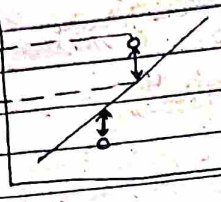
# Revision

① 


y (Actual)
ŷ (predicted By model)

model → Aim → to find Best-fit-Line.

$y = \theta_0 + \theta_1 x$

- target value / o/p variable
- Input var / feature

$\theta_0$ : offset / intercept
$\theta_1$ : slope / rate of change

Dataset
Trainning set
Testing set

$(x, y)$ → one trainning example
$(x^{(i)}, y^{(i)})$ → $i^{th}$ trainning example.

n : nos of features
m : nos of records / total nos of examples

★ hypothesis : $h_\theta(x) = \theta_0 + \theta_1 x$

$\theta$'s : parameter

MSE → mean sq. error $\frac{1}{n}\sum_{i=1}^{n}(y^i - \hat{y}^i)^2$

parameter estimation of LOR
- ① OLS ordinal least sq. (Mathematical approx)
- Gradient Descent (optimization Approx)

① OLS

$\hat{y} = \theta_0 + \theta_1 x$

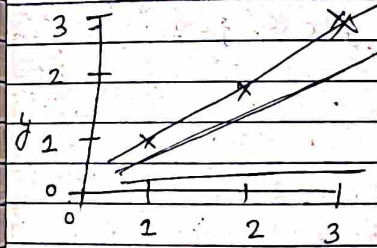
$\theta_1 = \frac{\sum_{i=1}^{n}(x^i - \bar{x})(y^i - \bar{y})}{\sum_{i=1}^{n}(x^i - \bar{x})^2}$

$\theta_0 = \bar{y} - \theta_1 \bar{x}$

gradient descent :

| hypothesis | Simplified |
|---|---|
| $h_\theta(x) = \theta_0 + \theta_1 x$ | $h_\theta(x) = \theta_1 x$ |
| parameters : $\theta_0, \theta_1$ | $\theta_1$ |
| cost func$^n$ J | cost func$^n$ J |
| $J(\theta_0, \theta_1) = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$ | $J(\theta_1) = \frac{1}{2m}\sum_{i=1}^{m} h_\theta(x^{(i)} - y^{(i)})^2$ |
| goal → minimize $J(\theta_0, \theta_1)$ $\theta_0, \theta_1$ | goal → minimize $J(\theta_1)$ $\theta_1$ |

$h_\theta(x)$
(for fixed $\theta_1$, this is a func$^n$ of $x$)

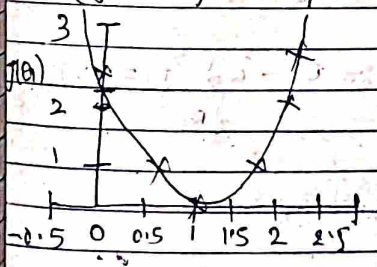



$J(\theta_1)$

$\frac{1}{2(3)}\left[(1-1)^2 + (2-2)^2 + (3-3)^2\right]$

No error

$J(\theta_1)$
(func$^n$ of the parameter $\theta_1$)



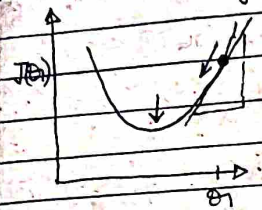

$m = 0.5$: $\frac{1}{2(3)}\left[(0.5\times1-1)^2 + (0.5\times2-2)^2 + (0.5\times3-3)^2\right]$

$m = 1.5$: $\frac{1}{2(3)}\left[(1\times1-1)^2 + (1\times2-2)^2 + (1\times3-3)^2\right]$

$m = 1.5$: $\frac{1}{2(3)}\left[(1.5\times1-1)^2 + (1.5\times2-2)^2 + (1.5\times3-3)^2\right]$

## gradient descent intuition

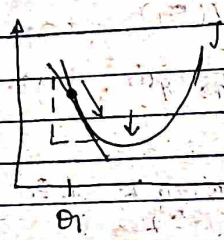
$J(\theta_1)$ $(\theta_1 \in \mathbb{R})$




