Machine learning

linear Regression

Morteza khorsand

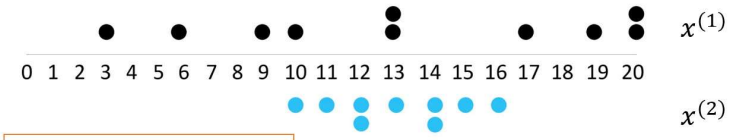
Linear regression

Mean

$$\mu = \bar{x} = \frac{x_{1+}\, x_{2+}\, x_{3+}\, \ldots x_n}{n} = \frac{1}{n}\sum_{i=1}^{n}(x_i)$$

$x^{(1)}$= [6,20, 20,9,17,13,3,10, 13 , 19]

$x^{(2)}$= [10,11,13,13,16 , 14 , 12, 12, 15 , 14]



$$\sigma^2 = \frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2$$ variance

$$\sigma = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})^2}$$ standard deviation

$$s^2 = \frac{1}{n-1}\sum_{i=1}^{n}(x_i - \bar{x})^2$$

In [19]:
```
x1=[6,20,20,9,17,13,3,10,13,19]
x2=[10,11,13,13,16,14,12,12,15,14]

mu1= sum(x1)/len(x1)
mu2= sum(x2) / len(x2)

print(mu1 ,"....." ,mu2)
```

```
13.0 ..... 13.0
```

In [20]:
```
#import math library
from math import sqrt

var=0

for i in range(len(x1)):
    var+=(x1[i]- mu1)**2

var*= 1/len(x1)
print(".....................")
std=round(sqrt(var) , 2)

display(var , std)
```

```
.....................
32.4
5.69
```

In [21]:
```
var2=0
for i in range(len(x2)):
    var2+=(x2[i]- mu2)**2

var2*= 1/len(x2)
print(".....................")
sqrt2=round(sqrt(var2) , 2)

display(var2 , sqrt2)
```

```
.....................
3.0
1.73
```

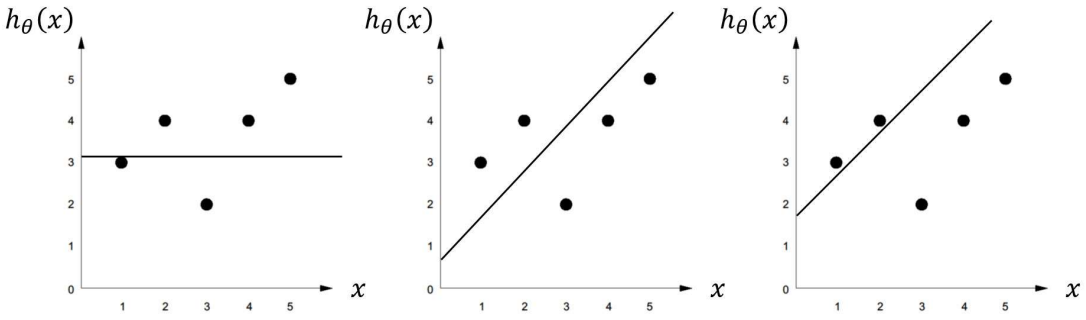
Linear Regression

Y= C + m X

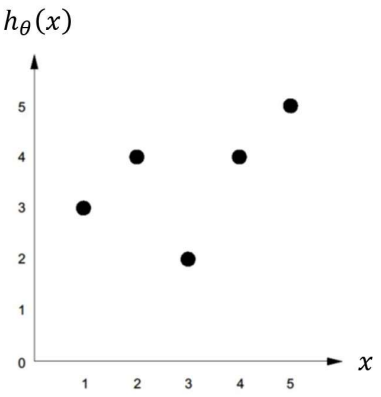
$$h_\theta(x) = \theta_0 + \theta_1 x$$

$h_\theta(x) = 0 + x$ $h_\theta(x) = 1 + 0.5\ x$ $h_\theta(x) = 2 + (-2)\ x$

$$h_\theta(x) = \theta_0 + \theta_1 x$$



$h_\theta(x) = 3.1 + 0\ x$ $h_\theta(x) = 0.75 + 1\ x$ $h_\theta(x) = 1.8 + 1\ x$



X	y
1	3
2	4
3	2
4	4
5	5
$\bar{x}$ = 3	$\bar{y}$ = 3.6

$$h_\theta(x) = \theta_0 + \theta_1 x$$

$$\theta_1 = \frac{\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^{n}(x_i - \bar{x})^2}$$

analytical method

SCAN ME
https://qrco.de/bdPcj1

In [22]:
```
X = [1, 2, 3,4,5]
y = [3,4,2,4,5]

x_bar= sum(X) / len(X)
y_bar = sum(y)/ len(y)
print(f"y_bar={y_bar} and  x_bar={x_bar}")

numerator=0
Denominator=0

for i in range(len(X)):
    numerator+= (X[i]- x_bar )* (y[i] - y_bar)
    Denominator+= (X[i] - x_bar) **2
theta_1=numerator/Denominator
theta_0= y_bar- (theta_1* x_bar)
print(f" slope is {theta_1} , intercept is {theta_0}")
```

```
y_bar=3.6 and  x_bar=3.0
 slope is 0.4 , intercept is 2.4
```

Linear regression 6

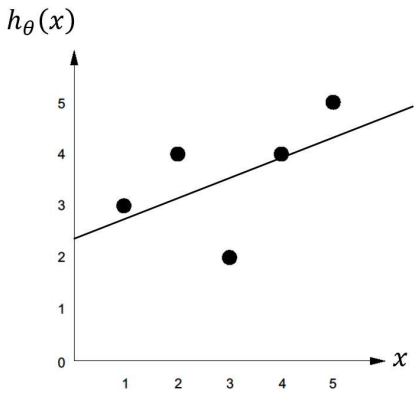
Model Evaluation

COST FUNCTION

Mean Absolute Error $\text{MAE} = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$

Mean *Square* Error $\text{MSE} = \frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$

Root *Mean Square* Error $\text{RMSE} = \sqrt{\frac{1}{2n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$

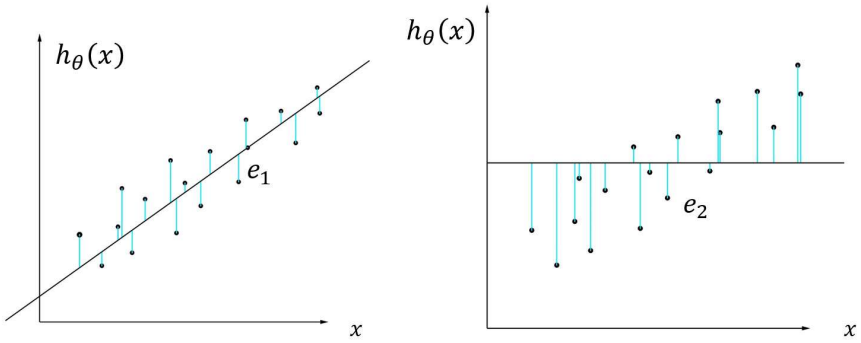


Linear regression 7

Relative Absolute Error – Residual Sum of Squre $\text{RAE} = \frac{\sum_{i=1}^{n}|y_i - \hat{y}_i|}{\sum_{i=1}^{n}|y_i - \bar{y}|}$

Relative Squre Error $\text{RSE} = \frac{\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}{\sum_{i=1}^{n}(y_i - \bar{y})^2} = \frac{{e_1}^2}{{e_2}^2}$

R^2 = 1- RSE



In [23]:
```
#y_hat
y_hat=[]
```

```
for i in range(len(X)):
    y_hat.append(X[i]* theta_1  + theta_0)
print(y_hat)
```

```
[2.8, 3.2, 3.6, 4.0, 4.4]
```

In [24]:
```
#calculating mean absolute error

MAE=0

for num in range(len(y)):
    MAE+= abs (y[i] - y_hat[i]) / len(y)

print(round(MAE,2) * 100)
```

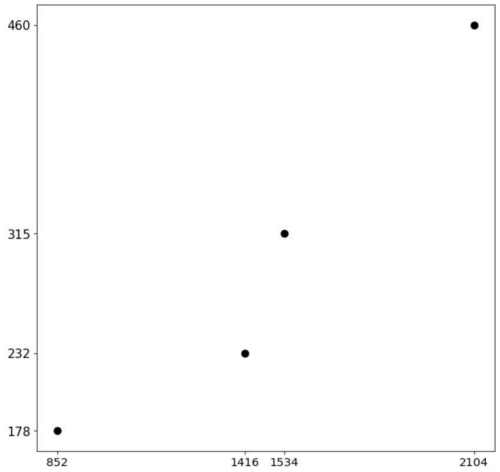
```
60.0
```

In [25]:
```
#calculating relative squre error



numerator=0
Denominator=0
for i in range(len(y)):
    numerator+= (y[i]- y_hat[i] )**2
    Denominator+= (y[i] - y_bar) **2
rse= numerator/Denominator

r_squre= 1-rse

print(f" the relative squre erroe is  {round(rse, 2)} , and R_squred is {round(r_squre
```

```
 the relative squre erroe is  0.69 , and R_squred is 0.31
```

SIZE (foot^2)	PRICE (1000$)
2104	460
1416	232
1534	315
852	178

$$h_\theta(x) = \theta_0 + \theta_1 x$$



? *pridict a house with* 1200 *foot squre*

In [26]:
```
X= [852, 1416, 1532, 2104]
y= [178, 232, 315, 460]

x_bar= sum(X) / len(X)
y_bar= sum(y)/ len(y)

display(f" X_bar is:{x_bar} , y_bar is:{y_bar}")
```

```
' X_bar is:1476.0 , y_bar is:296.25'
```

In [27]:
```
numerator=0
Denominator=0
for i in range(len(X)):
    numerator+= (X[i]- x_bar )* (y[i] - y_bar)
    Denominator+= (X[i] - x_bar) **2
w=numerator/Denominator
b= y_bar- (w* x_bar)
print(f" slope is {w} , intercept is {b}")
```

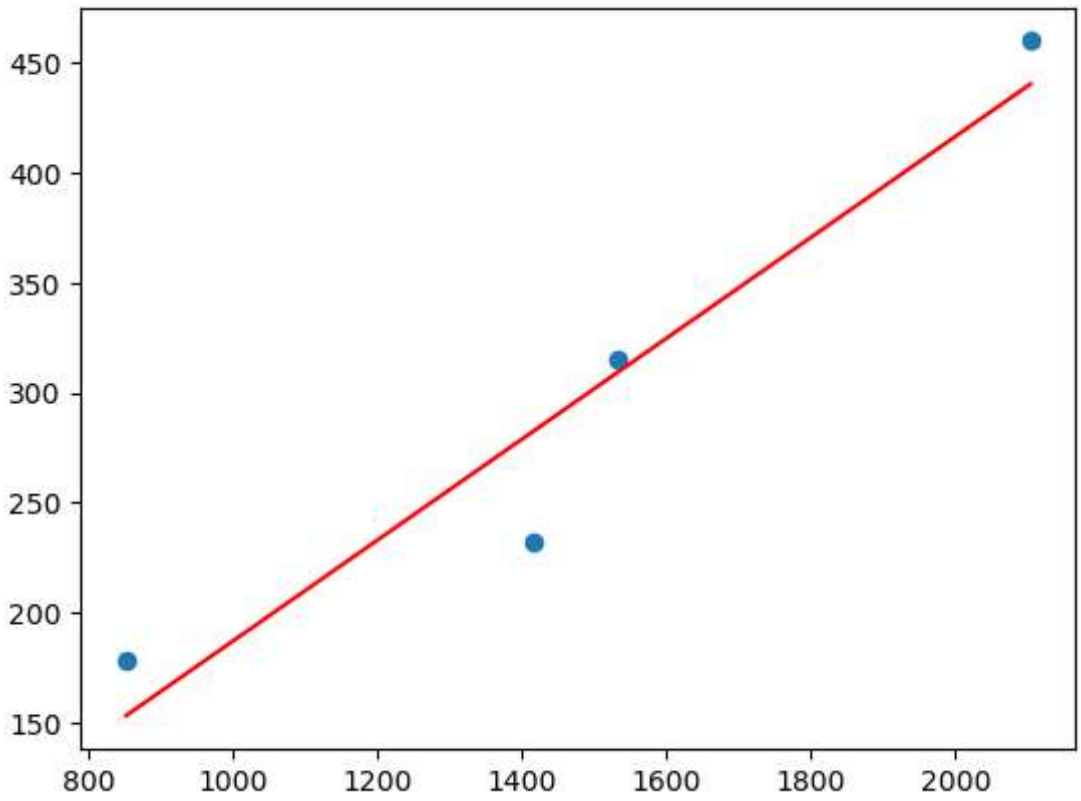
```
 slope is 0.22963810063555035 , intercept is -42.695836538072285
```

In [28]:
```
#y_hat = y_prediction
y_hat=[]
for i in range(len(X)):
    y_hat.append(X[i]* w  + b)
print(y_hat)
```

```
[152.95582520341662, 282.471713961867, 309.1097336355908, 440.46272719912565]
```

In [29]:
```
#plot
import matplotlib.pyplot as plt
```

```
plt.scatter(X ,y)
plt.plot(X ,y_hat , c= "red")
plt.show()
```



In [30]:
```
y_predicted= w*5000+b
print(y_predicted)
```

```
1105.4946666396795
```

In [31]:
```
#calculating relative squre error

numerator=0
Denominator=0
for i in range(len(y)):
    numerator+= (y[i]- y_hat[i] )**2
    Denominator+= (y[i] - y_bar) **2
rse= numerator/Denominator

r_squre= 1-rse

print(f" the relative squre erroe is  {round(rse , 2)} , and R_squre is {round(r_squre
```

```
 the relative squre erroe is  0.08 , and R_squre is 0.92
```

Matrix

$$A = \begin{bmatrix} a_{1\,1} & a_{1\,2} & \cdot & a_{1\,n} \\ a_{2\,1} & a_{2\,2} & \cdot & a_{2\,n} \\ a_{3\,1} & a_{3\,2} & \cdot & a_{3\,n} \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ a_{m\,1} & a_{m\,2} & \cdot & a_{m\,n} \end{bmatrix}$$

$A = [a_{ij}]_{m \times n}$

Size (feet^2)	#bedrooms	#floors	age	Price(1000$)
2104	5	1	45	460
1416	3	2	40	232
1534	3	2	30	315
852	2	1	36	178
...	...	...	...	...

$$X = \begin{bmatrix} 2104 & 5 & 1 & 45 \\ 1416 & 3 & 2 & 40 \\ 1534 & 3 & 2 & 30 \\ 852 & 2 & 1 & 36 \end{bmatrix}$$

$$x^{(2)} = \begin{bmatrix} 1416 \\ 3 \\ 2 \\ 40 \end{bmatrix} \begin{matrix} \leftarrow x_1^{(2)} \\ \\ \leftarrow x_3^{(2)} \\ \\ \end{matrix}$$

- n تعداد ویژگی‌ها
- $x^{(i)}$ ورودی‌ها در i امین نمونه آموزشی
- $x_j^{(i)}$ مقدار ویژگی j ام در i امین نمونه آموزشی

Main types

Square matrix: a matrix with the same number of rows and columns.

