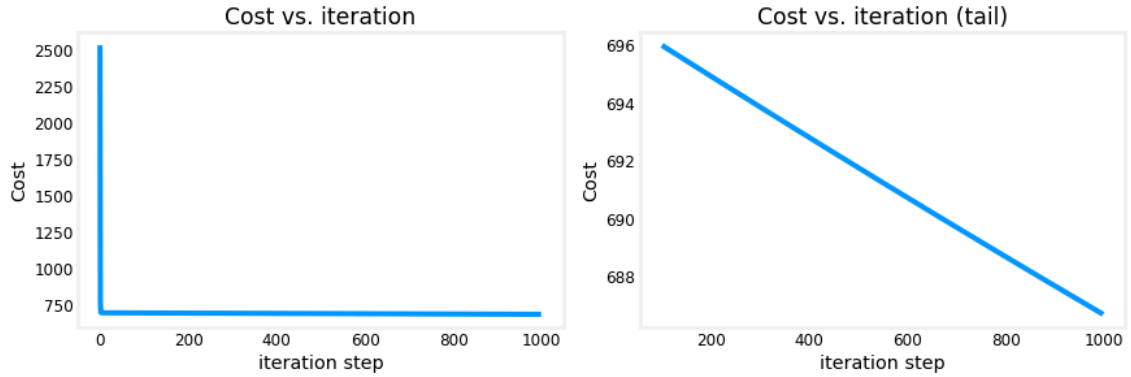
In the next cell you will test the implementation.

In [47]:
```
# 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, init
ial_w, initial_b,
                                                    compute_cost, c
ompute_gradient,
                                                    alpha, iteratio
ns)
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:0.2
f}, target value: {y_train[i]}")
```

```
Iteration    0: Cost  2529.46
Iteration  100: Cost   695.99
Iteration  200: Cost   694.92
Iteration  300: Cost   693.86
Iteration  400: Cost   692.81
Iteration  500: Cost   691.77
Iteration  600: Cost   690.73
Iteration  700: Cost   689.71
Iteration  800: Cost   688.70
Iteration  900: Cost   687.69
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
```

Expected Result:
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

In [48]:
```
# plot cost versus iteration
fig, (ax1, ax2) = plt.subplots(1, 2, constrained_layout=True, figsize=(12, 4))
ax1.plot(J_hist)
ax2.plot(100 + np.arange(len(J_hist[100:])), J_hist[100:])
ax1.set_title("Cost vs. iteration");  ax2.set_title("Cost vs. iteration (tail)")
ax1.set_ylabel('Cost')             ;  ax2.set_ylabel('Cost')
ax1.set_xlabel('iteration step')   ;  ax2.set_xlabel('iteration step')
plt.show()
```



These results are not inspiring! Cost is still declining and our predictions are not very accurate. The next lab will explore how to improve on this.

6 Congratulations!

In this lab you:

- Redeveloped the routines for linear regression, now with multiple variables.
- Utilized NumPy `np.dot` to vectorize the implementations

Techniques that help run GD better

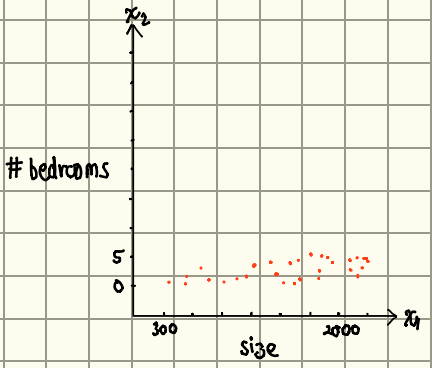
Feature Scaling makes GD run = faster

Eg: x_1 = size (sq ft) : ranges 300 - 2000 — big range, big numbers also
x_2 = no of bedrooms : ranges 0 - 5 — small range, not as big numbers

