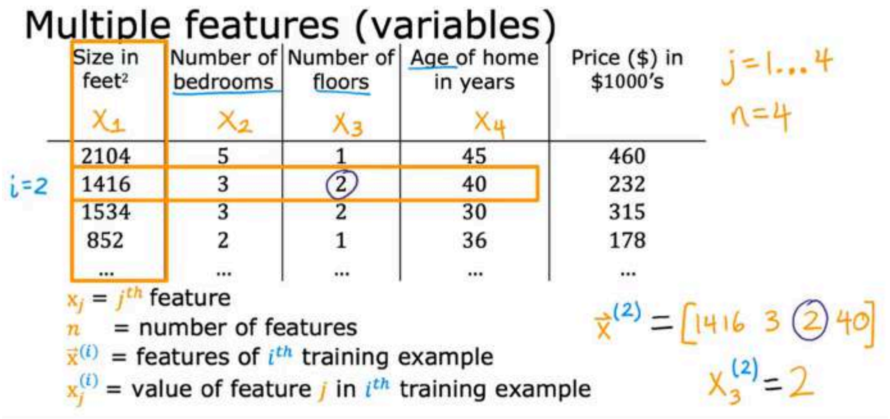
Week 2

Multiple features



Multiple features (variables)

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

$j = 1 \ldots 4$

$n = 4$

x_j = j^{th} feature

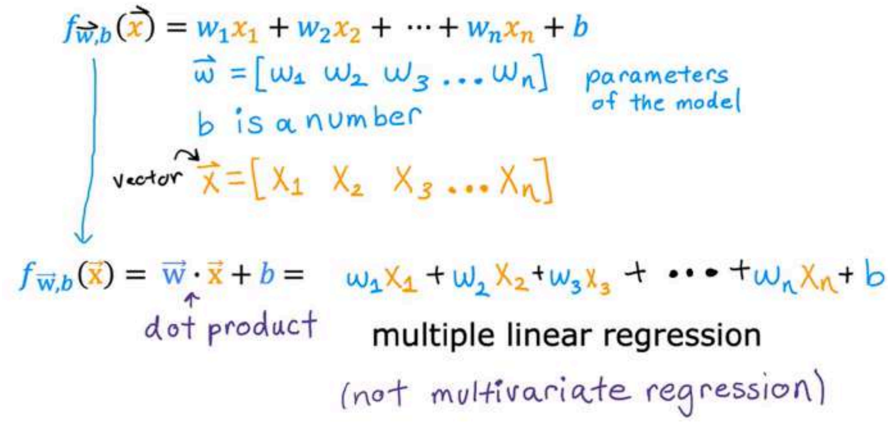
n = number of features

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

$x_j^{(i)}$ = value of feature j in i^{th} training example

$\vec{x}^{(2)} = [1416 \quad 3 \quad \boxed{2} \quad 40]$

$x_3^{(2)} = 2$



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

$\vec{w} = [w_1 \ w_2 \ w_3 \ldots w_n]$ parameters of the model

b is a number

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

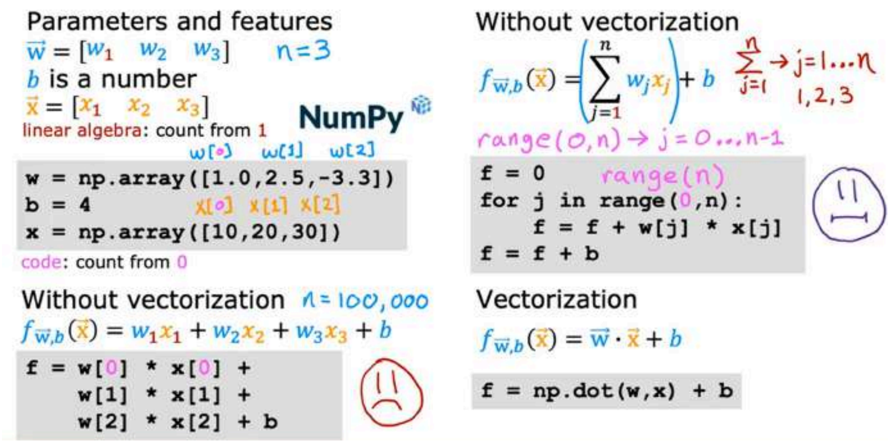
$$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b = w_1x_1 + w_2x_2 + w_3x_3 + \cdots + w_nx_n + b$$

dot product

multiple linear regression

(not multivariate regression)

Vectorization part 1



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]

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

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

code: count from 0

Without vectorization $n = 100{,}000$

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

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

Without vectorization

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

$\sum_{j=1}^{n} \rightarrow j = 1 \ldots n$

$1, 2, 3$

range(0,n) → j = 0...n-1

range(n)

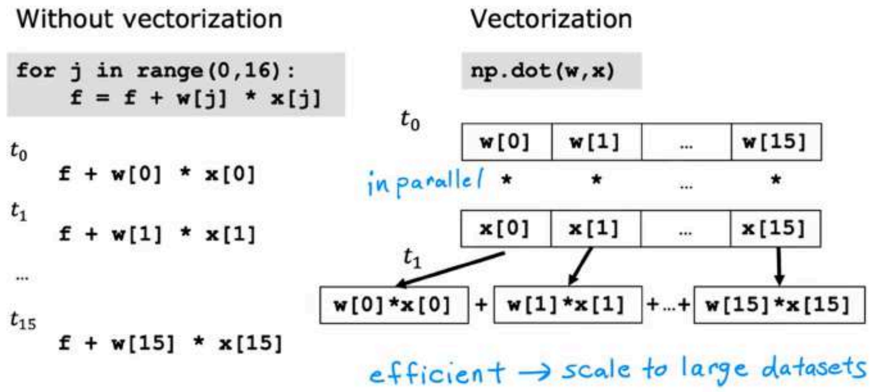
```
f = 0
for j in range(0,n):
    f = f + w[j] * x[j]
f = f + b
```