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

Upper triangular matrix: all entries, below the main diagonal, are zero.

$$A = \begin{bmatrix} 1 & 4 & 7 \\ 0 & 5 & 8 \\ 0 & 0 & 9 \end{bmatrix}$$

Lower triangular matrix: all entries of *A* above the main diagonal are zero.

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 2 & 5 & 0 \\ 3 & 6 & 9 \end{bmatrix}$$

Diagonal matrix: all the entries outside the main diagonal are all zero

$$A = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 5 & 0 \\ 0 & 0 & 9 \end{bmatrix}$$

$$A = \begin{bmatrix} 0 & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 0 \end{bmatrix}$$

Scalar matrix: a square matrix having a constant value for all the elements of the principal diagonal, and the other elements of the matrix are zero

$$A=\begin{bmatrix}3 & 0 & 0\\ 0 & 3 & 0\\ 0 & 0 & 3\end{bmatrix}$$

Identity matrix: all the elements on the main diagonal are equal to 1 and all other elements are equal to 0.

$$I_3=\begin{bmatrix}1 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{bmatrix}$$

zero matrix (null matrix) : is a matrix all of whose entries are zero

$$O_{3\times 2}=\begin{bmatrix}0 & 0\\ 0 & 0\\ 0 & 0\end{bmatrix}$$

transpose of a matrix: an operator which flips a matrix over its diagonal

$$A_{3\times 4}=\begin{bmatrix}1 & 4 & 7 & 10\\ 2 & 5 & 8 & 11\\ 3 & 6 & 9 & 12\end{bmatrix}$$

$A=[1,2,3\ldots,n]_{1\times n}$ $\quad A^T=\begin{bmatrix}1\\2\\3\\.\\.\\.\\n\end{bmatrix}_{n\times 1}$ $\quad$ A $\begin{bmatrix}1 & 2\\ 3 & 4\\ 5 & 6\end{bmatrix}$

$$A'=A^T_{4\times 3}=\begin{bmatrix}1 & 2 & 3\\ 4 & 3 & 6\\ 7 & 8 & 9\\ 10 & 11 & 12\end{bmatrix}$$

In [32]:
```
import numpy as np

list_1 = ([1,2,3,4])

matrix_1 = np.array(list_1)

print(matrix_1)

print(np.shape(matrix_1))
```

```
[1 2 3 4]
(4,)
```

In [33]:
```
A = np.array ([[1,4,7],[2,5,8],[3,6,9]])

print(A)

print(np.shape(A))        #A.shape

print(A.ndim)
```

```
[[1 4 7]
 [2 5 8]
 [3 6 9]]
(3, 3)
2
```

In [34]:
```
ones=np.ones((3,3))
print(ones)
```

```
[[1. 1. 1.]
 [1. 1. 1.]
 [1. 1. 1.]]
```

In [35]:
```
eye=np.eye(3)    #identity matrix
print(eye)
```

```
[[1. 0. 0.]
 [0. 1. 0.]
 [0. 0. 1.]]
```

In [36]:
```
import numpy as np
```

In [37]:
```
print(3*eye )            #scalar matrix
```

```
[[3. 0. 0.]
 [0. 3. 0.]
 [0. 0. 3.]]
```

In [38]:
```
print(np.zeros((3,4)))
```

```
[[0. 0. 0. 0.]
 [0. 0. 0. 0.]
 [0. 0. 0. 0.]]
```

In [39]:
```
p= np.array([[3,3,5],
             [5,7,9]])
print(p)

e=np.transpose(p)       #p.transpose
print(e)
np.shape(e)
```

```
[[3 3 5]
 [5 7 9]]
[[3 5]
 [3 7]
 [5 9]]
```

Out[39]:
```
(3, 2)
```

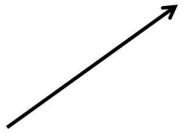
In [40]:
```
p@e
```

Out[40]:
```
array([[ 43,  81],
       [ 81, 155]])
```

In [41]:
```
y=np.array([[1,2,3]])
k=np.transpose(y)

print(y@k)

print(np.sum(y**2))
```

```
[[14]]
14
```

Vector Representation



$\vec{v}$

$$\begin{bmatrix} 4 \\ 5 \\ -1 \end{bmatrix}$$

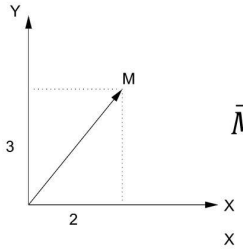
Physics approach
Length – Direction

Mathematic approach

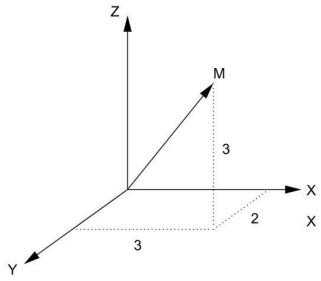
Computer science approach
List of numbers

Row vector: A matrix with one row, sometimes used to represent a vector

Column vector: A matrix with one column, sometimes used to represent a vector

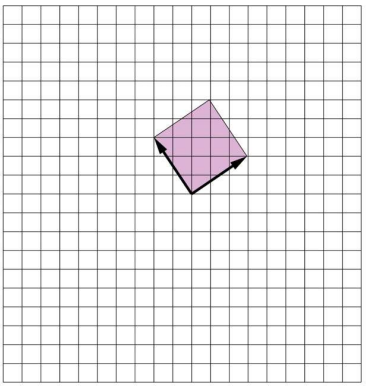


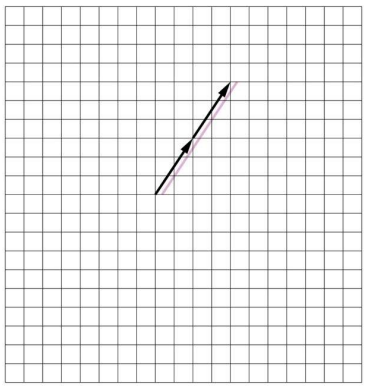
$\vec{M} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$

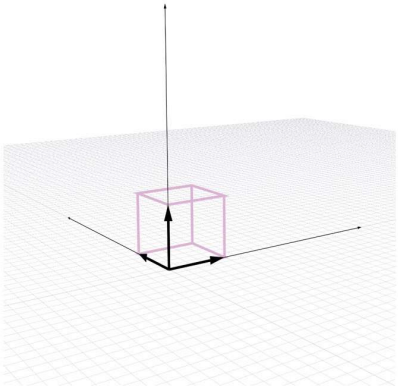


$\vec{M} = \begin{bmatrix} 3 \\ 2 \\ 3 \end{bmatrix}$

- **Span**: all the available spaces is been created by vectors.

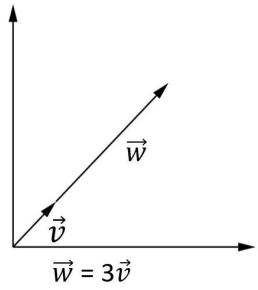


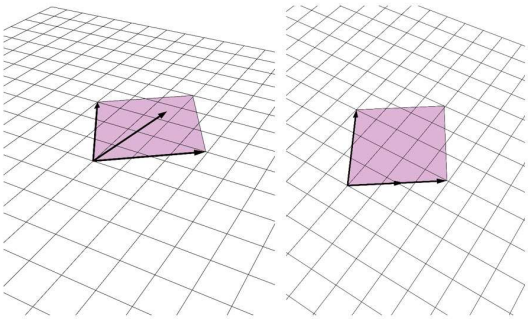




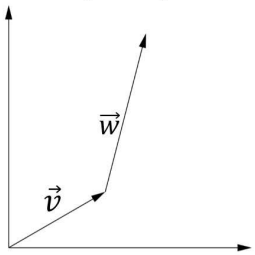
https://www.geogebra.org/m/VHQzeMX9

Linearly dependent vector :In the theory of vector spaces, a set of vectors is said to be linearly dependent if there is a nontrivial linear combination of the vectors that equals the zero





Linearly independent



- **linear transformation**

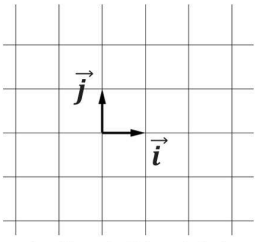
The matrix of a linear transformation is a matrix for which

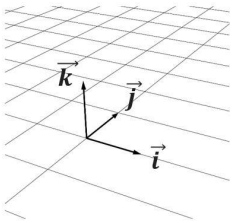
$\mathbf{T}(\vec{x})=\mathbf{A}\,\vec{x}$, for a vector $\vec{x}$ in the domain of T.

This means that applying the transformation T to a vector is the same as multiplying by this matrix

- **The standard unit vectors**: The standard unit vectors in two dimensions, i and j are length one vectors that point parallel to the x-axis and y-axis, respectively.

The standard unit vectors in three dimensions, i, j, and k are length one vectors that point parallel to the x-axis, y-axis, and z-axis respectively.

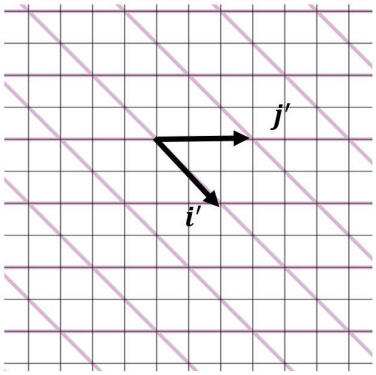




- **Changing the standard unit vectors**

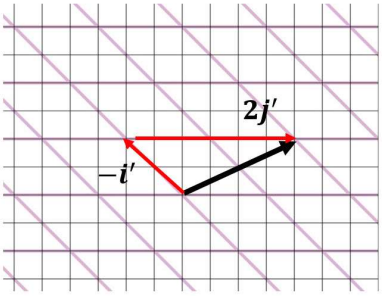
$j' = \begin{bmatrix} 3 \\ 0 \end{bmatrix}$

$i' = \begin{bmatrix} 2 \\ -2 \end{bmatrix}$



? Specify $\vec{v} = \begin{bmatrix} -1 \\ 2 \end{bmatrix}$ in new span

$$\begin{bmatrix} 2 & 3 \\ -2 & 0 \end{bmatrix} \begin{bmatrix} -1 \\ 2 \end{bmatrix} = \begin{bmatrix} 2 \\ 2 \end{bmatrix}$$

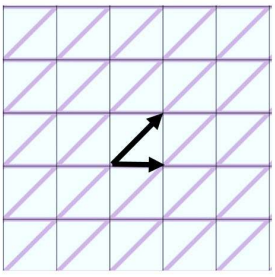


- **Shear matrix**: In mathematics, a shear matrix or transvector is an elementary matrix that represents the addition of a multiple of one row or column to another

$$S = \begin{pmatrix} 1 & 0 & 0 & \lambda & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix}.$$

? Specify $\vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ after shear transformation by $\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

$$\begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 \\ 2 \end{bmatrix} = \begin{bmatrix} 3 \\ 2 \end{bmatrix}$$



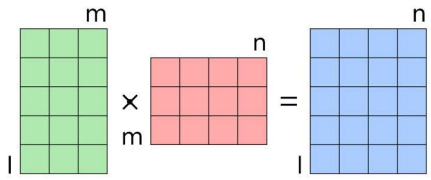
- **Identity matrix**

? Specify $\vec{v} = \begin{bmatrix} 1 \\ 2 \end{bmatrix}$ after transformation by $\begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$

$A \times I = I \times A = A$

Multiplication

For matrix multiplication, the number of columns in the first matrix must be equal to the number of rows in the second matrix. The resulting matrix, known as the **matrix product**, has the number of rows of the first and the number of columns of the second matrix. The product of matrices **A** and **B** is denoted as **AB**.



$$[\mathbf{AB}]_{i,j} = a_{i,1}b_{1,j} + a_{i,2}b_{2,j} + \cdots + a_{i,n}b_{n,j} = \sum_{r=1}^{n} a_{i,r}b_{r,j},$$

? A= $\begin{bmatrix} 1 & 2 & 3 \\ 4 & -5 & 6 \end{bmatrix}$ B= $\begin{bmatrix} 7 & 8 \\ 0 & 9 \end{bmatrix}$ A × B B × A

A= $\begin{bmatrix} 1 & 5 & -2 \\ 1 & 2 & -1 \\ 3 & 6 & -3 \end{bmatrix}$ B= $\begin{bmatrix} 0 & 3 & -1 \\ 0 & 3 & -1 \\ 0 & 9 & -3 \end{bmatrix}$ A × B

In [42]:
```
j= np.array([[2,3],[4,5],[6,7]])
p=np.array([[2,1], [3,0], [7,5]])

print(j)
print(".......")
print(p)
```

```
[[2 3]
 [4 5]
 [6 7]]
.......
[[2 1]
 [3 0]
 [7 5]]
```

In [43]:
```
j@p
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35516\256439442.py in <module>
----> 1 j@p

ValueError: matmul: Input operand 1 has a mismatch in its core dimension 0, with gufu
nc signature (n?,k),(k,m?)->(n?,m?) (size 3 is different from 2)
```

In [44]:
```
print(j.shape)
print(p.shape)
```

```
(3, 2)
(3, 2)
```

In [45]:
```
k= np.array([[2,3],[4,5],[6,7]])
m= np.array([[3,6,8],[7,0,1]])
#k@m
np.dot(k,m)
```

Out[45]:
```
array([[27, 12, 19],
       [47, 24, 37],
       [67, 36, 55]])
```

In [46]:
```
A= np.array([[1,5,-2],
             [1,2,-1],
             [3,6,-3]])

B=np.array([[0,3,-1],
            [0,3, -1],
            [0,9,-3]])

A@B
```

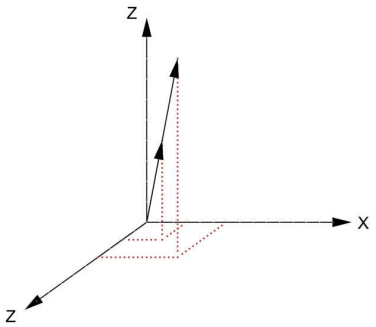
Out[46]:
```
array([[0, 0, 0],
       [0, 0, 0],
       [0, 0, 0]])
```

Linear regression

Scalar multiplication

matrix A is computed by multiplying every entry of A by k:

K .A = A.K = K $\left[a_{ij}\right]_{m\times n} = \left[ka_{ij}\right]_{m\times n}$



A= $\begin{bmatrix} 3 \\ 2 \\ 5 \end{bmatrix}$

2A =

Power of a Matrix

If A is a square matrix and k is a positive integer, the k power of A is given by $A = A \times A \times \cdots \times A$, where there are k copies of matrix A