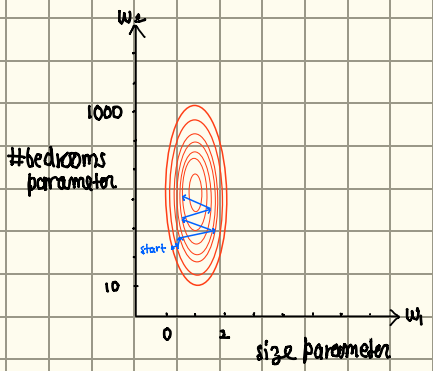
⇒ small change in w_1 will make large diff in estimated price
while large change in w_2 makes small change in estimated price

for a good model, final

⇒ w_1 is likely to be small range ∵ x_1 big range
⇒ w_2 is likely to be big range ∵ x_2 small range



Features scatterplot

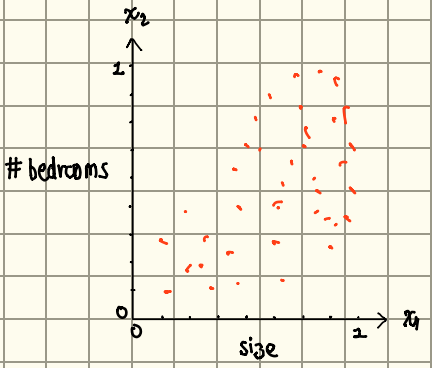


Cost Func Contour plot

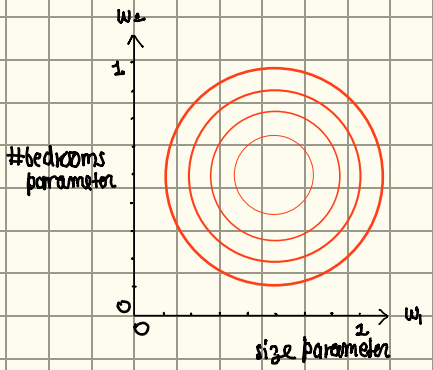
∴ contour plot skinny
GD may keep bouncing back & forth for a long time ∵ change in w_1 is too sensitive

Solution: Scale the feature : bring down the range of x_1 ⇒ w_1 range becomes wider

Goal after Scaling



Features scatterplot



Cost Func Contour plot

GD will run better now
steps aren't very sensitive

AUX: Take features that take very different range of values & scale them to have comparable ranges of values to each other.

Scale : simplest way (division by max) ⇒ $300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

$$x_{1,\text{scaled}} = \frac{x_1}{2000} \qquad x_{2,\text{scaled}} = \frac{x_2}{5}$$

$$\Rightarrow 0.15 \le x_{1,\text{scaled}} \le 1 \qquad 0 \le x_{2,\text{scaled}} \le 1$$

: mean normalization (centered around 0) ⇒ μ_1 = mean of data x_1, μ_2 = mean of data x_2 ⇒ $300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

$$x_1 = \frac{x_1 - \mu_1}{2000 - 300} \qquad x_2 = \frac{x_2 - \mu_2}{5 - 0}$$

$$\Rightarrow -0.18 \le x_1 \le 0.82 \qquad -0.46 \le x_2 \le 0.54$$

: z-score normalization ⇒ σ_1, σ_2 (std deviations) ⇒ $300 \le x_1 \le 2000$ $\quad 0 \le x_2 \le 5$

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

$$\Rightarrow -0.67 \le x_1 \le 3.1 \qquad -1.6 \le x_2 \le 1.9$$

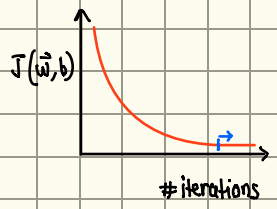
When scale : aim for $-1 \le x_j \le 1$, $-3 \le x_j \le 3$, $-0.3 \le x_j \le 0.3$ } acceptable ranges

: Example :

$0 \le x_j \le 3$		OK
-2	0.5	OK
-100	100	Range↑ Rescale
-0.001	0.001	Range↓ Rescale
98.6	105	Range fine but numbers large, rescale

Checking GD for convergence

↳ Plot $J(\vec{w}, b)$ vs no of iterations (learning curve)



- J should decrease after every single iteration (if J↑ anytime ⇒ α choice bad)
- When curve no longer dec ⇒ GD converged

↳ Automatic convergence test

⇒ Let $\varepsilon = 10^{-3}$ (some very small chosen number)