Vectorization

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

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

Vectorization part 2



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)
```

t_0

w[0] w[1] ... w[15]

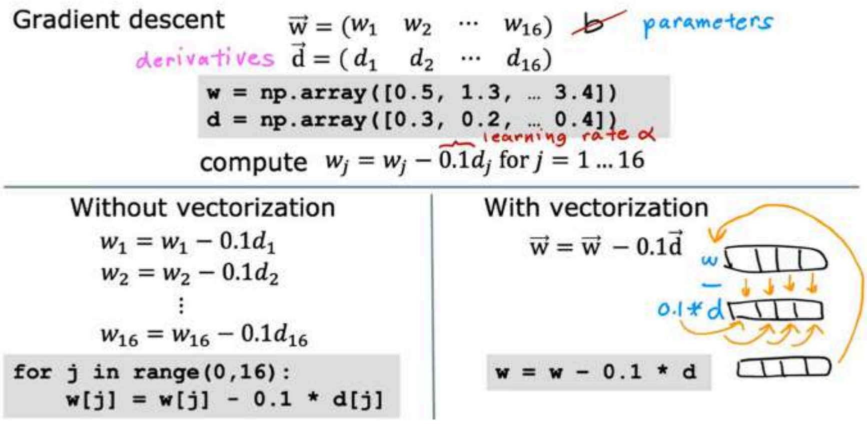
in parallel * * ... *

x[0] x[1] ... x[15]

t_1

w[0]*x[0] + w[1]*x[1] +...+ w[15]*x[15]

efficient → scale to large datasets



Gradient descent $\vec{w} = (w_1 \quad w_2 \quad \cdots \quad w_{16})$ ~~b~~ 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 $w_j = w_j - \overset{\text{learning rate }\alpha}{0.1}d_j$ for $j = 1 \dots 16$

Without vectorization

$w_1 = w_1 - 0.1d_1$

$w_2 = w_2 - 0.1d_2$

$\vdots$

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

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

With vectorization

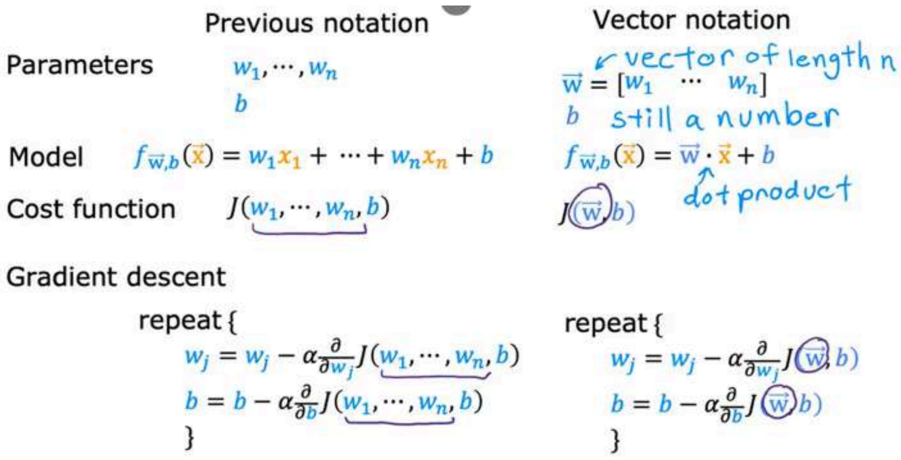
$\vec{w} = \vec{w} - 0.1\vec{d}$

w − 0.1*d

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

remember here, w is a vector containing multiple values of w's

Gradient descent for multiple linear regression

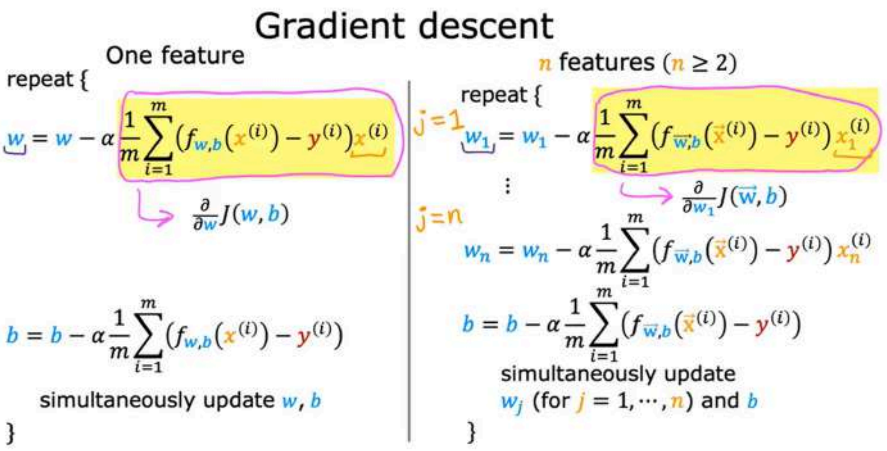


	Previous notation	Vector notation
Parameters	$w_1, \cdots, w_n$ b	vector of length n $\vec{w} = [w_1 \quad \cdots \quad w_n]$ b still a number
Model	$f_{\vec{w},b}(\vec{x}) = w_1x_1 + \cdots + w_nx_n + b$	$f_{\vec{w},b}(\vec{x}) = \vec{w} \cdot \vec{x} + b$ dot product
Cost function	$J(\underbrace{w_1, \cdots, w_n}, b)$	$J(\vec{w}, b)$

Gradient descent

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

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 {

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

→ $\frac{\partial}{\partial w}J(w,b)$

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

simultaneously update w, b

}

n features ($n \geq 2$)

repeat {

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

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

$\vdots$

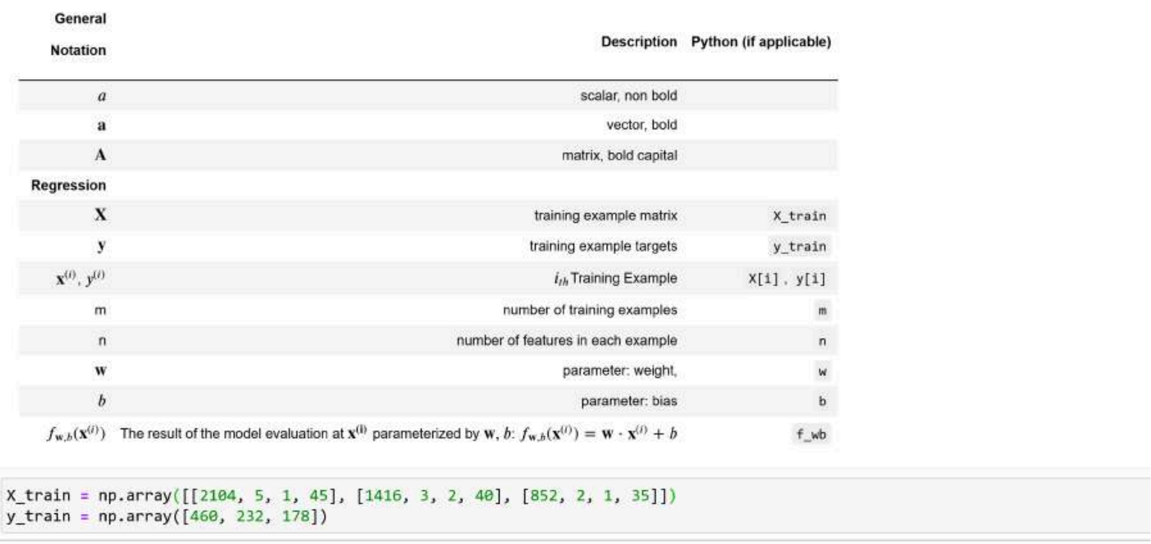
j=n $w_n = w_n - \alpha\frac{1}{m}\sum_{i=1}^{m}(f_{\vec{w},b}(\vec{x}^{(i)}) - y^{(i)})x_n^{(i)}$

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

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

}

C1_W2_Lab02_Multiple_Variable_Soln



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