$\boldsymbol{A_{n\times n} \times A_{n\times n} = A^2}$

? A= $\begin{bmatrix} 1 & -2 & 2 \\ 0 & 2 & 0 \\ 1 & -1 & -3 \end{bmatrix}$ A^2

A × A^2 =A^3

In [47]:
```
import numpy as np
```

In [48]:
```
k= np.array([[2,3], [4,5] , [6,7]])
np.shape(k)   #k.shape
```

Out[48]:
```
(3, 2)
```

In [49]:
```
print(k)
2*k
```

```
[[2 3]
 [4 5]
 [6 7]]
```

Out[49]:
```
array([[ 4,  6],
       [ 8, 10],
       [12, 14]])
```

In [50]:
```
k**2
```

Out[50]:
```
array([[ 4,  9],
       [16, 25],
       [36, 49]], dtype=int32)
```

In [51]:
```
1/k
```

Out[51]:
```
array([[0.5       , 0.33333333],
       [0.25      , 0.2       ],
       [0.16666667, 0.14285714]])
```

In [52]:
```
A_mat=np.array([[1,-2,2],
                [0,2,0],
                [1,-1,-3]])

A_mat@A_mat
```

Out[52]:
```
array([[ 3, -8, -4],
       [ 0,  4,  0],
       [-2, -1, 11]])
```

In [53]:
```
A=np.array([[2,3,4]])
B= np.array([[5,6,-7]])

A*B            #np.multiply(j, p)
```

Out[53]:
```
array([[ 10,  18, -28]])
```

In [54]:
```
A_mat=np.array([[1,-2,2],
                [0,2,0],
                [1,-1,-3]])
```

In [55]:
```
np.sum(A_mat)
```

Out[55]:
```
0
```

In [ ]:
```
#broadcating
```

In [56]:
```
d= np.array([[3,4,5],
             [7,10,15],
             [0,5,4]])
```

In [57]:
```
3+d
```

Out[57]:
```
array([[ 6,  7,  8],
       [10, 13, 18],
       [ 3,  8,  7]])
```

In [58]:
```
a= np.ones((3,3))
b=np.array([[4,2,7]])
```

In [59]:
```
a+b
```

Out[59]:
```
array([[5., 3., 8.],
       [5., 3., 8.],
       [5., 3., 8.]])
```

In [60]:
```
s=np.array([[2,3],
            [4,5]])

L= np.ones_like(s)
L
```

Out[60]:
```
array([[1, 1],
       [1, 1]])
```

In [61]:
```
np.zeros_like(s)
```

Out[61]:
```
array([[0, 0],
       [0, 0]])
```

In [62]:
```
A= np.array([[8,1,6],        #Magic squre
             [3,5,7],
             [4,9,2]])
```

In [63]:
```
A.max()
```

Out[63]:
```
9
```

In [64]:
```
A.argmax()    # index of maximum number
```

Out[64]:
```
7
```

In [65]:
```
A.max(axis=0)
```

Out[65]:
```
array([8, 9, 7])
```

In [66]:
```
A>3
```

Out[66]:
```
array([[ True, False,  True],
       [False,  True,  True],
       [ True,  True, False]])
```

In [67]:
```
# Colon  :
```

In [68]:
```
A= np.array([[8,1,6],        #Magic squre
             [3,5,7],
             [4,9,2]])
```

In [69]:
```
A [2,1]
```

Out[69]:
```
9
```

In [70]:
```
A[ : , 2]
```

Out[70]:
```
array([6, 7, 2])
```

In [71]:
```
A[0:2 , : ]
```

Out[71]:
```
array([[8, 1, 6],
       [3, 5, 7]])
```

In [72]:
```
# CONCATINATE
```

In [73]:
```
A= np.array([[8,1,6],        #Magic squre
             [3,5,7],
             [4,9,2]])

v= np.array([[9] , [7], [0]])

v=np.transpose(v)
```

In [74]:
```
np.concatenate((A , v) , axis = 0)     #hstack and vstack == concatinate    np.stack(
```

Out[74]:
```
array([[8, 1, 6],
       [3, 5, 7],
       [4, 9, 2],
       [9, 7, 0]])
```

In [ ]:
```
# FLATTEN
```

In [75]:
```
#function in nural networks
B=np.array([[1,2], [3,4], [5,6]])
B.shape
print(B)
```

```
[[1 2]
 [3 4]
 [5 6]]
```

In [76]:
```
C= B.ravel(order = "c")          #c language
D=B.ravel(order= "F")            #fortran language
print(D)
```

```
[1 3 5 2 4 6]
```

In [77]:
```
import numpy as np
```

In [78]:
```
#unique

#numpy.unique(ar, return_index=False, return_inverse=False, return_counts=False, axis=

a=np.unique([1, 1, 2, 2, 3, 3])
print(a)


b = np.array([[1, 1], [2, 3]])
c=np.unique(b)


print(a.shape, "-----"  , c.shape)
```

```
[1 2 3]
(3,) ----- (3,)
```

In [79]:
```
a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
np.unique(a, axis=0)
```

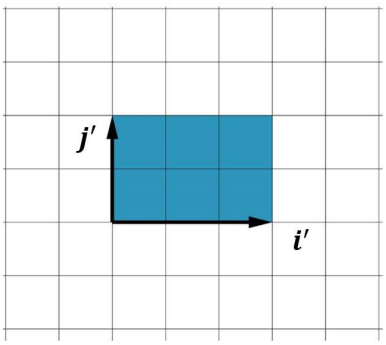
Out[79]:
```
array([[1, 0, 0],
       [2, 3, 4]])
```

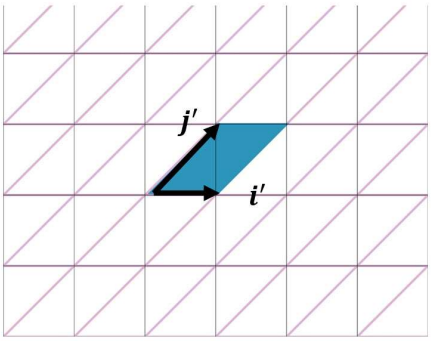
In [80]:
```
a = np.array([[1, 0, 0], [1, 0, 0], [2, 3, 4]])
np.unique(a, axis=1)
```

Out[80]:
```
array([[0, 0, 1],
       [0, 0, 1],
       [3, 4, 2]])
```


- **Determinant**: a scalar value that is a function of the entries of a square matrix.

Area between i , j





$A = \begin{bmatrix} 3 & 0 \\ 0 & 2 \end{bmatrix}$

$|A| = 6$

$B = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}$

$|A| = 1$

Laplace expansion

$$|B| \text{ or } \det(B) = \sum_{j=1}^{n} (-1)^{i+j} B_{i,j} M_{i,j},$$

Where $B_{i,j}$ is the entry of the *i*th row and *j*th column of *B*, and $M_{i,j}$ is the determinant of the submatrix obtained by removing the *i*th row and the *j*th column of *B*. The term $(-1)^{i+j} M_{i,j}$ is called the **cofactor** of $B_{i,j}$ in B

Cofactor

$$\sum_{k=1}^{n} a_{ik}(-1)^{i+k} M_{ik} = \sum^{n} a_{kj} (-1)^{k+j} M_{kj}$$

In linear algebra, a minor of a matrix A is the determinant of some smaller square matrix, cut down from A by removing one or more of its rows and columns. Minors obtained by removing just one row and one column from square matrices (first minors) are required for calculating matrix cofactors, which in turn are useful for computing both the determinant and inverse of square matrices

$$A = \begin{bmatrix} 2 & 3 \\ 5 & 4 \end{bmatrix}$$

$$|A| = (2 \times 4) - (3 * 5)$$

$$B = \begin{bmatrix} 2 & 3 & 5 \\ 1 & 0 & 0 \\ 2 & 1 & 0 \end{bmatrix}$$

$$|B| = 2\begin{vmatrix} 0 & 0 \\ 1 & 0 \end{vmatrix} - 3\begin{vmatrix} 1 & 0 \\ 2 & 0 \end{vmatrix} + 5\begin{vmatrix} 1 & 0 \\ 2 & 1 \end{vmatrix} = 2(0-0) - 3(0-0) + 5(1-0) = 5$$

Properties of Determinants

1.All-zero Property

If all the elements of a row (or column) are zero, then the determinant is zero.

In [81]:
```
B= np.array([[2,3,5],
             [1,0,0],
```

```
            [2,1,0]])

# np.linalg

print(round(np.linalg.det(B)))
```

```
5
```


2 .Reflection Property

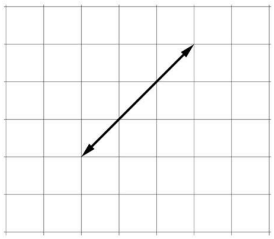
The determinant remains unaltered if its rows are changed into columns and the columns into rows. This is known as the property of reflection

$$|A| = |A'|$$

3. Proportionality (Repetition) Property

If all elements of a row (or column) are proportional (identical) to the elements of some other row (or column), then the determinant is zero.

$$|A| = \begin{vmatrix} a & ka & d \\ b & kb & e \\ c & kc & f \end{vmatrix} = 0$$



$$A = \begin{bmatrix} 2 & -1 \\ 2 & -1 \end{bmatrix}$$

$|A| = 0$

area=0

Matrix Inverse

The inverse of a square matrix A, sometimes called a reciprocal matrix, is a matrix A^{-1} ($\check{A}$) such that:

$A.A^{-1} = A^{-1}.A = I_n$

A square matrix A has an inverse iff the determinant |A| !=0

Calculate matrix inverse

Adjoint of a matrix

adjA = $[\,(-1)^{j+i}\,|A_{ji}|\,]$

$$A^{-1} = \frac{1}{|A|}\,.\text{adj A}$$

$$A = \begin{bmatrix}1 & 0 & 2\\ 2 & 1 & 0\\ 3 & 2 & 1\end{bmatrix} \quad A' = \begin{bmatrix}1 & 2 & 3\\ 0 & 1 & 2\\ 2 & 0 & 1\end{bmatrix} \quad \text{adjA} = \begin{bmatrix}1 & 4 & -2\\ -2 & -5 & 4\\ 1 & -2 & 1\end{bmatrix} \quad |A| = 3 \quad A^{-1} = \begin{bmatrix}1/3 & 4/3 & -2/3\\ -2/3 & -5/3 & 4/3\\ 1/3 & -2/3 & 1/3\end{bmatrix}$$

$$B = \begin{bmatrix}1 & 3 & 3\\ 1 & 4 & 3\\ 1 & 3 & 4\end{bmatrix} \quad B^{-1} = \begin{bmatrix}7 & -3 & -3\\ -1 & 1 & 0\\ -1 & 0 & 1\end{bmatrix}$$

In [82]:
```
import numpy as np
A= np.array([[1,0,2],
             [2,1,0],
             [3,2,1]])

display(np.around(np.linalg.inv(A), decimals=1))
```

```
array([[ 0.3,  1.3, -0.7],
       [-0.7, -1.7,  1.3],
       [ 0.3, -0.7,  0.3]])
```

In [83]:
```
D= np.array([[2,3,6],
             [5,6,9]])

np.linalg.inv(D)
```

```
---------------------------------------------------------------------------
LinAlgError                               Traceback (most recent call last)
~\AppData\Local\Temp\ipykernel_35516\2216543689.py in <module>
      2            [5,6,9]])
      3
----> 4 np.linalg.inv(D)

<__array_function__ internals> in inv(*args, **kwargs)

C:\ProgramData\Anaconda3\lib\site-packages\numpy\linalg\linalg.py in inv(a)
    538     a, wrap = _makearray(a)
    539     _assert_stacked_2d(a)
--> 540     _assert_stacked_square(a)
    541     t, result_t = _commonType(a)
    542

C:\ProgramData\Anaconda3\lib\site-packages\numpy\linalg\linalg.py in _assert_stacked_
square(*arrays)
    201         m, n = a.shape[-2:]
    202         if m != n:
--> 203             raise LinAlgError('Last 2 dimensions of the array must be squar
e')
    204
    205 def _assert_finite(*arrays):

LinAlgError: Last 2 dimensions of the array must be square
```

In [84]:

```
dataset= np.array([[2104, 5,1,45],
                   [1416,3,2,40],
                   [1532,3,2,30],
                   [852,2,1,36]])

display(np.linalg.det(dataset))

display(np.linalg.inv(dataset))
```

```
6211.9999999999945
array([[-1.60978751e-03, -1.88345138e-02,  1.40051513e-02,
         1.12685126e-02],
       [ 1.06117193e+00,  8.21571153e+00, -6.03219575e+00,
        -5.42820348e+00],
       [-7.85576304e-02,  3.88087572e+00, -2.11654862e+00,
        -2.45009659e+00],
       [-1.86735351e-02, -1.18480361e-01,  6.24597553e-02,
         1.30714746e-01]])
```

Linear regression equation

$\theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \ldots + \theta_n x_n = y$

$4x_0 + 6x_1 + 5x_2 + 4x_3 = 10$

$\theta_{n\times n} \cdot X_{n\times 1} = y_{n\times 1}$

$|\theta| \neq 0$

$(\theta^{-1} \cdot \theta) \cdot X = \theta^{-1} \cdot Y$

$I_n \cdot X = \theta^{-1} \cdot Y$

$X = \theta^{-1} \cdot Y$

? $\begin{cases} 2x_0 + 3x_1 + x_2 = 7 \\ x_0 + 2x_1 + 3x_2 = 9 \\ 3x_0 + x_1 + 2x_2 = 8 \end{cases}$

$\begin{bmatrix} 2 & 3 & 1 \\ 1 & 2 & 3 \\ 3 & 1 & 2 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 7 \\ 9 \\ 8 \end{bmatrix}$

$|\theta| = 18$

$X = \theta^{-1} \cdot Y$

$\theta^{-1} = \frac{1}{18} \begin{bmatrix} 1 & -5 & 7 \\ 7 & 1 & -5 \\ -5 & 7 & 1 \end{bmatrix}$

$X = \frac{1}{18} \begin{bmatrix} 1 & -5 & 7 \\ 7 & 1 & -5 \\ -5 & 7 & 1 \end{bmatrix} \begin{bmatrix} 7 \\ 9 \\ 8 \end{bmatrix} = \frac{1}{18} \begin{bmatrix} 18 \\ 18 \\ 36 \end{bmatrix} = \begin{bmatrix} 1 \\ 1 \\ 2 \end{bmatrix}$

Normal equation

In linear regression analysis, the normal equations are a system of equations whose solution is the Ordinary Least Squares (OLS) estimator of the regression coefficients

	Size (feet^2)	#Bedrooms	#Floors	Age (years)	Price(1000$)
x_0	x_1	x_2	x_3	x_4	y
1	2104	5	1	45	460
1	1416	3	2	40	232
1	1534	3	2	30	315
1	852	2	1	36	178
	...	...	...	...	...