If in any iteration $J(\vec{w}, b)$ decreases by $\leq \varepsilon$ ⇒ converged

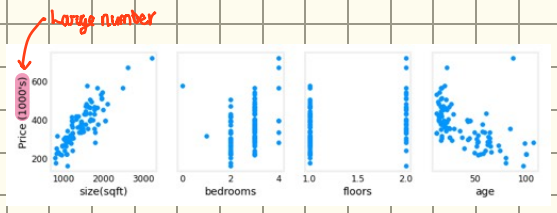
Choosing Learning Rate

α↓ : inefficient ; α↑ : may never converge

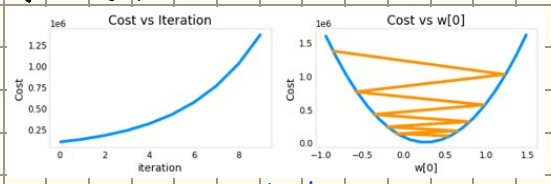
Good way: choose small α & keep inc & make sure J always ↓ until satisfactory α

C1 W2 Lab03

Size (sqft)	Number of Bedrooms	Number of floors	Age of Home	Price (1000s dollars)
952	2	1	65	271.5
1244	3	2	64	232
1947	3	2	17	509.8
...	...	...	...	...

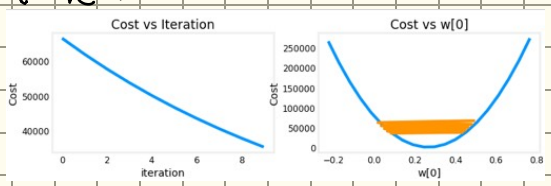


α = 9.9e-7



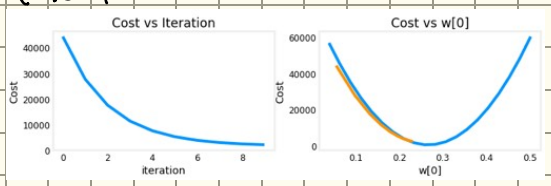
Overshoot

α = 9e-7



J↓ but still α↑ ∵ oscillating
decreasing high

α = 1e-7



nice α

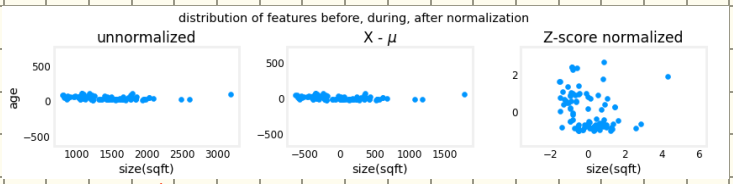
Z-score normalisation : all features will have $\mu_j = 0$ & $\sigma_j = 1$

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

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

    Returns:
      X_norm (ndarray (m,n)): input normalized by column
      mu (ndarray (n,))     : mean of each feature
      sigma (ndarray (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)
```

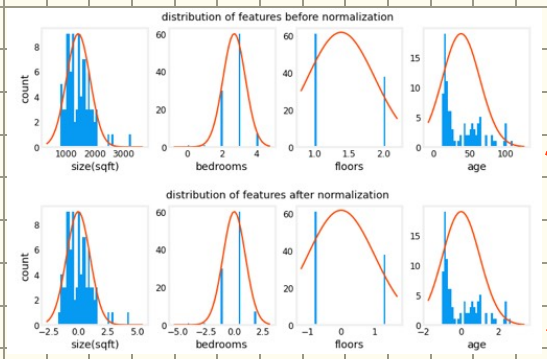
$\mu_j = \frac{1}{m}\sum_{i=0}^{m-1} x_j^{(i)}$

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

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



size range too large



notice range change while distribution remains same

Now we can run gradient descent with a much larger value of α, making GD faster

In order to predict after normalizing, we must normalize our input features with the same σ, μ derived when scaling the features

```
In [17]: # First, normalize out example.
         x_house = np.array([1200, 3, 1, 40])
         x_house_norm = (x_house - X_mu) / X_sigma
         print(x_house_norm)
         x_house_predict = np.dot(x_house_norm, w_norm) + b_norm
         print(f" predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old = ${x_house_predict*1000:0.0f}