```
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).

$$\mathbf{X} = \begin{pmatrix} x^{(0)}_0 & x^{(0)}_1 & \cdots & x^{(0)}_{n-1} \\ x^{(1)}_0 & x^{(1)}_1 & \cdots & x^{(1)}_{n-1} \\ \cdots \\ x^{(m-1)}_0 & x^{(m-1)}_1 & \cdots & x^{(m-1)}_{n-1} \end{pmatrix}$$

notation:

- $\mathbf{x}^{(i)}$ is vector containing example i. $\mathbf{x}^{(i)} = (x^{(i)}_0, x^{(i)}_1, \cdots, x^{(i)}_{n-1})$
- $x^{(i)}_j$ is element j in example i. The superscript in parenthesis indicates the example number while the subscript represents an element.

Display the input data.

```
import copy, math
import numpy as np
import matplotlib.pyplot as plt

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

# 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]"""
```

- $\mathbf{w}$ 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

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

- b is a scalar parameter.

```
b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453, -26.42131618]) #w and b are selected for optimal solns as they are
better
print(f"w_init shape: {w_init.shape}, b_init type: {type(b_init)}")
```

$f_{\mathbf{w},b}(\mathbf{x}) = w_0x_0 + w_1x_1 + \ldots + w_{n-1}x_{n-1} + b$

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

Single Prediction element by element

```
def predict_single_loop(x, w, b):
    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
    # row wise
    '''
# 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 '''
```

Single Prediction, vector

```
def predict(x, w, b):
    p = np.dot(x, w) + b
    return p
```

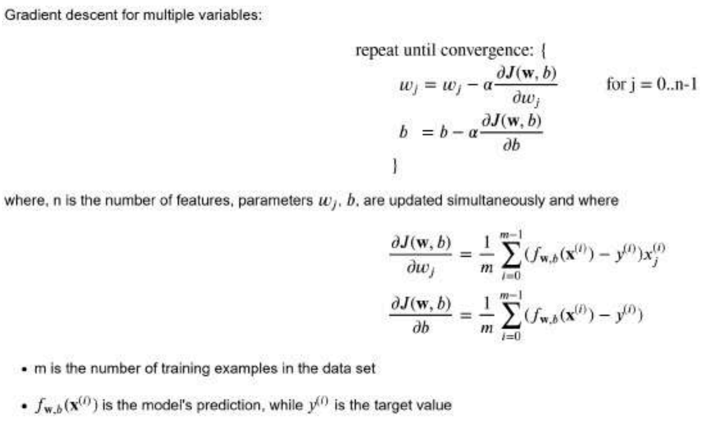
Compute Cost With Multiple Variables

$$J(\mathbf{w},b) = \frac{1}{2m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})^2$$

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

```
def compute_cost(X, y, w, b):
    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
```

Gradient Descent with multiple variables



Gradient descent for multiple variables:

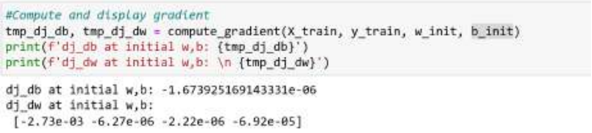
$$\begin{align*} \text{repeat}&\text{ until convergence:} \; \lbrace \newline\; & w_j = w_j - \alpha \frac{\partial J(\mathbf{w},b)}{\partial w_j} \; & \text{for j = 0..n-1}\newline &b\ \ = b - \alpha \frac{\partial J(\mathbf{w},b)}{\partial b} \newline \rbrace \end{align*}$$

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

$$\begin{align} \frac{\partial J(\mathbf{w},b)}{\partial w_j} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)})x_{j}^{(i)} \\ \frac{\partial J(\mathbf{w},b)}{\partial b} &= \frac{1}{m} \sum\limits_{i = 0}^{m-1} (f_{\mathbf{w},b}(\mathbf{x}^{(i)}) - y^{(i)}) \end{align}$$

- 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

```
def compute_gradient(X, y, w, 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]
        dj_db = dj_db + err
    dj_dw = dj_dw / m
    dj_db = dj_db / m

    return dj_db, dj_dw
```



```
#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]
```

Gradient Descent With Multiple Variables

```
def gradient_descent(X, y, w_in, b_in, cost_function, gradient_function, alpha, num_iters):

    # 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


#------------client code
# 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]}")
```

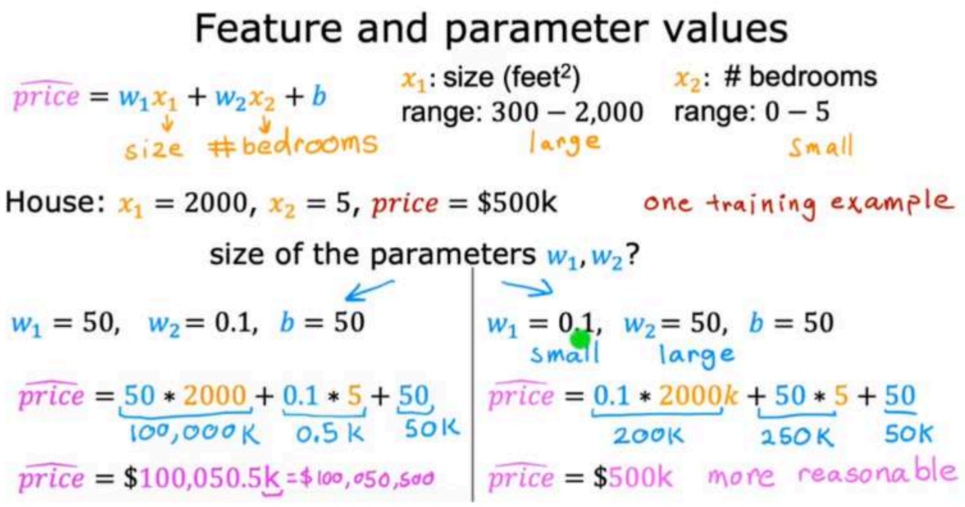
final code using vectorization

```
import numpy as np

import copy

'''
x_train = np.array([1.0, 2.0])

y_train = np.array([300.0, 500.0])

w_init = 0

b_init = 0

alpha = 0.001

num_iters = 100000
```

```
def costfn(x_train, y_train, w,b):

    j_wb = 0

    m = x_train.shape[0]

    for i in range(m):

        f_wb = w*x_train[i]+b

        j_wb = j_wb + (f_wb-y_train[i])**2

    j_wb = j_wb/(2*m)



    return j_wb

def gradientfn(x_train, y_train, w,b):

    m = x_train.shape[0]

    d_jwb_w = 0

    d_jwb_b = 0



    for i in range(m):

        f_wb = w*x_train[i] + b

        d_jwb_w = d_jwb_w + (f_wb-y_train[i])*x_train[i]

        d_jwb_b = d_jwb_b + (f_wb-y_train[i])

    d_jwb_w = d_jwb_w/m

    d_jwb_b = d_jwb_b/m



    return d_jwb_w, d_jwb_b

def gradient_descent(x_train, y_train, alpha, w_init, b_init, num_iters, costfn, gradientfn):

    w,b = w_init, b_init

    dj_dw , dj_db = gradientfn(x_train, y_train, w,b)

    jwb = []



    for i in range(num_iters):

        dj_dw , dj_db = gradientfn(x_train, y_train, w,b)

        w = w - alpha*dj_dw

        b = b - alpha*dj_db

        if (i%1000)==0:

            jwb.append(costfn(x_train, y_train, w,b))

    return w,b,jwb

w_final, b_final,j_wb = gradient_descent(x_train, y_train, alpha, w_init,b_init, num_iters, costfn, gradientfn)

print(f'w_final:{w_final:.2f}, b_final:{b_final:.2f}')

print(j_wb)
'''



X_train = np.array([[2104, 5, 1, 45], [1416, 3, 2, 40], [852, 2, 1, 35]])
```

```
y_train = np.array([460, 232, 178])
alpha = 5.0e-7
num_iters = 1000

b_init = 785.1811367994083
w_init = np.array([ 0.39133535, 18.75376741, -53.36032453, -26.42131618])

'''def predict_single_loop(x, y, w, b):

    n = x.shape[0]
    f_wb = 0
    for i in range(n):
        f_wb += w[i]*x[i]
    f_wb +=b
    return f_wb

print(predict_single_loop(X_train[0], y_train, w_init, b_init))'''
'''
def predict(x,w,b):
    f_wb = np.dot(x,w) + b
    print(f_wb)
    #here f_wb is a list of predictions, number of f_wb is equal to x.shape[0] (the number of rows)
predict(X_train, w_init,b_init)
'''


def compute_cost(X, y, w,b):
    m = X.shape[0]
    cost = 0
    for i in range(m):
        f_wb_i = np.dot(X[i],w)+b
        cost += (f_wb_i-y[i])**2

    cost = cost/(2*m)

    return cost

def compute_gradient(X, y, w,b):
    m,n = X.shape
    dj_dw = np.zeros(n)
    dj_db = 0
    for i in range(m):
        error = np.dot(X[i],w)+b  - y[i]
        for j in range(n):
            dj_dw[j] += error*X[i,j]
        dj_db += error
    dj_db /= m
    dj_dw /= m

    return dj_dw, dj_db


def gradient_descent(X_train, y_train, alpha, num_iters, w_init, b_init, compute_cost, compute_gradient):
    jwb = []
    w = copy.deepcopy(w_init)
    b = b_init
    for i in range(num_iters):
        dj_dw, dj_db = compute_gradient(X_train, y_train, w, b)
        w = w - alpha*dj_dw
        b = b - alpha*dj_db

        if i%100==0:
            jwb.append(compute_cost(X_train, y_train, w_init, b_init))
    return w,b,jwb


b_init = 0
w_init = np.zeros(X_train.shape[1])


w_final,b_final,j_wb = gradient_descent(X_train, y_train, alpha, num_iters, w_init, b_init, compute_cost, compute_gradient)
print(w_final, b_final)

print(j_wb)


f_wb = np.dot(X_train,w_final) + b_final
print(f_wb)
```