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

$\frac{d}{d\theta_1} J(\theta_1) \geq 0$

$\theta_1 := \theta_1 - \alpha \cdot (\text{positive numb})$




$\frac{dJ(\theta_1)}{d\theta_1} \leq 0$

$\theta_1 := \theta_1 - \alpha (\text{neg numb.})$

## gradient descent algo :

repeat until convergence {

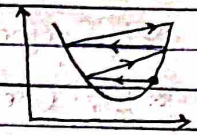
$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$ (simultaneously upd $j=0$ & $j=1$)

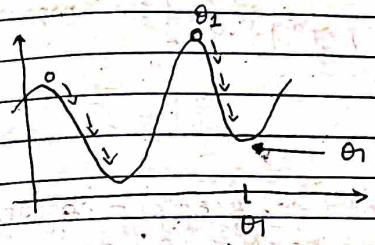
}

| correct : simultaneous update | incorrect |
|---|---|
| $temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ | $temp0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_0} J(\theta_0, \theta_1)$ |
| $temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ | $\theta_0 := temp0$ |
| $\theta_0 := temp0$ | $temp1 := \theta_1 - \alpha \frac{\partial}{\partial \theta_1} J(\theta_0, \theta_1)$ |
| $\theta_1 := temp1$ | $\theta_1 := temp1$ |

if $\alpha$ is too small gradient descent can be slow

if $\alpha$ is too large gradient descent can overshoot the minimum.






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

| o gradient descent algo | o linear regression model |
|---|---|
| repeat until convergence { | $h_\theta(x) = \theta_0 + \theta_1 x$ |
| $\theta_j := \theta_j - \alpha \frac{d}{d\theta_j} J(\theta_0, \theta_1)$ (for $j=1$ and $j=0$) | $J(\theta_0, \theta_1) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$ |
| } | |

$\frac{d}{d\theta_0} J(\theta_0, \theta_1) \rightarrow \frac{1}{2m} \sum_{i=1}^{m} 2 [\theta_0 + \theta_1 x^{(i)} - y^{(i)}]^{2-1} [1]$

$\rightarrow \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$

$\frac{d}{d\theta_1} J(\theta_1, \theta_1) \rightarrow \frac{1}{2m} \sum_{i=1}^{m} 2 [\theta_0 + \theta_1 x^{(i)} - y^{(i)}] x^{(i)}$

$\rightarrow \frac{1}{m} \sum_{i=1}^{m} [x^{(i)} \theta_0 + \theta_1 x^{2(i)}]$

$\rightarrow \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$

gradient descent algo:

repeat until convergence {

$$\theta_0 := \theta_0 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})$$

$$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) \cdot x^{(i)}$$

} Update $\theta_0$ & $\theta_1$ simultaneously

As we approach local minimum, gradient descent will automatically take smaller steps. So, no need to decrease $\alpha$ over time.

★ Regression with Multiple features (variables):

| $x_1$ Size (feet)$^2$ | $x_2$ Number of bedrooms | $x_3$ Number of floors | $x_4$ Age of home (in yrs) | $y$ Price ($1000) |
|---|---|---|---|---|
| 2104 | 5 | 1 | 45 | 460 |
| 1416 | 3 | 2 | 40 | 232 |
| 1534 | 3 | 2 | 30 | 315 |
| 852 | 2 | 1 | 36 | 178 |

$\rightarrow m = 4$

notations: $n$ = number of features

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

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

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

hypothesis:

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

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$

i.e. $h_\theta(x) = 80 + 0.1 x_1 + 0.01 x_2 + 3 x_3 - 2 x_4$ → age.

(multivariate LoR) — $\theta_4$ : impact

for convinience of notation, define $x_0 = 1$

$(x_0^{(i)} = 1)$

$h_\theta(x) = \theta_0 x_0 + \theta_1 x_1 + \cdots + \theta_n x_n$

$$x = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1} \qquad \theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$$h_\theta(x) = \theta^T x$$

$$\underbrace{[\theta_0 \ \theta_1 \ \cdots \ \theta_n]}_{\theta^T} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{bmatrix}$$

$(n+1) \times 1$ matrix

Hypothesis: $h_\theta(x) = \theta^T x = \theta_0 x_0 + \theta_1 x_1 + \theta_2 x_2 + \dots + \theta_n x_n$

parameters: $\theta_0, \theta_1, \dots, \theta_n$ — $\theta$ ($n+1$ dimensional vector)

cost func: $J(\theta_0, \theta_1, \dots, \theta_n) = \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2$