$X = \begin{bmatrix} 1 & 2104 & 5 & 1 & 45 \\ 1 & 1416 & 3 & 2 & 40 \\ 1 & 1534 & 3 & 2 & 30 \\ 1 & 852 & 2 & 1 & 36 \end{bmatrix}$ $\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \theta_3 \\ \theta_4 \end{bmatrix}$ $y = \begin{bmatrix} 460 \\ 232 \\ 315 \\ 178 \end{bmatrix}$

$X\theta = y$ $\quad X^{-1}X\,\theta = X^{-1}y$ $\quad I\,\theta = X^{-1}y$ $\quad \theta = X^{-1}y$

X matrix is often noninvertible, so we rarely use this method

$X\theta = y$ $X^T X\,\theta = X^T y$ معادله نرمال $A\theta = B$ $A^{-1}A\theta = A^{-1}B$ $\theta = A^{-1}B$

$$\theta = (X^T X)^{-1} X^T y$$

$$\theta = X^{+} y$$

X^{+} شبه وارون
Pseudo inverse

Sometimes $X^T X$ is not invertible

Reason:
1. Redundancy (linearly dependent)
2. Having a lot of features.

Solution : removing some features

In [85]:
```
#numpy.linalg.det

R= np.array([[8,1,6],
             [3,5,7],
             [4,9,2]])

np.linalg.pinv(R)
```

Out[85]:
```
array([[ 0.14722222, -0.14444444,  0.06388889],
       [-0.06111111,  0.02222222,  0.10555556],
       [-0.01944444,  0.18888889, -0.10277778]])
```

In [86]:
```
v=[[2,3,4],
   [5,9,12],
   [4,6,8]]

np.linalg.pinv(v)
```

Out[86]:
```
array([[ 0.6 , -1.  ,  1.2 ],
       [-0.12,  0.24, -0.24],
       [-0.16,  0.32, -0.32]])
```

In [87]:
```
np.linalg.det(R)
```

Out[87]:
```
-359.9999999999997
```

In [88]:
```
a = np.array([[[1, 2], [3, 4]], [[1, 2], [2, 1]], [[1, 3], [3, 1]]])
a.shape
(3, 2, 2)
np.linalg.det(a)
```

Out[88]:
```
array([-2., -3., -8.])
```

In [89]:
```
R= np.array([8,1,6])

np.mean(R)
np.std(R)         #R.std()
np.var(R)
```

Out[89]:
```
8.666666666666666
```

In [90]:
```
#import libraries
import numpy as np
import matplotlib.pyplot as plt
```

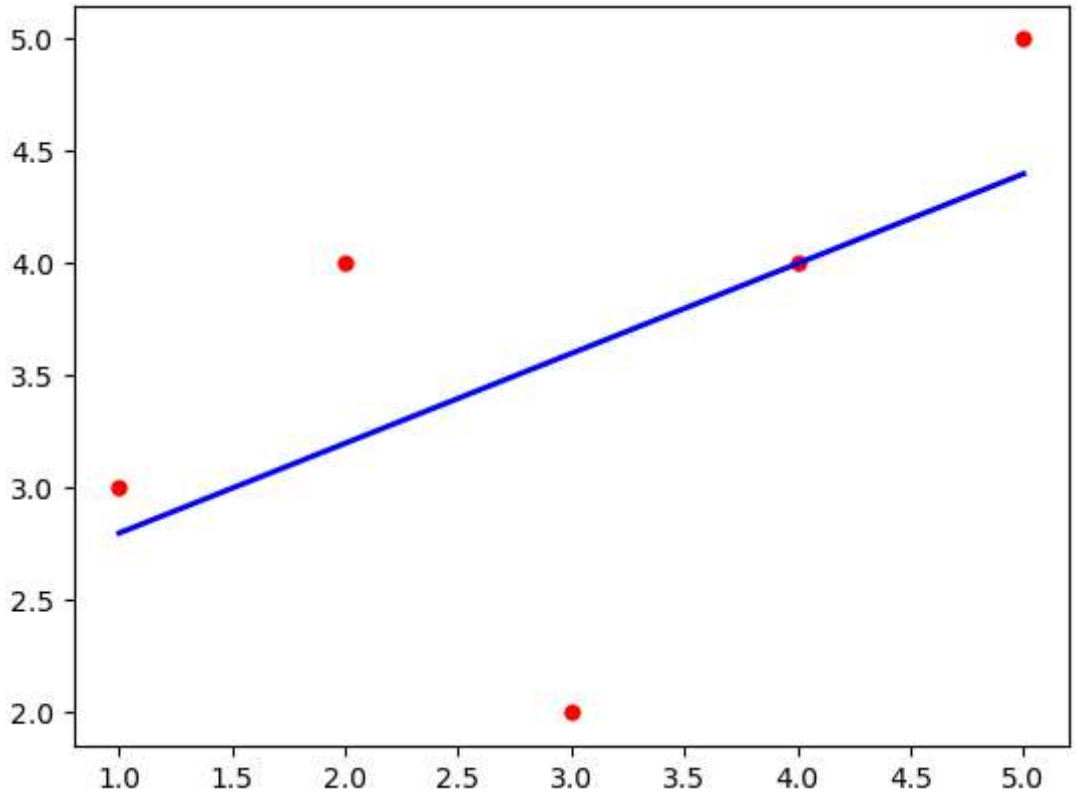
In [91]:
```
X= np.array([[1,1], [1,2], [1,3], [1,4], [1,5]])
y=np.array([[3],[4],[2],[4],[5]])
y.shape
```

Out[91]:
```
(5, 1)
```

In [92]:
```
# theta= inv(X) @y
theta= np.linalg.pinv(X)@y
print(theta.shape)
print(f" the slope is {theta[1]} and the intercept is {theta[0]}")
```

```
(2, 1)
 the slope is [0.4] and the intercept is [2.4]
```

In [93]:
```
def plot_hyp (X, y , theta):
    #plot data
    plt.scatter(X[: , 1], y , s=25 , c="r", marker="o")
    h=X@theta                                  #theta matrix multiplication
    plt.plot(X[ : ,1], h , c="b", lw=2)
    plt.show()                                  #no return?!

plot_hyp (X, y , theta)
```



In [94]:
```
#cost function

def mse(X, y , theta):
    """ a vectorized implementation of sum of squred error"""
    predisctions= X@theta
    erros= predisctions - y
    return 0.5 * np.sum(erros**2)
a=mse(X , y , theta)
print(f" the cost function is: {a:.2}")
```

```
 the cost function is: 1.8
```

In [95]:
```
#calculating y_hat ans y bar
y_bar= sum(y / len(y))
y_hat= []
for i in range(len(X)):
    y_hat.append(X[i][1]*theta[1]+ theta[0])
print(y_hat)
```

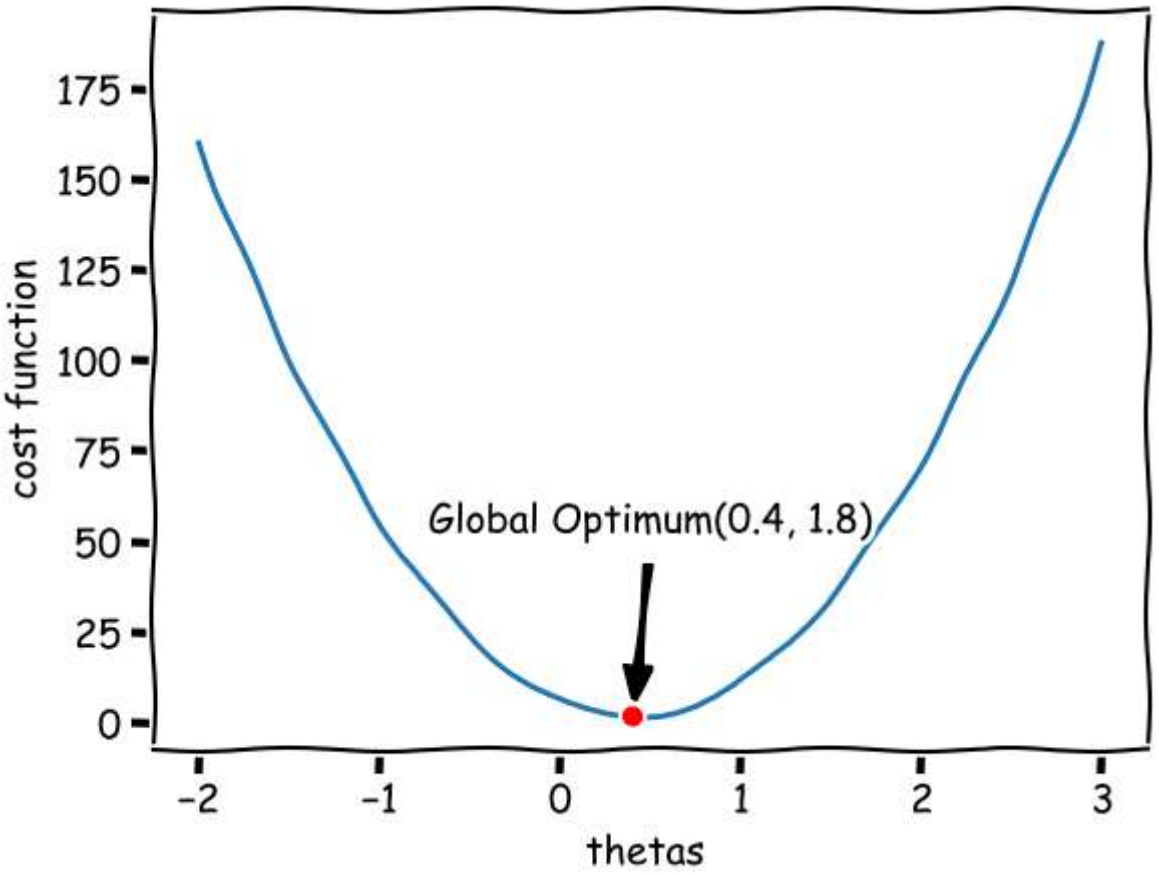
```
[array([2.8]), array([3.2]), array([3.6]), array([4.]), array([4.4])]
```

In [96]:
```
#rse
def rse(y , y_hat , y_bar):
    enumerator=0
    denomerator=0
    for i in range(len(y)):
        enumerator+= (y[i]-y_hat[i])**2
        denomerator+= (y[i]-y_bar)**2
    return enumerator/denomerator
a=rse(y , y_hat , y_bar)
a
```

Out[96]:
```
array([0.69230769])
```

In [97]:
```
#cost function diagram
thetas= np.linspace(-2 , 3 , num=50)
J=[]
for z in thetas:
    cost=mse(X , y , np.array(([2.4],[z])))      #intercept is  constant to dipict 2d
    J+=[cost]
```

In [98]:
```
with plt.xkcd():
    plt.plot(thetas , J)
    plt.plot(0.4,1.8 , "ro")
    plt.xlabel(r"thetas ")
    plt.ylabel(r"cost function")
    plt.annotate("Global Optimum(0.4, 1.8)" , xy=(0.4, 1.8) , xytext=(0.5,50) ,
                 arrowprops= dict(width =3, headwidth=10,facecolor="black", shrink=0.1
                                  horizontalalignment="center",verticalalignment="bott
```



In [ ]:

Example

Data Set Information: We perform energy analysis using 12 different building shapes simulated in Ecotect. The buildings differ with respect to the glazing area, the glazing area distribution, and the orientation, amongst other parameters. We simulate various settings as functions of the afore-mentioned characteristics to obtain 768 building shapes. The dataset comprises 768 samples and 8 features, aiming to predict two real valued responses. It can also be used as a multi-class classification problem if the response is rounded to the nearest integer. Specifically: X1 Relative Compactness # - تراکم نسبی X2 Surface Area # مساحت سطح X3 Wall Area # مساحت دیوار

X4 Roof Area # مساحت سقف X5 Overall Height # ارتفاع کلی X6 Orientation # جهت گیری ساختمان X7 Glazing Area #
X8 Glazing Area Distribution # توزیع منطقه شفاف y1 Heating Load # بار گرمایی ناحیه شفاف

x1 = مساحت سقف x2= مساحت دیوار

x3 = ارتفاع کلی x4 = ناحیه شفاف x5 = کمینه دمای متوسط در 10 سال X6= وجود عایق

In [99]:
```
import numpy as np
import pandas as pd
data=pd.read_csv(r"F:\data\linearworkshop\energy.csv")
data.tail(10)
```

Out[99]:

	X0	X1	X2	X3	X4	X5	X6	X7	X8	Y
758	1	0.66	759.5	318.5	220.5	3.5	4	0.4	5	14.92
759	1	0.66	759.5	318.5	220.5	3.5	5	0.4	5	15.16
760	1	0.64	784.0	343.0	220.5	3.5	2	0.4	5	17.69
761	1	0.64	784.0	343.0	220.5	3.5	3	0.4	5	18.19
762	1	0.64	784.0	343.0	220.5	3.5	4	0.4	5	18.16
763	1	0.64	784.0	343.0	220.5	3.5	5	0.4	5	17.88
764	1	0.62	808.5	367.5	220.5	3.5	2	0.4	5	16.54
765	1	0.62	808.5	367.5	220.5	3.5	3	0.4	5	16.44
766	1	0.62	808.5	367.5	220.5	3.5	4	0.4	5	16.48
767	1	0.62	808.5	367.5	220.5	3.5	5	0.4	5	16.64

In [100…
```
data.describe()
```

Out[100]:

	X0	X1	X2	X3	X4	X5	X6	X7	
count	768.0	768.000000	768.000000	768.000000	768.000000	768.00000	768.000000	768.000000	768.
mean	1.0	0.764167	671.708333	318.500000	176.604167	5.25000	3.500000	0.234375	2.
std	0.0	0.105777	88.086116	43.626481	45.165950	1.75114	1.118763	0.133221	1.
min	1.0	0.620000	514.500000	245.000000	110.250000	3.50000	2.000000	0.000000	0.
25%	1.0	0.682500	606.375000	294.000000	140.875000	3.50000	2.750000	0.100000	1.
50%	1.0	0.750000	673.750000	318.500000	183.750000	5.25000	3.500000	0.250000	3.
75%	1.0	0.830000	741.125000	343.000000	220.500000	7.00000	4.250000	0.400000	4.
max	1.0	0.980000	808.500000	416.500000	220.500000	7.00000	5.000000	0.400000	5.