Feature scaling Part 1



Feature and parameter values

$\widehat{price} = \underset{\text{size}}{w_1 x_1} + \underset{\text{\#bedrooms}}{w_2 x_2} + b$

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

x_2: # bedrooms range: 0 – 5 Small

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

size of the parameters w_1, w_2?

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

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

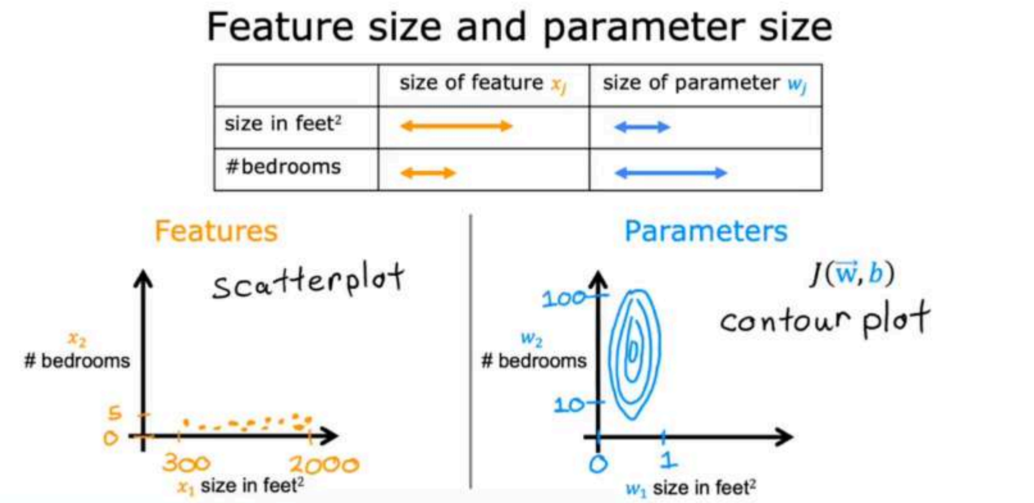
$\widehat{price} = \$100{,}050.5\text{k} = \$100{,}050{,}500$

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

small large

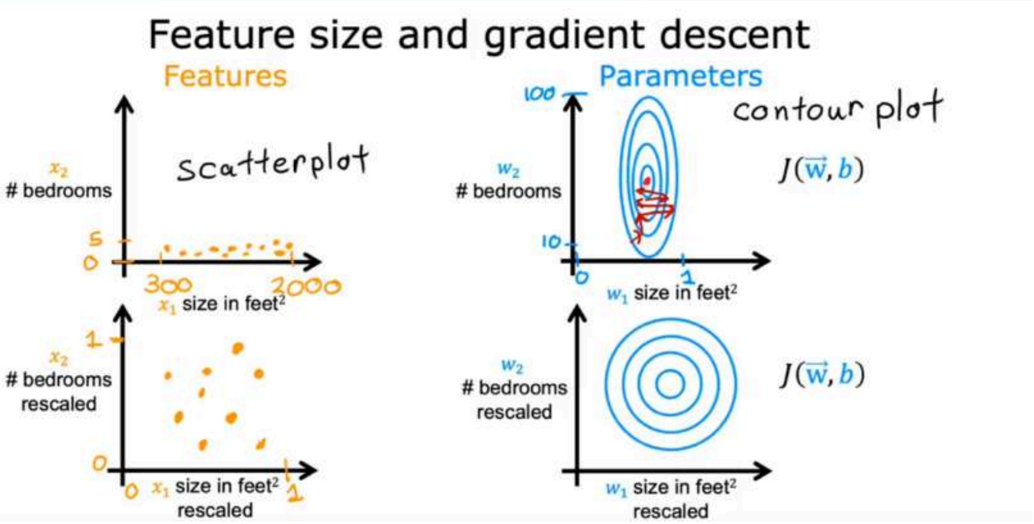
$\widehat{price} = \underbrace{0.1 * 2000k}_{200K} + \underbrace{50 * 5}_{250K} + \underbrace{50}_{50K}$