$J(\theta)$

gradient descent:

repeat {

$\theta_j := \theta_j - \alpha \frac{d}{d\theta_j} J(\theta_0, \dots, \theta_n)$

}

(simultaneously update for every $j = 0, \dots, n$)

$\theta_0 := \theta_0 - \alpha \frac{d}{d\theta_0} J(\theta)$

$\rightarrow \frac{d}{d\theta_0} \left[ \frac{1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 \right]$

$\frac{d}{d\theta_0} \left[ \frac{1}{2m} \sum_{i=1}^{m} ((\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \dots + \theta_n x_n^{(i)}) - y^{(i)})^2 \right]$

$2 \times \frac{1}{2m} \sum_{i=1}^{m} [(\theta_0 x_0 + \theta_1 x_1 + \theta_n x_n) - y^{(i)}] [x_0^{(i)}]$

$\frac{1}{m} \sum_{i=1}^{m} [x_0^{(i)} (h_\theta(x^{(i)}) - y^{(i)})]$

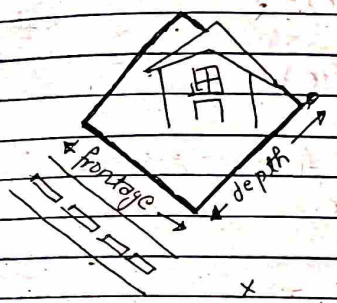
* cost drop at every term.

$\theta_j := \theta_j - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$

polynomial regression:

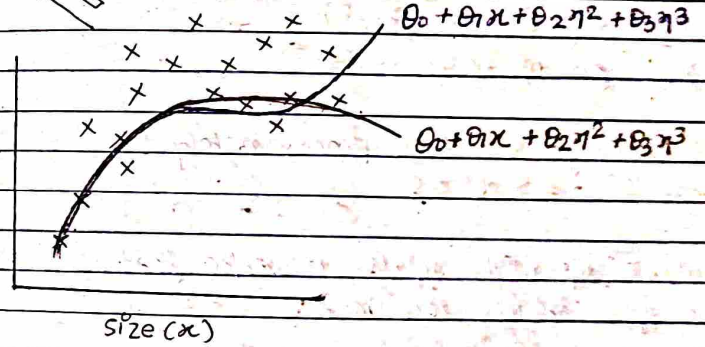
need of it: house price prediction

$h_\theta(x) = \theta_0 + \theta_1 \times \underbrace{\text{frontage}}_{x_1} + \theta_2 \times \underbrace{\text{depth}}_{x_2}$




Area = frontage * depth

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




$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$

$\rightarrow \theta_0 + \theta_1 (size) + \theta_2 (size)^2 + \theta_3 (size)^3$

$x_1 \rightarrow (size)$

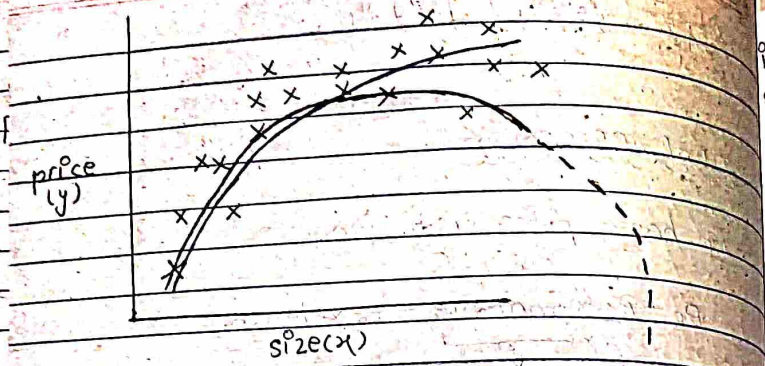
$x_2 \rightarrow (size)^2$

$x_3 \rightarrow (size)^3$

size: 1 - 1000

$size^2$: 1 - 1000,000

$size^3$: 1 - $10^9$




$h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2(size)^2$

$h_\theta(x) = \theta_0 + \theta_1(size) + \theta_2(\sqrt{size})$

Feature scaling : makes sure features are on similar scale.