In [101…
```
X= np.array(data.drop(["Y"] , axis= 1))
y=np.array(data["Y"])

y=y.reshape((768,1))
```

In [102…
```
print(X.shape)
print(y.shape)
```

```
(768, 9)
(768, 1)
```

In [103…
```
theta= np.linalg.pinv(X)@y

theta
```

Out[103]:
```
array([[ 8.40145212e+01],
       [-6.47739915e+01],
       [-6.26063386e-02],
       [ 3.61294044e-02],
       [-4.93678715e-02],
       [ 4.16993881e+00],
       [-2.33281250e-02],
       [ 1.99326802e+01],
       [ 2.03771772e-01]])
```

In [105…
```
y_hat= X@theta


#y_hat
```

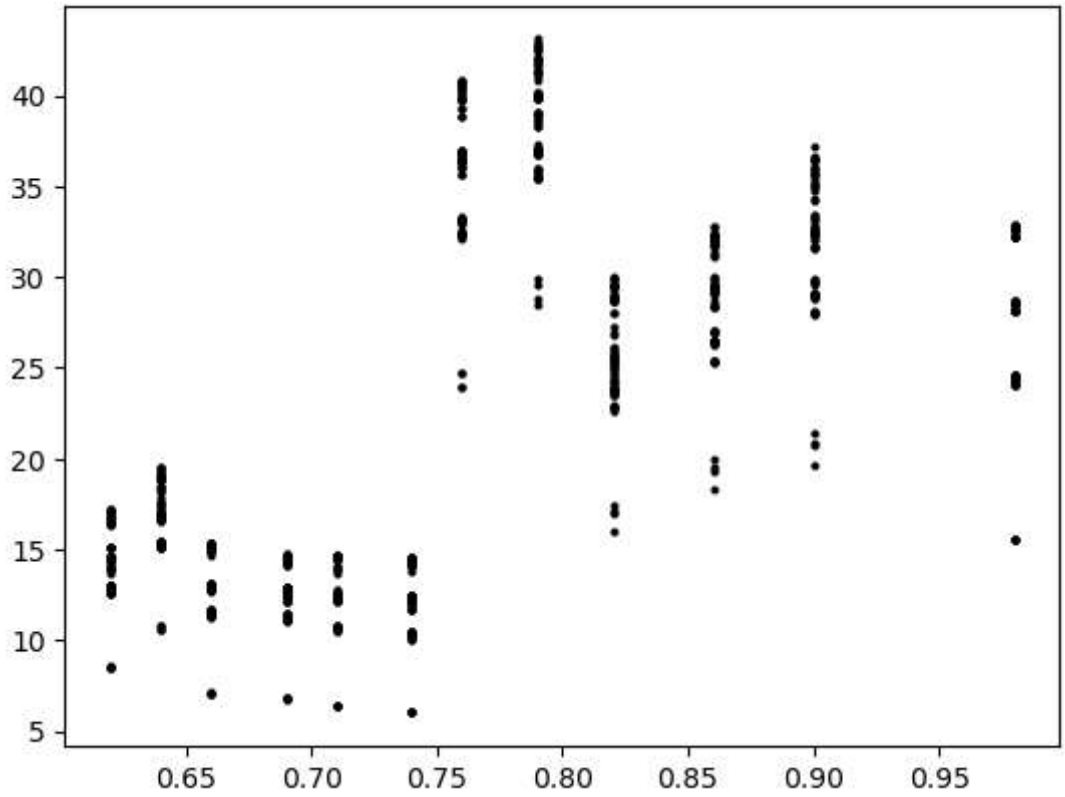
In [106…
```
x_predict=np.array( [[1,0.98,72,90,72,3,2,0.16,1]])
x_predict@theta
```

Out[106]:
```
array([[31.58167359]])
```

In [107…
```
numerator=sum ((y - y_hat)**2)
Denominator=sum ((y - y.mean())**2)

rse= numerator / Denominator


R_squared= 1 - rse


print(f" the accuracy of the algorhim is: {round(float(R_squared*100), 2)} ")
```

```
 the accuracy of the algorhim is: 91.62
```

In [108…
```
import matplotlib.pyplot as plt
plt.scatter(X[ : , 1] , y , c="black" , s=4)
plt.show()
```



Concrete compressive strength

In [109…

```
import pandas as pd
concrete_data=pd.read_csv("Concrete_Data .csv")


display(concrete_data.head())
```

	X0	X1	X2	X3	X4	X5	X6	X7	X8	y
0	1	540.0	0.0	0.0	162.0	2.5	1040.0	676.0	28	79.99
1	1	540.0	0.0	0.0	162.0	2.5	1055.0	676.0	28	61.89
2	1	332.5	142.5	0.0	228.0	0.0	932.0	594.0	270	40.27
3	1	332.5	142.5	0.0	228.0	0.0	932.0	594.0	365	41.05
4	1	198.6	132.4	0.0	192.0	0.0	978.4	825.5	360	44.30

Number of instances (observations): 1030 Number of Attributes: 9 Attribute breakdown: 8 quantitative input variables, and 1 quantitative output variable Missing Attribute Values: None x1= Cement (component 1) -- quantitative -- kg in a m3 mixture -- Input Variable # سیمان x2=Blast Furnace Slag (component 2) -- quantitative -- kg in a m3 mixture -- Input Variable # سرباره کوره بلند x3=Fly Ash (component 3) -- quantitative -- kg in a m3 mixture -- Input Variable # خاکستر بادی x4=Water (component 4) -- quantitative -- kg in a m3 mixture -- Input Variable # آب x5=Superplasticizer (component 5) -- quantitative -- kg in a m3 mixture -- Input Variable # فوق روان کننده x6=Coarse Aggregate (component 6) -- quantitative -- kg in a m3 mixture -- Input Variable # سنگدانه تیز گوشه x7=Fine Aggregate (component 7) -- quantitative -- kg in a m3 mixture -- Input Variable # سنگدانه گرد گوشه x8=Age -- quantitative -- Day (1~365) -- Input Variable # سن y=Concrete compressive strength -- quantitative -- MPa -- Output Variable #مقاومت فشاری بتن

In [110…
```
X= np.array(concrete_data.drop(["y"] , axis= 1))
y=np.array(concrete_data["y"])

print(X.shape)
print(y.shape)
```

```
(1030, 9)
(1030,)
```

In [111…
```
y=y.reshape(1030,1)

print(y.shape)
```

```
(1030, 1)
```

In [112…
```
theta= np.linalg.pinv(X)@y

theta
```

Out[112]:
```
array([[-2.33312136e+01],
       [ 1.19804334e-01],
       [ 1.03865809e-01],
       [ 8.79343215e-02],
       [-1.49918419e-01],
       [ 2.92224595e-01],
       [ 1.80862148e-02],
       [ 2.01903511e-02],
       [ 1.14222068e-01]])
```

In [113…
```
y_hat= X@theta
y_hat
```

Out[113]:
```
array([[53.46346329],
       [53.73475651],
       [56.81258504],
       ...,
       [26.46841169],
       [29.12237014],
       [31.89770807]])
```

In [114…
```
x_predict=np.array( [[1,470,0,118.4,153,1.8,700,500,28]])
x_predict@theta
```

Out[114]:
```
array([[46.93047727]])
```

In [115…
```
numerator=sum ((y - y_hat)**2)
Denominator=sum ((y - y.mean())**2)

rse= numerator / Denominator


R_squared= 1 - rse


print(f" the accuracy of the algorhim is: {round(float(R_squared*100), 2)} ")
```

```
 the accuracy of the algorhim is: 61.55
```

In [ ]:
```
import matplotlib.pyplot as plt
plt.scatter(X[ : , 4] , y , c="black" , s=4)
```

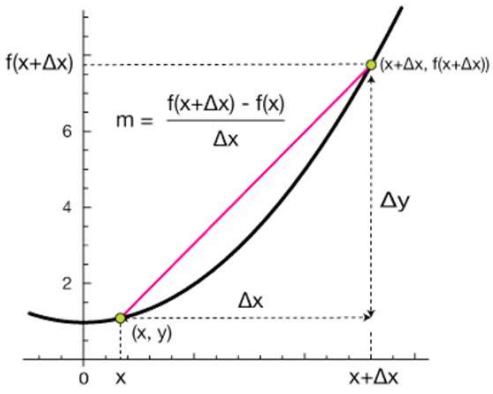
```
plt.show()
```

In [ ]:
```
plt.scatter(X[ : , 1] , X[ : , 4] , c="red" , s=4)
plt.show()
```


- **Derivative** (differential algebra)

Y= f(x)

$$y' = f'(x) = \frac{dy}{dx}$$



Derivative

f(x)	$\frac{dy}{dx}$
a 5 $2\sqrt{3}$	**0**
$\boldsymbol{x^m}$ x^3 θ^2	$\mathbf{m}\boldsymbol{x^{m-1}}$

f(x)	$\frac{dy}{dx}$
$\mathbf{a}\boldsymbol{x^m}$ $2x^3$ $2x^3+4x^2+3x+7$	$\mathbf{am}\boldsymbol{x^{m-1}}$
$\boldsymbol{u} + \boldsymbol{g}$ $(x^2+5) + (3x^3+3x)$	$\boldsymbol{u'} + \boldsymbol{g'}$

Derivative

f(x)	$\frac{dy}{dx}$
$\boldsymbol{u} \times \boldsymbol{g}$ $(x^2-1)\,(x^5+3x)$ $(3x^2+2)\,(x^3-3x)$	$\boldsymbol{u'} \times \mathbf{g} + \boldsymbol{g'} \times \mathbf{u}$
$\boldsymbol{u} - \boldsymbol{g}$ $(x^2+5) - (3x^3+3x)$	$\boldsymbol{u'} - \boldsymbol{g'}$

f(x)	$\frac{dy}{dx}$
$\frac{\boldsymbol{u}}{\boldsymbol{g}}$ $\frac{(x^2-1)}{(x^5+3x)}$ $\frac{(3x^2+2)}{(x^3-3x)}$	$\frac{\boldsymbol{u'} \times \mathbf{g} - \boldsymbol{g'} \times \mathbf{u}}{\boldsymbol{g^2}}$
$(\boldsymbol{u})^{\boldsymbol{m}}$ $(2x^2-x)^2$ $(3x^3-2x)^6$	$\mathbf{m}(\boldsymbol{u^{m-1}})(\boldsymbol{u'})$

Partial Derivative

z= f(x,y)

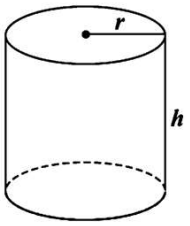
$$z'_x = \frac{\partial z}{\partial x}$$
$$z'_y = \frac{\partial z}{\partial y}$$

Z(x, y)= $x^2+5xy^2+4y^3$

U(x, y) = $\frac{(x+y)}{(x-y)}$

F(X, Y) = $(2x^2-y)^2$

Consider a cylinder of height h and radius r. Calculate the volume changes of the cylinder in two cases. In the first case, only the radius of the cylinder is allowed to change, and in the second case, only the height of the cylinder is changed



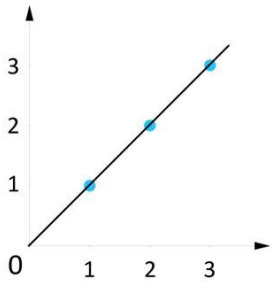
F(r , h) = $\pi r^2 h$

COST FUNCTION

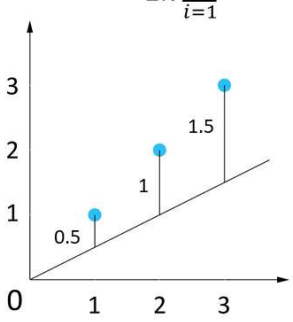
$$h_\theta(x) = \theta_0 + \theta_1 x$$

MSE $J(\theta_0, \theta_1) = \frac{1}{2n}\sum_{i=1}^{n}(\hat{y}_i - y_i)^2 = \frac{1}{2n}\sum_{i=1}^{n}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

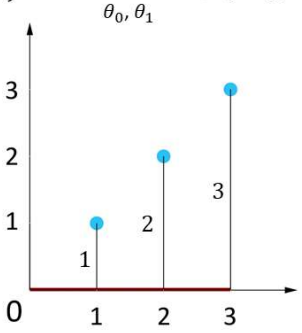
Minimize $J(\theta_0, \theta_1)$ $_{\theta_0, \theta_1}$



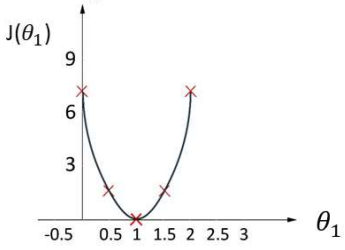
$J(0, \theta_1) = \frac{1}{2}(0^2 + 0^2 + 0^2) = 0$



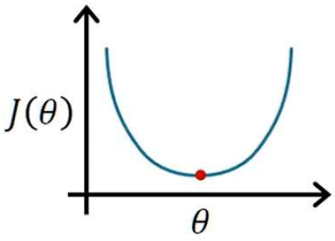
$J(0, \theta_1) = \frac{1}{2}(0.5^2 + 1.0^2 + 1.5^2) = 1.75$



$J(0, \theta_1) = \frac{1}{2}(1.0^2 + 2.0^2 + 3.0^2) = 7.0$



- **Cost function derivative**



$\theta \in R$: $J(\theta) = a\theta^2 + b\theta + c$

$\frac{d}{d\theta} J(\theta) \stackrel{\text{def}}{=} 0$ J= 0,1,2,…,n

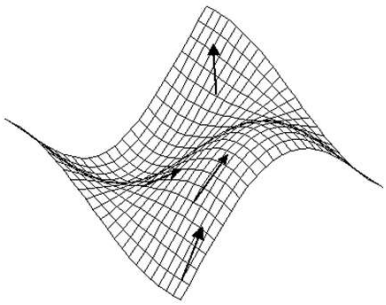
$\theta \in R^{n+1}$: $J(\theta_1, \theta_2, \theta_3, \ldots, \theta_n) = \frac{1}{2}\sum_{i=1}^{n}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

https://www.geeksforgeeks.org/ml-normal-equation-in-linear-regression/

- **Gradient Descent**

gradient vector

The gradient vector can be interpreted as the "direction and rate of fastest increase". If the gradient of a function is non-zero at a point p, the direction of the gradient is the direction in which the function increases most quickly from p, and the magnitude of the gradient is the rate of increase in that direction, the greatest absolute directional derivative. Further, a point where the gradient is the zero vector is known as a stationary point. The gradient thus plays a fundamental role in optimization theory, where it is used to maximize a function by gradient ascent. (Wikipedia)

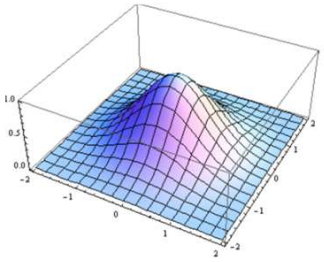


Gradient Vectors Shown at Several Points on the Surface of cos(x) sin(y)

Gradient Descent

gradient vector: a vector whose value at a point p is the vector whose components are the partial derivatives of function f at p. shows with ∇f.