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



Feature size and parameter size

	size of feature x_j	size of parameter w_j
size in feet²	⟷	↔
#bedrooms	↔	⟷

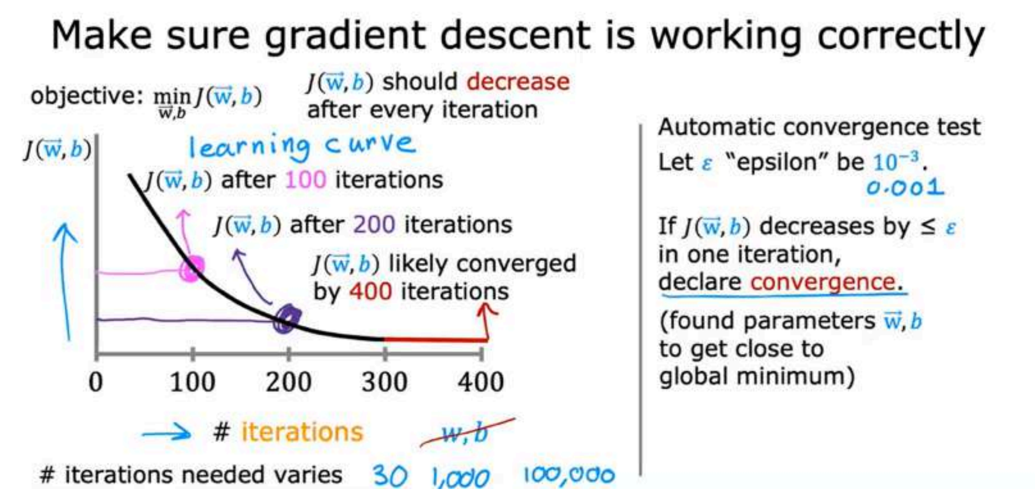


Feature size and gradient descent

the cost function bounces a lot until it reaches the center for a feature set not scaled.

Feature scaling part 2 - in notes

Checking gradient descent for convergence


Make sure gradient descent is working correctly

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

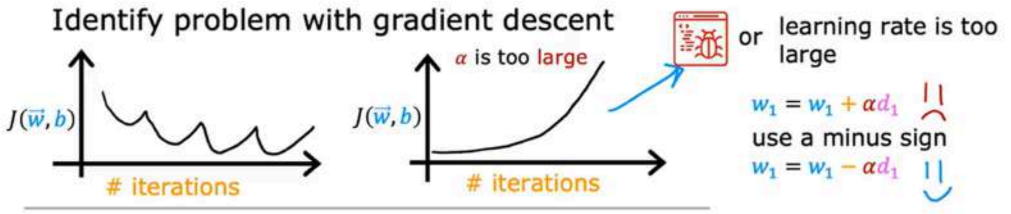
$J(\vec{w},b)$ should decrease after every iteration

Automatic convergence test
Let ε "epsilon" be 10^{-3}.
0.001

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

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

iterations needed varies 30 1,000 100,000

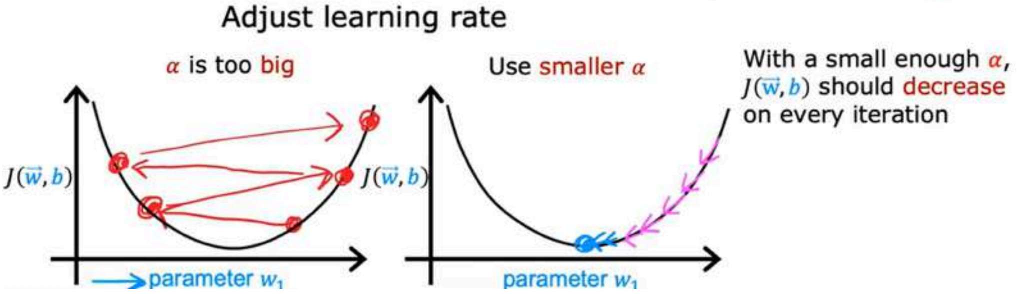

Identify problem with gradient descent

or learning rate is too large

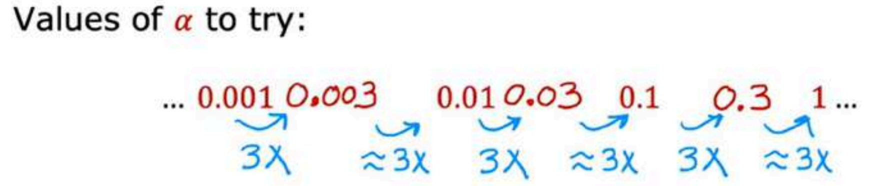
$w_1 = w_1 + \alpha d_1$

use a minus sign

$w_1 = w_1 - \alpha d_1$

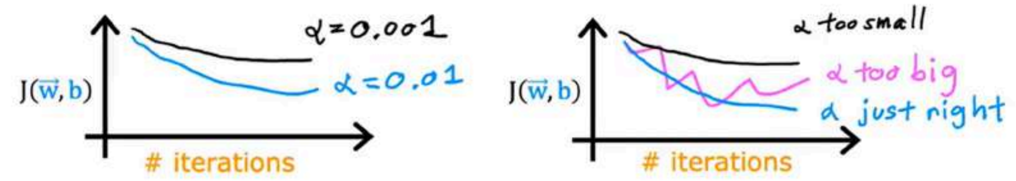

Adjust learning rate

With a small enough α, $J(\vec{w},b)$ should decrease on every iteration


Values of α to try:

... 0.001 0.003 0.01 0.03 0.1 0.3 1 ...