eg $x_1$ : size (0-2000 feet$^2$)
$x_2$ : nos of bedrooms (1-5)

Get every feature into approximately a range : $-1 \le x_i \le 1$

replace $x_i$ with $x_i - \mu$ to make features have approximately zero mean
(don't apply to $x_0 = 1$)

eg : $x_1 = \frac{size - 1000}{2000}$ $\quad -0.5 \le x_1 \le 0.5$

$-0.5 \le x_2 \le 0.5$

$x_2 = \frac{bedrooms - 2}{4}$

$x_1 \leftarrow \frac{x_1 - \mu_1}{s_1}$ ($\mu_1$ → avg. value of $x_1$ in training set; $s_1$ → range (max-min) or standard deviation)

$x_2 \leftarrow \frac{x_2 - \mu_2}{s_2}$

```
import pandas as pd
import numpy as np

df = pd.read_csv('placement.csv')
df
```

| | cgpa | package |
|---|---|---|
| 0 | 6.89 | 3.26 |
| 1 | 5.12 | 1.98 |
| 2 | 7.82 | 3.25 |
| 3 | 7.42 | 3.67 |
| 4 | 6.94 | 3.57 |
| | ... | ... |
| | ... | ... |
| 200 | 6.22 | 2.33 |

```
df.iloc[:,0:1]
```
(iloc take indexes to perform slicing) — every row, first col

| | cgpa |
|---|---|
| 0 | 6.89 |
| 1 | 5.12 |
| 2 | 7.82 |

200 rows × 1 columns

⇒ returns type pandas.core.frame.DataFrame

```
df.iloc[:,-1]
```
0 3.26
1 1.98
2 3.25
...
⇒ returns type pandas.core.series

```
y = df.iloc[:,-1].values
```
→ array([3.26, 1.98, 3.25, ..., 2.33])
one-dimensional target values

```
X = df.iloc[:,0:1].values
```
→ array([[6.89], [5.12], ..., [2.49]])
two-dimensional

```
# feature scaling

def normalize_feature(feature):
    return (feature - np.mean(feature))/ np.std(feature)

X = normalize_feature(df.iloc[:,0:1].values)
```
→ array([[-0.094213], [-1.75348666], ..., [1.25366987]]).

When we say "similar scale" we mean that the value of different features are within a comparable range

In the context of feature scaling, the goal is to transform the values of each feature so that they proportionally comparable to each other. This ensure no single feature dominates the other due to differences in their magnitudes

① Min-Max scaling (Normalization)

$$X_{normalized} = \frac{X - X_{min}}{X_{max} - X_{min}}$$

range b/w 0 to 1

② Standardization (Z-score normalization)

$$X_{standardized} = \frac{X - \mu}{\sigma}$$

This transforms value to have mean ($\mu$) of 0 and a standardized deviation ($\sigma$) of 1

```
def hypothesis(theta0, theta1, x):
    return theta0 + theta1 * x
```

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

```
def cost_function(theta0, theta1, x, y):
    m = len(y)
    h = hypothesis(theta0, theta1, x)
    return (1/(2*m)) * np.sum((h-y)**2)
```

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

$(x, y)$ → example of data

$(x^{(i)}, y^{(i)})$ → $i^{th}$ example dataset

```
def gradient_descent(theta0, theta1, x, y,
    learning_rate = 0.0001, iterations = 1000,
    convergence_threshold = 1e-4):
    m = len(y)
    costs = [] # to store cost value at each iteration
```

for iteration in range(iterations):

h = hypothesis(theta0, theta1, x)

delta_theta0 = (1/m) * np.sum(h-y)

$$\frac{\partial J(\theta)}{\partial \theta_0} = \frac{1}{2m}\sum_{i=1}^{m}(h_\theta(x^{(i)}) - y^{(i)})^2$$

$$\Downarrow \quad \theta_0 + \theta_1 x$$

$$\Rightarrow \frac{1}{m}\sum_{i=1}^{m}[(h_\theta(x^{(i)}) - y^{(i)})]^2$$

delta_theta1 = (1/m) * np.sum((h-y) * x)

$$\frac{\partial J(\theta)}{\partial \theta_1} = \frac{1}{m}\sum_{i=1}^{m}[x(h_\theta(x^{(i)}) - y^{(i)})]$$

theta0 -= learning_rate * delta_theta0

theta1 -= learning_rate * delta_theta1

repeat until convergence {

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1)$$ (simultaneously update $j=0$ & $j=1$)