∇f=grad(f)



U = f(x_1+ x_2+x_3+... x_n)

∇ U =(u'_{x_1}, u'_{x_2}, u'_{x_3}, ..., u'_{x_n})

? Z= x^2+5xy^2 + $4y^3$

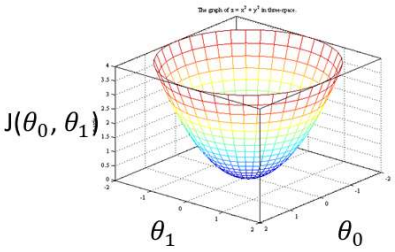
Specify gradient vector at point(1,1)

? J(θ_0, θ_1)= x^2y

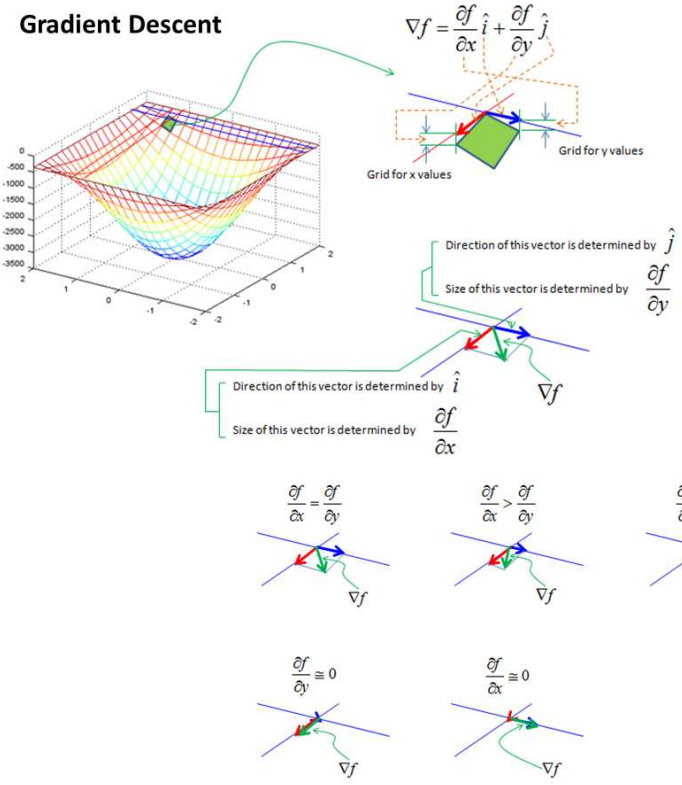
Specify gradient vector at point(2,3)

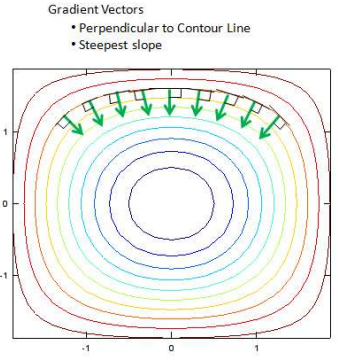
? J = ${\theta_0}^2$+ ${\theta_1}^2$

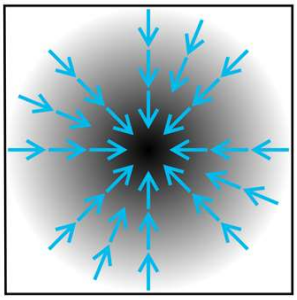
Specify the gradient vector at point (0,0)



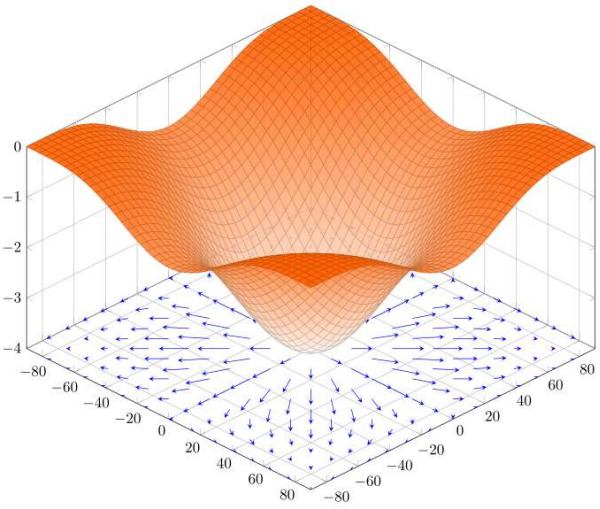
Gradient Descent





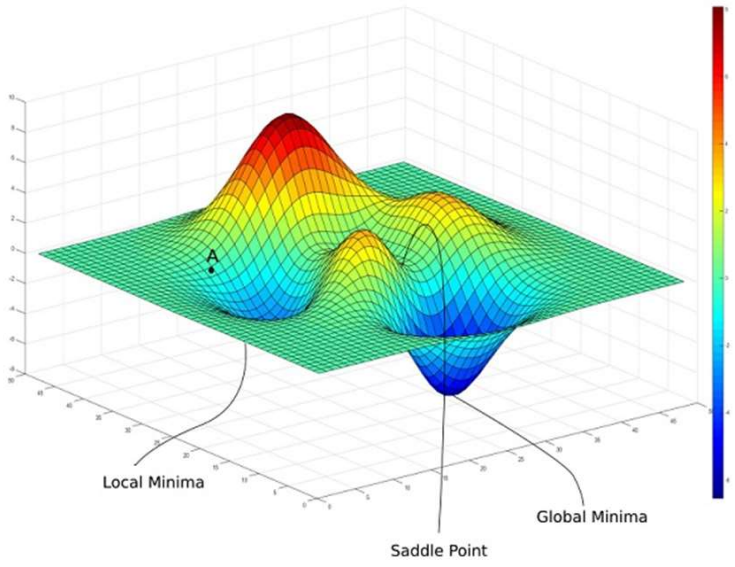


Gradient Descent



The gradient of the function $f(x,y) = -(\cos^2 x + \cos^2 y)^2$ is depicted as a projected vector field on the bottom plane.

Gradient Descent



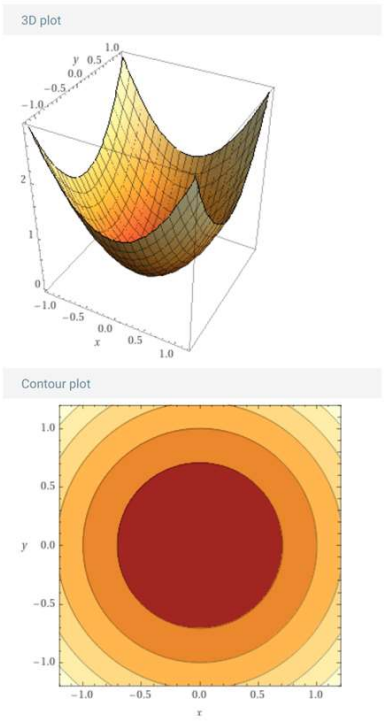
Gradient Descent

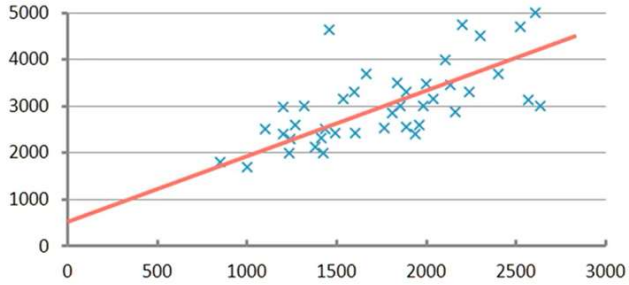
Hypothesis $h_\theta(x) = \theta_0 + \theta_1 x$

Parameter θ_0, θ_1

Cost function $J(\theta_0, \theta_1) = \frac{1}{2n}\sum_{i=1}^{n}\left(h_\theta\left(x^{(i)}\right) - y^{(i)}\right)^2$

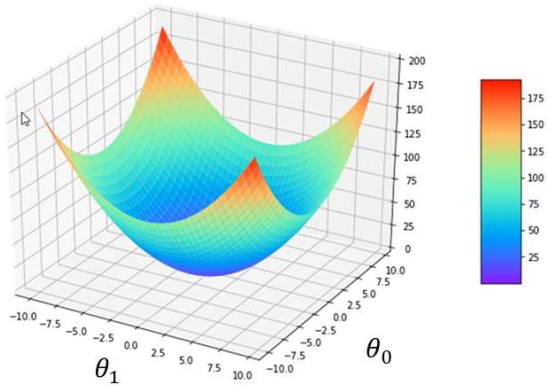
Goal Minimize $J(\theta_0, \theta_1)$



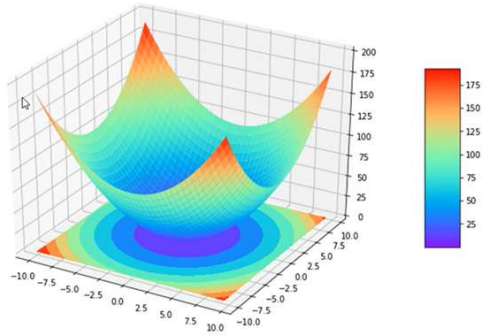


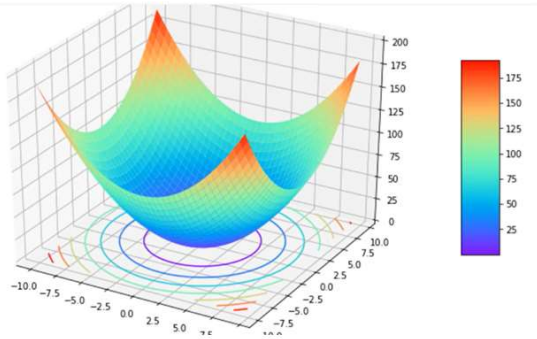
$h_\theta(x) = \theta_0 + \theta_1 x$

$J(\theta_0, \theta_1)$

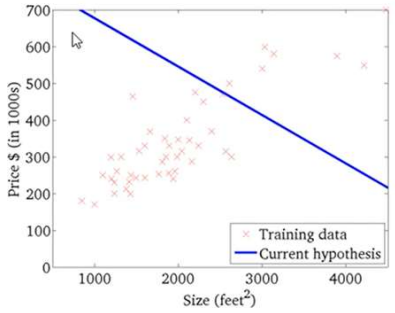


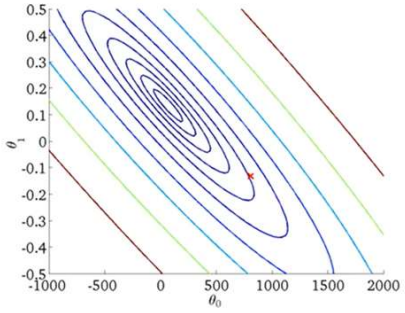
$J = \theta_0^2 + \theta_1^2$

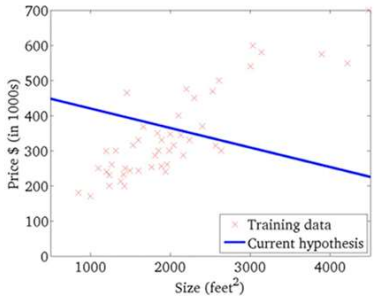


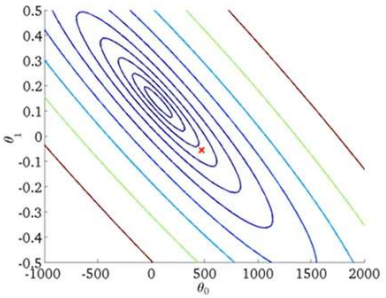


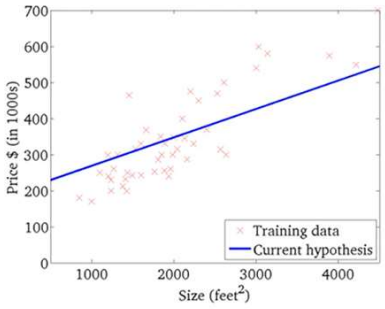
```
fig = plt.figure(figsize=(12, 8))
ax = fig.gca(projection='3d')
surf = ax.plot_surface(X, Y, Z, cmap=plt.cm.rainbow)
cset = ax.contourf(X, Y, Z, zdir='z', offset=0, cmap=plt.cm.rainbow)
fig.colorbar(surf, shrink=0.5, aspect=5)
plt.show()
```

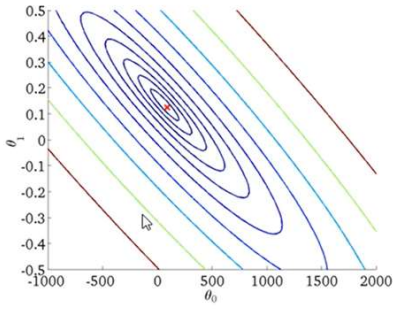








$h_\theta(x) = \theta_0 + \theta_1 x$



$J(\theta_0, \theta_1)$

- Method

start with a random initial value for the parameters θ_0, θ_1

change the parameters in such a way that the value of the cost function decreases

We repeat the steps until we reach a minimum value for the cost function (convergence).