3X ≈3X 3X ≈3X 3X ≈3X



how to choose alpha effectively

C1_W2_Lab03_Feature_Scaling_and_Learning_Rate_Solne

- After z-score normalization, all features will have a mean of 0 and a standard deviation of 1.
- $$x_j^{(i)} = \frac{x_j^{(i)} - \mu_j}{\sigma_j}$$
- where jj selects a feature or a column in the $\mathbf{X}$X matrix. µjµj is the mean of all the values for feature (j) and σjσj is the standard deviation of feature (j).
- $$\mu_j = \frac{1}{m} \sum_{i=0}^{m-1} x_j^{(i)}$$
  $$\sigma_j^2 = \frac{1}{m} \sum_{i=0}^{m-1} (x_j^{(i)} - \mu_j)^2$$
- **Implementation Note:** When normalizing the features, it is important to store the values used for normalization - the mean value and the standard deviation used for the computations. After learning the parameters from the model, we often want to predict the prices of houses

we have not seen before. Given a new x value (living room area and number of bed- rooms), we must first normalize x using the mean and standard deviation that we had previously computed from the training set.

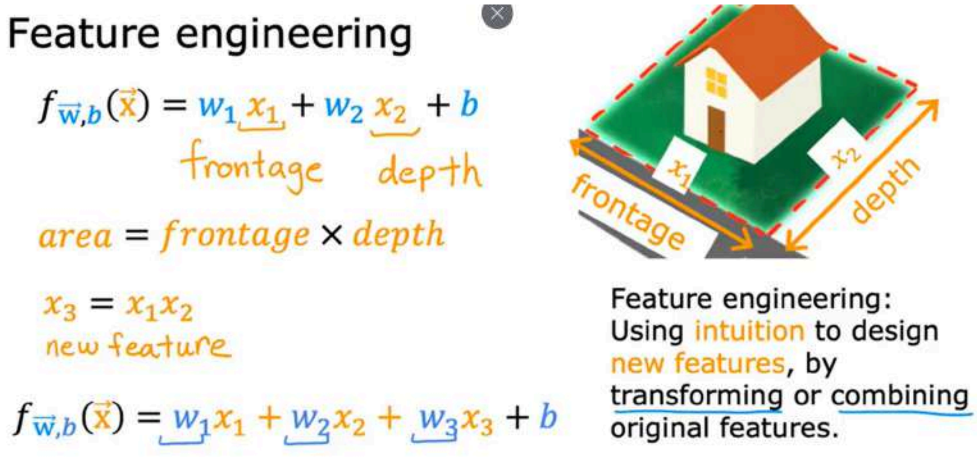
```
def zscore_normalize_features(X):
    # find the mean of each column/feature
    mu     = np.mean(X, axis=0)                 # mu will have shape (n,)
    # find the standard deviation of each column/feature
    sigma  = np.std(X, axis=0)                  # sigma will have shape (n,)
    # element-wise, subtract mu for that column from each example, divide by std for that column
    X_norm = (X - mu) / sigma

    return (X_norm, mu, sigma)


"""or just"""

mu     = np.mean(X_train,axis=0)
sigma  = np.std(X_train,axis=0)
X_mean = (X_train - mu)
X_norm = (X_train - mu)/sigma
```

A few points to note:

- with multiple features, we can no longer have a single plot showing results versus features.
- when generating the plot, the normalized features were used. Any predictions using the parameters learned from a normalized training set must also be normalized.

```
"""**Prediction** The point of generating our model is to use it to predict housing prices that are not in the data set. Let's
predict the price of a house with 1200 sqft, 3 bedrooms, 1 floor, 40 years old. Recall, that you must normalize the data with the
mean and standard deviation derived when the training data was normalized."""
# 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}")

#output
#[-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


Feature engineering

$$f_{\vec{w},b}(\vec{x}) = w_1 \underbrace{x_1}_{\text{frontage}} + w_2 \underbrace{x_2}_{\text{depth}} + b$$

$area = frontage \times depth$

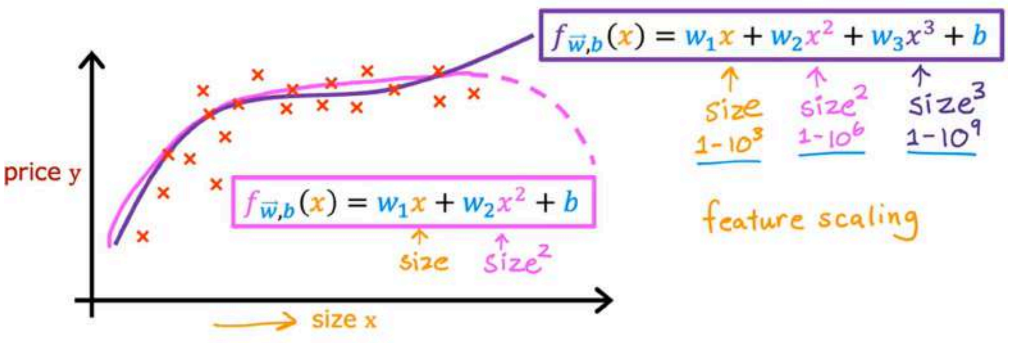
$x_3 = x_1 x_2$
new feature

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

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

here we don't just remove the features we used to make the new feature but here we add it so that the model can see how useful or not the 3 features are, let the model decide, just let all of them be there.

polynomial regression



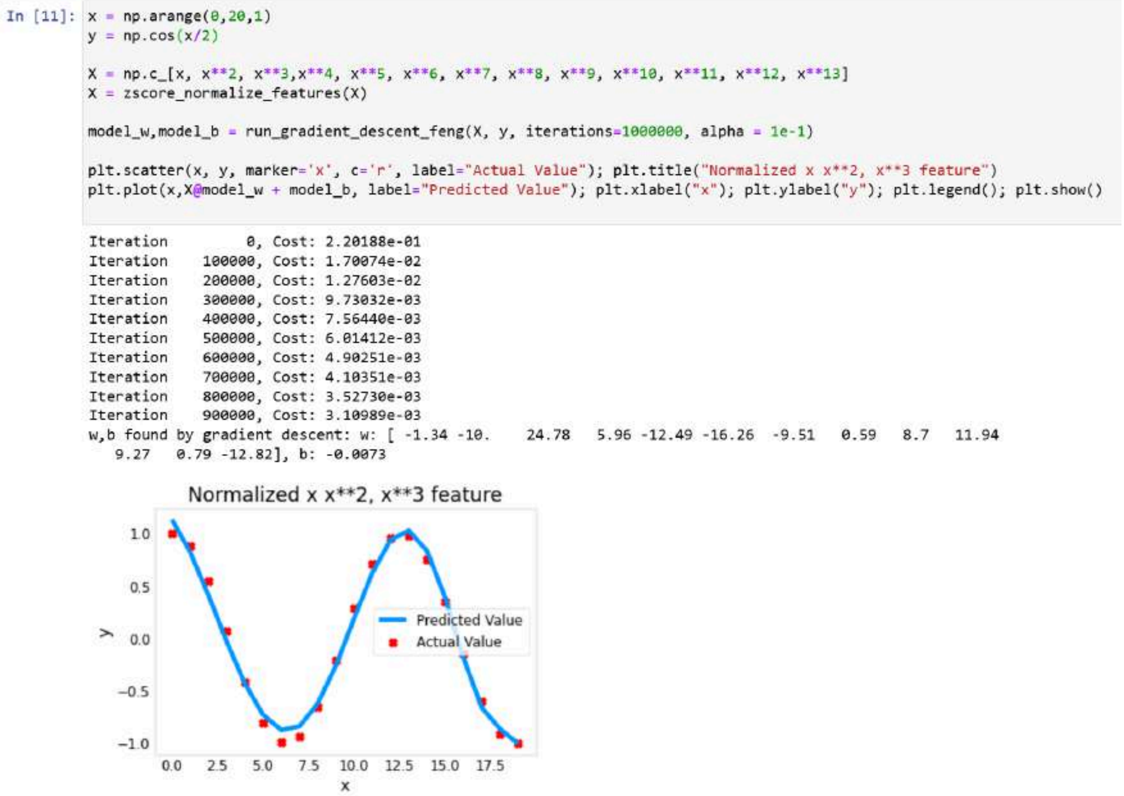
$f_{\vec{w},b}(x) = w_1x + w_2x^2 + w_3x^3 + b$
size
1-10^3
size^2
1-10^6
size^3
1-10^9
feature scaling
price y
$f_{\vec{w},b}(x) = w_1x + w_2x^2 + b$
size
size^2
size x

C1_W2_Lab04_FeatEng_PolyReg_Soln