         [-0.53  0.43 -0.79  0.06]
          predicted price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old = $318709
```

Feature Engineering

↳ Choosing most appropriate features for good model

Eg: House price prediction : 2 features : x_1 : length
x_2 : width

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

It is possible that maybe area is better representative of price of the house.
To know what feature will help model be better,
Introduce new feature $x_3 = x_1 x_2$

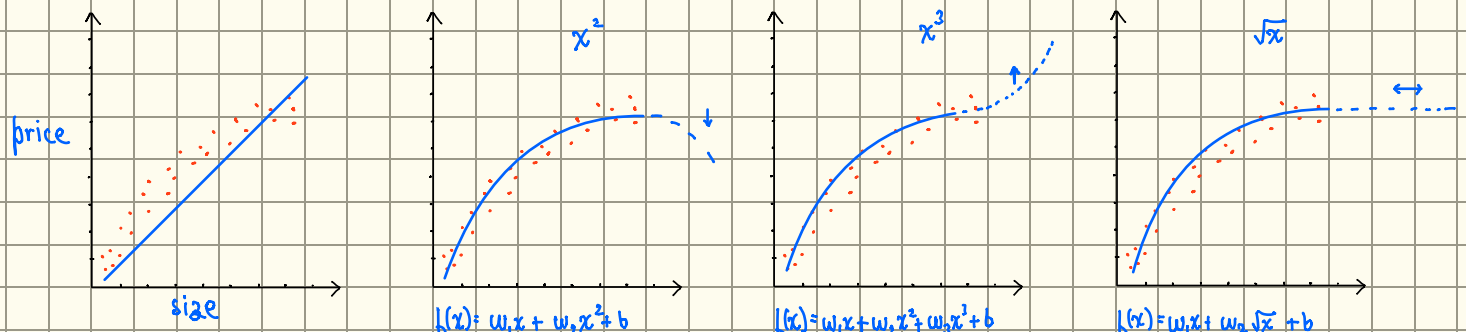
New model $\Rightarrow f_{\vec{w},b}(\vec{x}) = w_1 x_1 + w_2 x_2 + w_3 x_3 + b$

After training the model, weights w_1, w_2 & w_3 will determine which feature has more impact on the prediction
If w_3 greater $\Rightarrow$ area is better representative of price of house

$\Rightarrow$ Feature engineering : Using intuition to design new features, by transforming or combining original features

Polynomial Regression : combining multiple linear regression w feature engineering

↳ Lets us fit curves (polynomials) & logarithm functions to our data



no w, b of linear model can fit this data well

$f(x) = w_1 x + w_2 x^2 + b$
but it starts to go down, doesn't make sense

$f(x) = w_1 x + w_2 x^2 + w_3 x^3 + b$
makes more sense

$f(x) = w_1 x + w_2 \sqrt{x} + b$
also makes a bit sense

Above three = polynomial regression examples

: features can be x, x^2 $\Rightarrow$ big number range $\Rightarrow$ feature scaling very imp

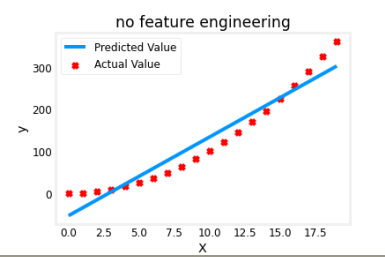
C1 W2 Lab 04

Example simple quadratic $y = 1 + x^2$

No Feature Engineering

$x = \begin{bmatrix} 0 \\ \vdots \\ 19 \end{bmatrix}$ $y = 1 + x**2$ (features, targets)

$w, b = \text{run\_gd}(x, y, \text{iterations} = 1000, \alpha = 0.01)$



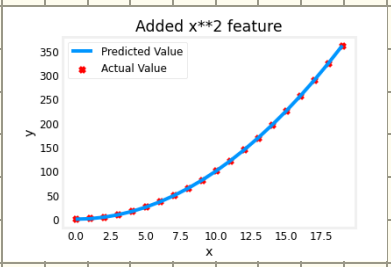
model $\Rightarrow f_{w,b}(x) = wx + b$

w: 18.7, b: -52.08

Feature engineering

$x = \begin{bmatrix} 0 \\ \vdots \\ 19 \end{bmatrix}$ $X = x**2 = \begin{bmatrix} 0 \\ \vdots \\ 19^2 \end{bmatrix}$ $y = x**2 + 1$

$w, b = \text{run\_gd}(X, y, \text{iterations} = 10000, \alpha = 0.0001)$



model $\Rightarrow f_{w,b}(x) = wx^2 + b$

w: 1, b: 0.3701

$\Rightarrow y = 1 * x^2 + 0.3701$ $\Rightarrow$ very close to target model $y = 1 * x^2 + 1$ (run longer $\Rightarrow$ better b)

Selecting features

: Run GD w training data & various transformed input features

Eg: $f_{\vec{w},b}(\vec{x}) = w_1 x + w_2 x^2 + w_3 x^3 + w_4 \sqrt{x} + b$

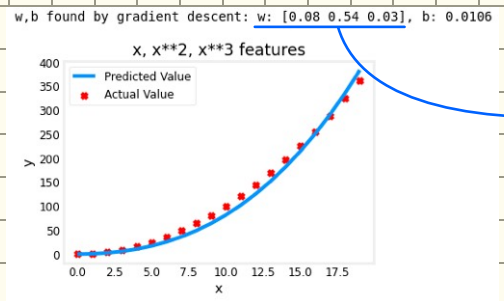
: After gradient descent, the feature that fits the data best will clearly have the highest weight of all

Eg: $x = \begin{bmatrix} 0 \\ \vdots \\ 19 \end{bmatrix}$ $X = [x, x^2, x^3]$ $y = x^2$

$\vec{w}, b = \text{run\_gd}(X, y, \text{iter} = 10000, \alpha = 0.0001)$

we should normalize the features
$X = \text{zscore\_normalize\_features}(X)$
this way we could get w $\alpha = 0.1$ also
or maybe less iterations also