- **Mathematic**

Repeat until convergence

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ for j =0 and i = 0

α , η : Learning rate

$J(\theta_0, \theta_1)$

$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix}$

$\theta^T = [\theta_0 \quad \theta_1]$

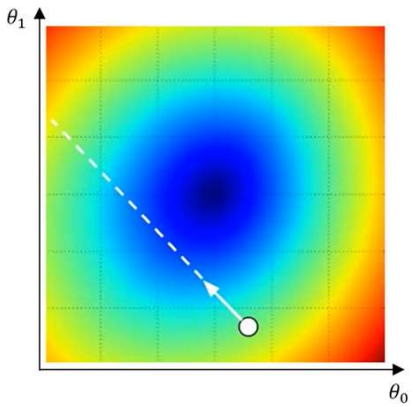
$\nabla J = \begin{bmatrix} \frac{\partial J}{\partial \theta_0} \\ \frac{\partial J}{\partial \theta_1} \end{bmatrix}$

$-\nabla J = \begin{bmatrix} -\frac{\partial J}{\partial \theta_0} \\ -\frac{\partial J}{\partial \theta_1} \end{bmatrix}$

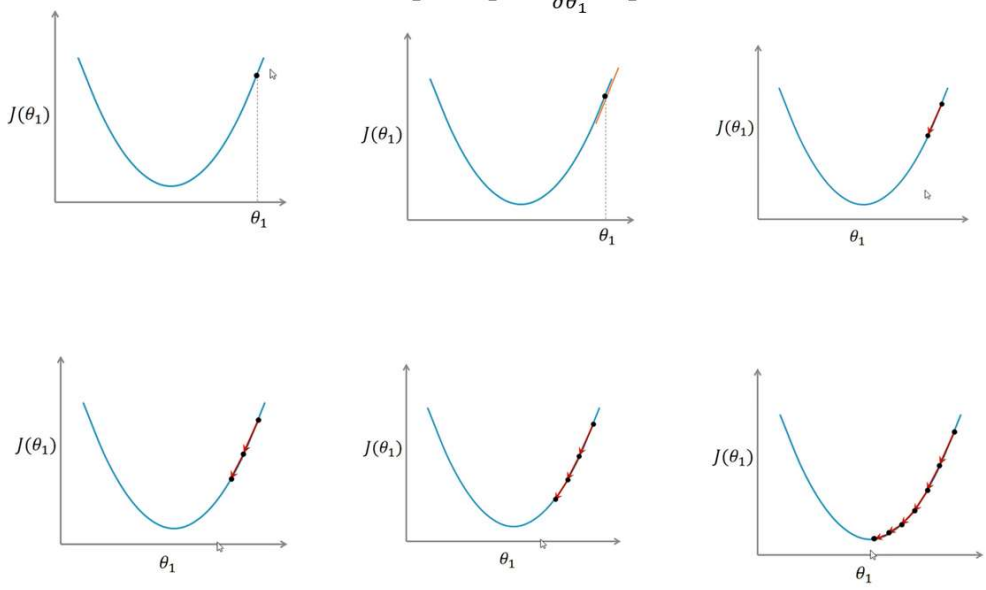
$\theta^{(new)} := \theta + \alpha(-\nabla J)$

$\begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} = \begin{bmatrix} \theta_0 \\ \theta_1 \end{bmatrix} + \alpha \begin{bmatrix} -\frac{\partial J}{\partial \theta_0} \\ -\frac{\partial J}{\partial \theta_1} \end{bmatrix}$

$\theta_j := \theta_j - \alpha\left(\frac{\partial J}{\partial \theta_j}\right)$

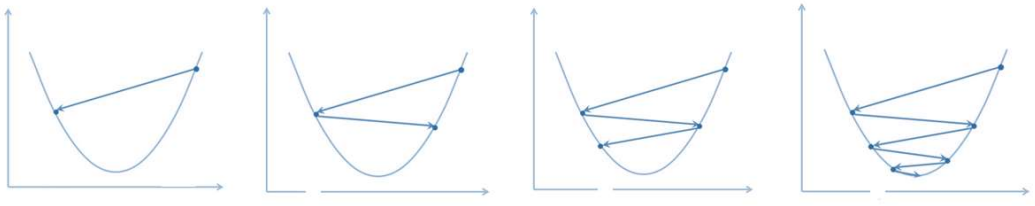


$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$



If the learning rate is too small, the gradient descent converges slowly.

If the learning rate is too large, the gradient descent may converge slowly or even diverge.



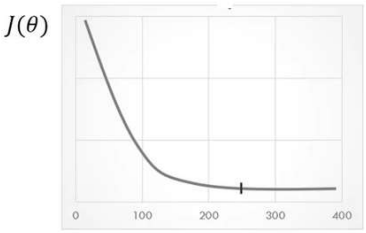
slow convergence

divergence

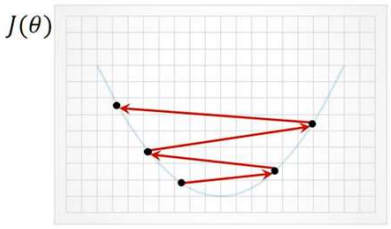
Determine the value of α

Convergence test

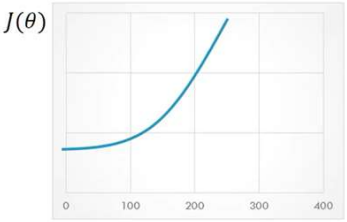
Convergence has occurred if the value of j(θ) changes less than 10^{-3} in one iteration.



Repeated times convergence



divergence



Repeated times divergence

Try a number like 1/1000 and multiply by one number each time to reach the appropriate rate. It changes logarithmically.

Mathematic

$$\theta_1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_1)$$

$$J(\theta_0, \theta_1) = \frac{\partial}{\partial \theta_j} \frac{1}{2} \sum_{i=1}^{n} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)^2 = \frac{\partial}{\partial \theta_j} \frac{1}{2} \sum_{i=1}^{n} \left(\theta_0 + \theta_1 x^{(i)} - y^{(i)} \right)^2$$

J=0

$$\frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1) = \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right)$$

J=1

$$\frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1) = \sum_{i=1}^{m} \left(h_\theta \left(x^{(i)} \right) - y^{(i)} \right) . x^{(i)}$$

Gradient descent	Normal equation
Need to choose α	Not to need choosing α
Need to multiple repetition	Not to need repetition
Works good even for large amount of n	Because of calculating x transpose is slow for large n
N>=100	N<10000

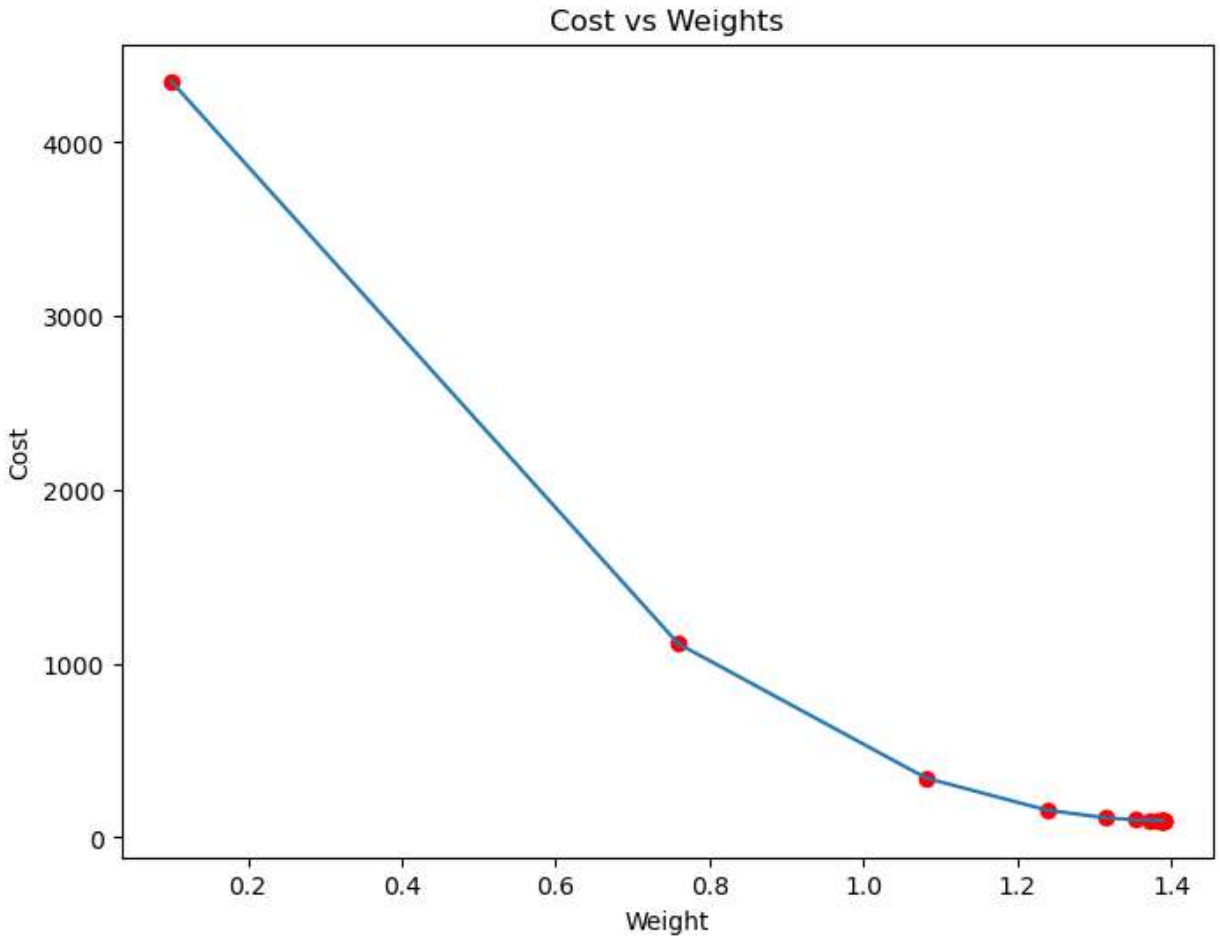
In [116…

```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt

def mean_squared_error(y_true, y_predicted):

    # Calculating the loss or cost
    cost = np.sum((y_true-y_predicted)**2) / len(y_true)
    return cost

# Gradient Descent Function
# Here iterations, learning_rate, stopping_threshold
# are hyperparameters that can be tuned
def gradient_descent(x, y, iterations = 1000, learning_rate = 0.0001,
                     stopping_threshold = 1e-6):

    # Initializing weight, bias, learning rate and iterations
    current_weight = 0.1
    current_bias = 0.01
    iterations = iterations
    learning_rate = learning_rate
    n = float(len(x))

    costs = []
    weights = []
    previous_cost = None

    # Estimation of optimal parameters
    for i in range(iterations):
```

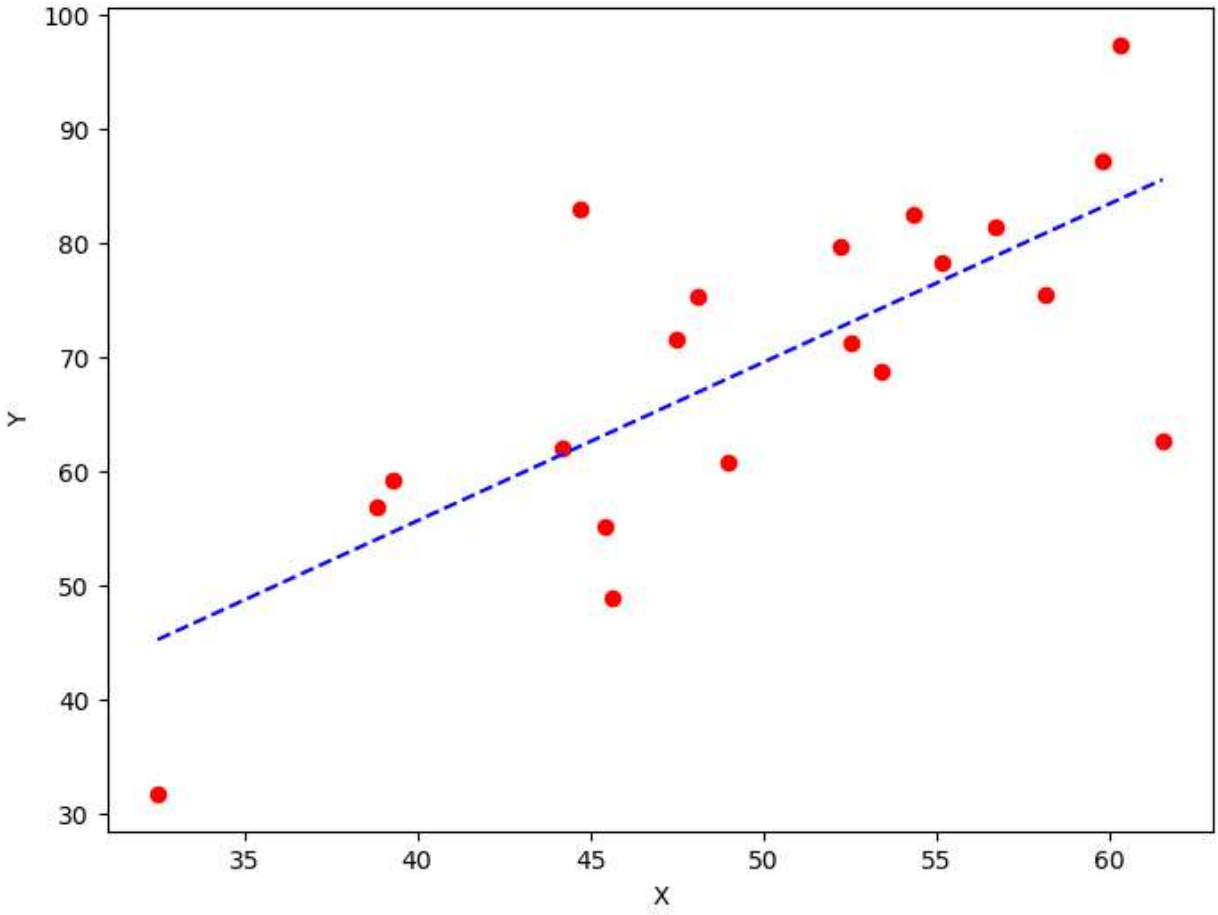
```
        # Making predictions
        y_predicted = (current_weight * x) + current_bias

        # Calculationg the current cost
        current_cost = mean_squared_error(y, y_predicted)

        # If the change in cost is less than or equal to
        # stopping_threshold we stop the gradient descent
        if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)

        # Calculating the gradients
        weight_derivative = -(2/n) * sum(x * (y-y_predicted))
        bias_derivative = -(2/n) * sum(y-y_predicted)

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)

        # Printing the parameters for each 1000th iteration
        print(f"Iteration {i+1}: Cost {current_cost}, Weight \
        {current_weight}, Bias {current_bias}")


    # Visualizing the weights and cost at for all iterations
    plt.figure(figsize = (8,6))
    plt.plot(weights, costs)
    plt.scatter(weights, costs, marker='o', color='red')
    plt.title("Cost vs Weights")
    plt.ylabel("Cost")
    plt.xlabel("Weight")
    plt.show()

    return current_weight, current_bias



def main():

    # Data
    X = np.array([32.50234527, 53.42680403, 61.53035803, 47.47563963, 59.81320787,
           55.14218841, 52.21179669, 39.29956669, 48.10504169, 52.55001444,
           45.41973014, 54.35163488, 44.1640495 , 58.16847072, 56.72720806,
           48.95588857, 44.68719623, 60.29732685, 45.61864377, 38.81681754])
    Y = np.array([31.70700585, 68.77759598, 62.5623823 , 71.54663223, 87.23092513,
           78.21151827, 79.64197305, 59.17148932, 75.3312423 , 71.30087989,
           55.16567715, 82.47884676, 62.00892325, 75.39287043, 81.43619216,
           60.72360244, 82.89250373, 97.37989686, 48.84715332, 56.87721319])

    # Estimating weight and bias using gradient descent
    estimated_weight, eatimated_bias = gradient_descent(X, Y, iterations=2000)
    print(f"Estimated Weight: {estimated_weight}\nEstimated Bias: {eatimated_bias}")

    # Making predictions using estimated parameters
```

```
    Y_pred = estimated_weight*X + eatimated_bias

    # Plotting the regression line
    plt.figure(figsize = (8,6))
    plt.scatter(X, Y, marker='o', color='red')
    plt.plot([min(X), max(X)], [min(Y_pred), max(Y_pred)], color='blue',markerfacecolo
             markersize=10,linestyle='dashed')
    plt.xlabel("X")
    plt.ylabel("Y")
    plt.show()


if __name__=="__main__":
    main()
```

```
Iteration 1: Cost 4352.088931274409, Weight         0.7593291142562117, Bias 0.022885
58130709
Iteration 2: Cost 1114.8561474350017, Weight         1.081602958862324, Bias 0.029180
14748569513
Iteration 3: Cost 341.42912086804455, Weight         1.2391274084945083, Bias 0.03225
308846928192
Iteration 4: Cost 156.64495290904443, Weight         1.3161239281746984, Bias 0.03375
132986012604
Iteration 5: Cost 112.49704004742098, Weight         1.3537591652024805, Bias 0.03447
9873154934775
Iteration 6: Cost 101.9493925395456, Weight         1.3721549833978113, Bias 0.034832
195392868505
Iteration 7: Cost 99.4293893333546, Weight         1.3811467575154601, Bias 0.0350006
2439068245
Iteration 8: Cost 98.82731958262897, Weight         1.3855419247507244, Bias 0.035079
16814736111
Iteration 9: Cost 98.68347500997261, Weight         1.3876903144657764, Bias 0.035113
776874486774
Iteration 10: Cost 98.64910780902792, Weight         1.3887405007983562, Bias 0.03512
6910596389935
Iteration 11: Cost 98.64089651459352, Weight         1.389253895811451, Bias 0.035129
54755833985
Iteration 12: Cost 98.63893428729509, Weight         1.38950491235671, Bias 0.0351270
53821718185
Iteration 13: Cost 98.63846506273883, Weight         1.3896276808137857, Bias 0.03512
2052266051224
Iteration 14: Cost 98.63835254057648, Weight         1.38968776283053, Bias 0.0351158
2492978764
Iteration 15: Cost 98.63832524036214, Weight         1.3897172043139192, Bias 0.03510
899846107016
Iteration 16: Cost 98.63831830104695, Weight         1.389731668997059, Bias 0.035101
879159522745
Iteration 17: Cost 98.63831622628217, Weight         1.389738813163012, Bias 0.035094
61674147458
```




```
Estimated Weight: 1.389738813163012
Estimated Bias: 0.03509461674147458
```