```
x = np.array([1, 2, 3, 4])
X = x.reshape(-1, 1)
will give you:
[[1],
[2],
[3],
[4]]
### Breakdown:
- `-1` means "figure out the right number of rows automatically."
- `1` means "make it 1 column."
- This format (`n x 1`) is especially useful when feeding data into machine learning models like `sklearn`, which expect a 2D array of features.


x = np.arange(0, 20, 1)
will give you:
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9,
       10, 11, 12, 13, 14, 15, 16, 17, 18, 19])
0-start, 20-stop, 1-step
```

in feature engineering, we may have an intuition about what features may be useful or not, giving those features to GD will help us know if they impact the end result more or less and whether they are imp or not. basically GD will tell us whether the new features we engineered are good or bad but giving is a low or high weight.

it is important to do normalization when applying feature engineering. eg. x,x2, x3 need to be normalized as they would have varying scale



```
In [11]: x = np.arange(0,20,1)
         y = np.cos(x/2)

         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: 2.20188e-01
Iteration    100000, Cost: 1.70074e-02
Iteration    200000, Cost: 1.27603e-02
Iteration    300000, Cost: 9.73032e-03
Iteration    400000, Cost: 7.56440e-03
Iteration    500000, Cost: 6.01412e-03
Iteration    600000, Cost: 4.90251e-03
Iteration    700000, Cost: 4.10351e-03
Iteration    800000, Cost: 3.52730e-03
Iteration    900000, Cost: 3.10989e-03
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
```

C1_W2_Lab05_Sklearn_GD_Soln

```
"""code to implement linear regression using GD using scikit-learn"""
import numpy as np
import matplotlib.pyplot as plt
from sklearn.linear_model import SGDRegressor
from sklearn.preprocessing import StandardScaler
from lab_utils_multi import  load_house_data
from lab_utils_common import dlc
np.set_printoptions(precision=2)
plt.style.use('./deeplearning.mplstyle')

"""
- Scikit-learn has a gradient descent regression model [sklearn.linear_model.SGDRegressor] Like your previous implementation of
gradient descent, this model performs best with normalized inputs.
- [sklearn.preprocessing.StandardScaler] will perform z-score normalization as in a previous lab. Here it is referred to as
'standard score'.
"""

X_train, y_train = load_house_data()
X_features = ['size(sqft)','bedrooms','floors','age']
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)}")

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_}")

b_norm = sgdr.intercept_
w_norm = sgdr.coef_
print(f"model parameters:                   w: {w_norm}, b:{b_norm}")
# both below give same output, 2 diff ways of doing
# 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
```

check documentation of scikit here - scikit-learn: machine learning in Python — scikit-learn 1.6.1 documentation user guide or api

C1_W2_Linear_Regression

1-D numpy array's shape gives - (m,)
2-D numpy array's shape gives - (m,n)

Visualize your data

It is often useful to understand the data by visualizing it.

- For this dataset, you can use a scatter plot to visualize the data, since it has only two properties to plot (profit and population).
- Many other problems that you will encounter in real life have more than two properties (for example, population, average household income, monthly profits, monthly sales).When you have more than two properties, you can still use a scatter plot to see the relationship between each pair of properties.