```
w,b found by gradient descent: w: [0.08 0.54 0.03], b: 0.0106
```

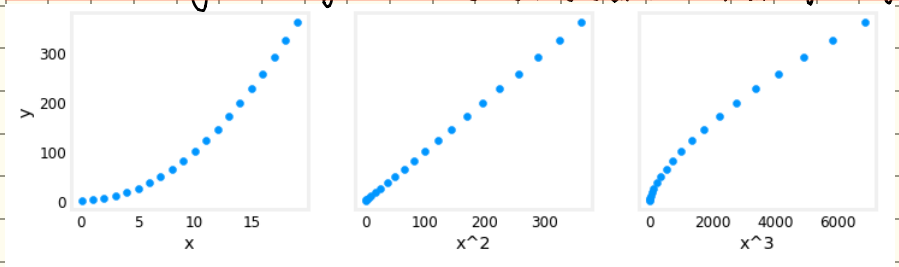


$w_2 \uparrow \Rightarrow$ naturally shows x^2 fits the data well
If GD ran for longer $w_1 \downarrow$ $w_3 \downarrow$ & $w \uparrow$

Alternate view : Above, a polynomial feature was chosen based on how well it matched the target data

: We still are doing linear regression, but now instead of fitting data of y against x, now we are trying to fit some power of y^n against some x^n

: We still are doing linear regression once we have created new transformed features



x^2 best fits polynomial y

Cost function of x^2 will be lower against y

Feature engineering w complex functions

$x = \begin{bmatrix} 0 \\ \vdots \\ 19 \end{bmatrix}$ $X = [x, x^2, x^3, x^4, \ldots, x^{13}]$ $y = \cos(x/2)$

$X = \text{normalize}(X)$

$\vec{w}, b = \text{run\_gd}(X, y, \text{iter} = 10000, \alpha = 0.1)$

```
w,b found by gradient descent: w: [ -1.34 -10.    24.78   5.96 -12.49 -16.26  -9.51   0.59   8.7   11.94
  9.27   0.79 -12.82], b: -0.0073
```