In [117…

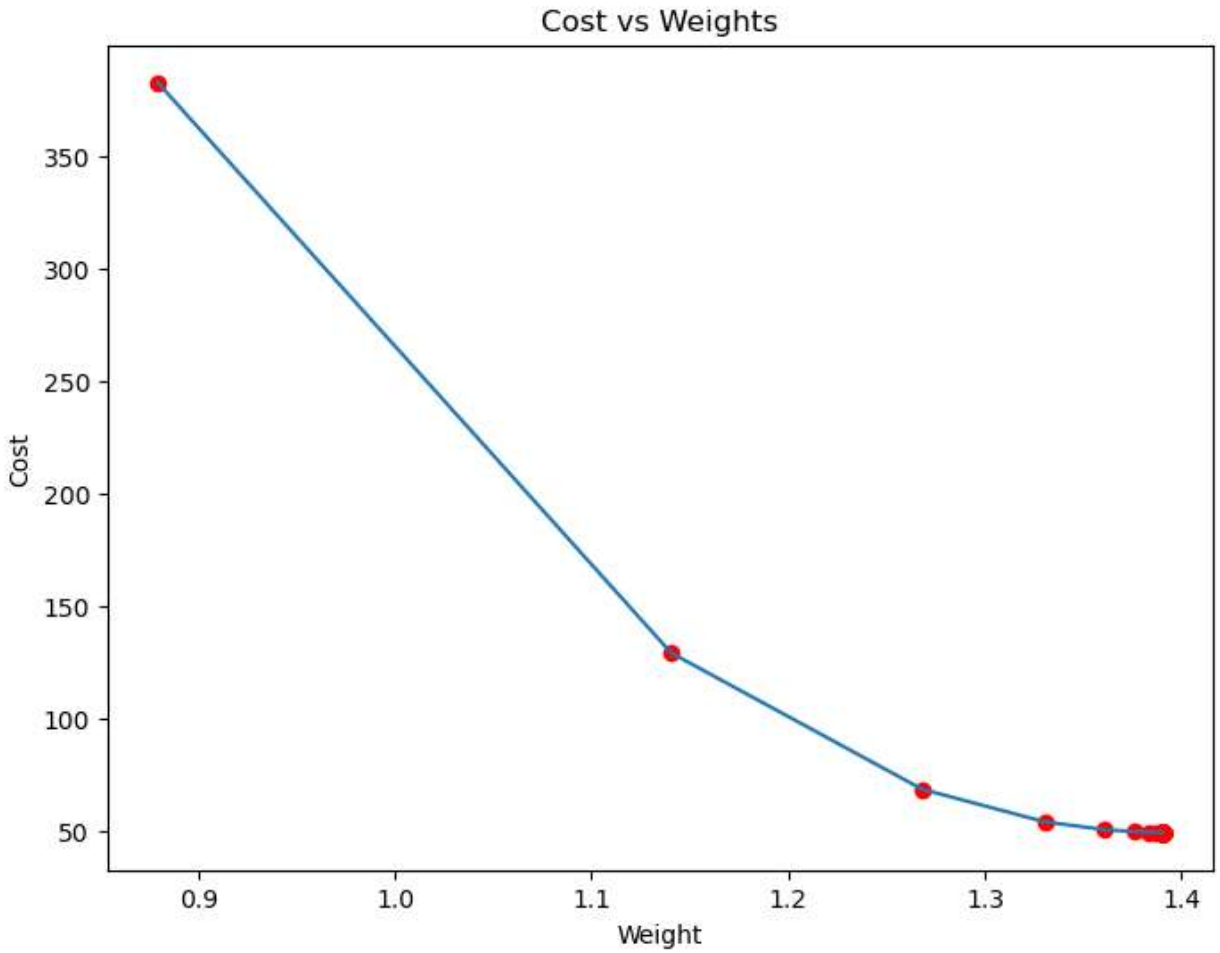
```
# Importing Libraries
import numpy as np
import matplotlib.pyplot as plt



def train (X, y, iterations = 1000, learning_rate = 0.0001,
                      stopping_threshold = 1e-6):

    m, n = X.shape

    # Initializing weight, bias, learning rate and iterations
    current_weight = np.random.rand(n ,1)
    current_bias = np.zeros([1,1])

    iterations = iterations
    learning_rate = learning_rate


    costs = []
    weights = []
    biass=[]
    previous_cost = None

    # Estimation of optimal parameters
    for i in range(iterations):

        # Making predictions
        prediction = (X@current_weight) + current_bias
```

```
        # Calculationg the current cost
        current_cost =  np.sum((1/(m*2)) *  (prediction - y)**2)



        # If the change in cost is less than or equal to
        # stopping_threshold we stop the gradient descent
        if previous_cost and abs(previous_cost-current_cost)<=stopping_threshold:
            break

        previous_cost = current_cost

        costs.append(current_cost)
        weights.append(current_weight)
        biass.append(current_bias)

        # Calculating the gradients
        weight_derivative =  X.T@(prediction-y)
        bias_derivative = ( prediction-y )

        # Updating weights and bias
        current_weight = current_weight - (learning_rate * weight_derivative)
        current_bias = current_bias - (learning_rate * bias_derivative)

    return costs , weights
```

In [6]:

```
X = np.array([[32.50234527],
              [53.42680403],
              [61.53035803],
              [47.47563963],
              [59.81320787],
              [55.14218841],
              [52.21179669],
              [39.29956669],
              [48.10504169],
              [52.55001444],
              [45.41973014],
              [54.35163488],
              [44.1640495 ],
              [58.16847072],
              [56.72720806],
              [48.95588857],
              [44.68719623],
              [60.29732685],
              [45.61864377],
              [38.81681754]])

y = np.array([[31.70700585],
              [68.77759598],
              [62.5623823],
              [71.54663223],
              [87.23092513],
              [78.21151827],
```

```
              [79.64197305],
              [59.17148932],
              [75.3312423 ],
              [71.30087989],
              [55.16567715],
              [82.47884676],
              [62.00892325],
              [75.39287043],
              [81.43619216],
              [60.72360244],
              [82.89250373],
              [97.37989686],
              [48.84715332],
              [56.87721319]])



a=train (X, y, iterations = 10, learning_rate = 0.00001, stopping_threshold = 1e-6)

weights= a[1]
costs=a[0]

"""
# Visualizing the weights and cost at for all iterations
plt.figure(figsize = (8,6))
plt.plot(weights, costs)
plt.scatter(weights, costs, marker='o', color='red')
plt.title("Cost vs Weights")
plt.ylabel("Cost")
plt.xlabel("Weight")
plt.show()
"""
#print(np.round(costs , 2).shape)
#print(np.round(weights , 2).shape)

#print(weights)

list_w=[]
for i in range(len(weights)):
    list_w.append(weights[i][0][0])


# Visualizing the weights and cost at for all iterations
plt.figure(figsize = (8,6))
plt.plot(list_w, costs)
plt.scatter(list_w, costs, marker='o', color='red')
plt.title("Cost vs Weights")
plt.ylabel("Cost")
plt.xlabel("Weight")
plt.show()
```



Sklearn

In [118…

```
import sklearn

from sklearn.linear_model import LinearRegression



from sklearn import metrics
```

In [122…

```
import numpy as np
import pandas as pd
data=pd.read_csv(r"F:\data\linearworkshop\energy.csv")
data.tail(10)
```

Out[122]:

	X0	X1	X2	X3	X4	X5	X6	X7	X8	Y
758	1	0.66	759.5	318.5	220.5	3.5	4	0.4	5	14.92
759	1	0.66	759.5	318.5	220.5	3.5	5	0.4	5	15.16
760	1	0.64	784.0	343.0	220.5	3.5	2	0.4	5	17.69
761	1	0.64	784.0	343.0	220.5	3.5	3	0.4	5	18.19
762	1	0.64	784.0	343.0	220.5	3.5	4	0.4	5	18.16
763	1	0.64	784.0	343.0	220.5	3.5	5	0.4	5	17.88
764	1	0.62	808.5	367.5	220.5	3.5	2	0.4	5	16.54
765	1	0.62	808.5	367.5	220.5	3.5	3	0.4	5	16.44
766	1	0.62	808.5	367.5	220.5	3.5	4	0.4	5	16.48
767	1	0.62	808.5	367.5	220.5	3.5	5	0.4	5	16.64

In [119…

```
X= np.array(data.drop(["Y"] , axis= 1))
y=np.array(data["Y"])


y=y.reshape((768,1))
```

In [120…

```
#test and tran split
X_train , X_test , y_train , y_tset = sklearn.model_selection.train_test_split(X, y ,

linear= LinearRegression()

linear.fit(X_train ,y_train)

theta= linear.coef_
print(theta)

print("----------------------------")
#predict new data
y_new= linear.predict([[1,0.90, 418, 270,105,6,2,0,0]])
print(y_new)

print("----------------------------")
y_hat= linear.predict(X_test)
print(y_hat)


print("----------------------------")
#accuracy
from sklearn import metrics

print(linear.score(X_test , y_tset)*100)


print("----------------------------")
#r2score
print(round(metrics.r2_score(y_hat , y_tset)*100,2))
```

```
[[ 0.00000000e+00 -6.32407038e+01 -1.00707132e+12  1.00707132e+12
   2.01414264e+12  3.99204372e+00  3.56025723e-03  2.00250935e+01
   1.92440477e-01]]
----------------------------
[[6.24384218e+13]]
----------------------------
[[34.28144544]
 [32.68769544]
 [18.56269544]
 [10.43769544]
 [28.81269544]
 [34.75019544]
 [25.03144544]
 [26.12519544]
 [25.81269544]
 [28.21894544]
 [12.00019544]
 [12.87519544]
 [11.00019544]
 [33.59394544]
 [17.37519544]
 [25.40644544]
 [ 7.81269544]
 [11.75019544]
 [29.68769544]
 [34.75019544]
 [19.12519544]
 [29.68769544]
 [30.34394544]
 [29.31269544]
 [10.81269544]
 [15.00019544]
 [12.37519544]
 [26.12519544]
 [17.56269544]
 [12.81269544]
 [18.00019544]
 [25.03144544]
 [28.21894544]
 [26.50019544]
 [29.87519544]
 [31.56269544]
 [11.00019544]
 [40.46894544]
 [14.81269544]
 [34.28144544]
 [34.56269544]
 [36.06269544]
 [12.31269544]
 [29.87519544]
 [31.18769544]
 [29.31269544]
 [34.00019544]
 [34.06269544]
 [18.56269544]
 [33.87519544]
 [30.53144544]
 [28.81269544]
 [11.37519544]
 [ 8.18769544]
```

[14.37519544]
[32.31269544]
[11.37519544]
[11.62519544]
[8.37519544]
[10.18769544]
[35.50019544]
[25.81269544]
[30.34394544]
[8.00019544]
[28.03144544]
[25.81269544]
[27.93769544]
[29.31269544]
[32.12519544]
[39.87519544]
[16.12519544]
[17.00019544]
[15.00019544]
[34.00019544]
[39.87519544]
[32.50019544]
[30.06269544]
[14.00019544]
[12.25019544]
[18.93769544]
[13.06269544]
[26.68769544]
[26.12519544]
[11.81269544]
[22.84394544]
[31.00019544]
[14.00019544]
[29.50019544]
[25.81269544]
[12.06269544]
[28.40644544]
[30.93769544]
[40.06269544]
[16.25019544]
[25.40644544]
[31.37519544]
[12.93769544]
[29.12519544]
[14.81269544]
[14.62519544]
[12.75019544]
[11.75019544]
[32.87519544]
[35.31269544]
[11.37519544]
[15.18769544]
[16.06269544]
[9.06269544]
[18.00019544]
[9.43769544]
[7.50019544]
[28.03144544]
[29.12519544]
[17.75019544]

```
 [ 6.87519544]
 [28.40644544]
 [27.34394544]
 [18.18769544]
 [28.40644544]
 [10.06269544]
 [12.75019544]
 [30.59394544]
 [15.68769544]
 [34.06269544]
 [14.37519544]
 [12.87519544]
 [12.18769544]
 [17.62519544]
 [14.00019544]
 [10.43769544]
 [15.06269544]
 [31.75019544]
 [29.12519544]
 [17.37519544]
 [ 9.43769544]
 [40.06269544]
 [18.75019544]
 [13.43769544]
 [33.68769544]
 [11.56269544]
 [35.87519544]
 [34.46894544]
 [13.25019544]
 [13.25019544]
 [14.18769544]
 [13.12519544]
 [16.12519544]
 [10.06269544]
 [32.68769544]
 [12.56269544]
 [40.46894544]
 [34.46894544]
 [40.28144544]
 [22.84394544]]
----------------------------
91.2534249531597
----------------------------
90.55
```

In [121…

```python
#accuracy from the scrach

numerator=sum (( y_tset - y_hat)**2)
Denominator=sum (( y_tset -  y_tset.mean())**2)

rse= numerator / Denominator


R_square= 1 - rse


print(f" the accuracy of the algorhim is: {round(float(R_square*100), 2)} ")
```

```
 the accuracy of the algorhim is: 91.25
```