}

# check for convergence based on gradients

if np.abs(delta_theta0) < convergence_threshold

and np.abs(delta_theta1) < convergence_threshold:

print(f"Converged after {iteration} iterations")

break # when meet minima

# Store cost for every iteration

cost = cost_function(theta0, theta1, x, y)

costs.append(cost)

# print cost for every 100 iteration

if iteration % 100 == 0:

print(f"Iteration {iteration}, cost: {cost}")

return theta0, theta1, costs

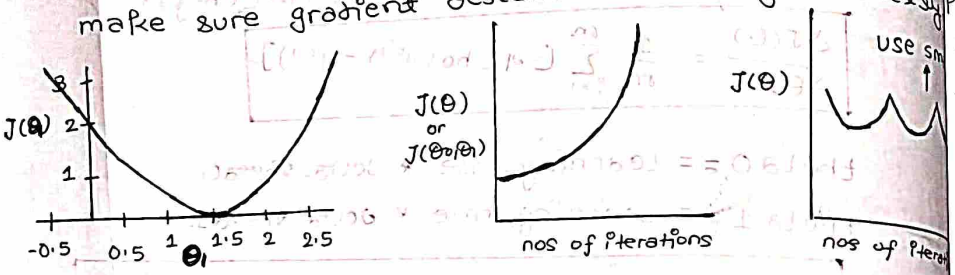
- Convergence criteria

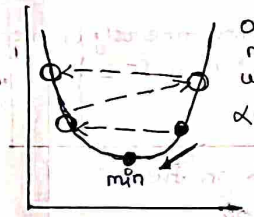
  - for convex functions, optimum occurs when $\frac{\partial}{\partial \theta} J(\theta) = 0$

  - In practice, stop when $\frac{\partial}{\partial \theta} J(\theta) < \epsilon$ → $1e^{-3}$, $1e^{-4}$, $1e^{-5}$

  - As we approach a local minimum, gradient descent will automatically takes smaller steps. So, no need to decrease ($\alpha$) over time.

make sure gradient descent is working correctly




for sufficiently small $\alpha$, $J(\theta)$ should decrease on every iteration

But, if $\alpha$ is too small, gradient descent can be slow to converge

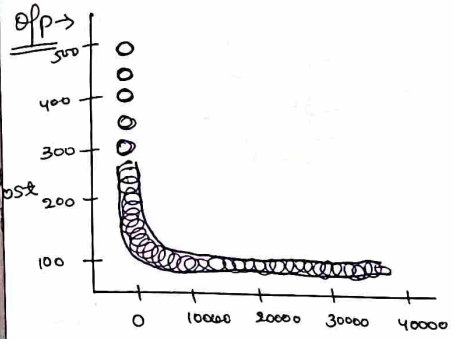
```
# Initialize theta0 and theta1.
theta0 = 1.0 or 0
theta1 = 1.0

# Run gradient-descent
final_theta0, final_theta1, costs =
  gradient_descent(theta0, theta1, x, y)
```

plotting the cost over iterations.

```
iterations_range = range(0, len(costs) * 100, 100)
plt.plot(iterations_range, costs, marker='o')
plt.title('Cost over iterations')
plt.xlabel('Iterations')
plt.ylabel('Cost')
plt.show()

# print the final parameters
print(f"Final parameters: theta0 = {final_theta0},
      theta1 = {final_theta1}")
```




```
Iteration 0, Cost: 526.28
Iteration 100, Cost: 56.01
Iteration 200, Cost: 47.74
.....................
Iteration 600, Cost: 47.59
Iteration 700, Cost: 47.59

Converged after 753 iterations
```

Final parameters: theta0: 2.996, theta1: $2.42e^{-07}$

(X) = np.array([[1,2,3],
[1,3,4],
[1,4,5]])

↓ 3x3 dimension

(feature matrix)

| x1 | x2 | x3 | Y |
|---|---|---|---|
| 1 | 2 | 3 | 1 |
| 1 | 3 | 4 | 2 |
| 1 | 4 | 5 | 3 |

y = np.array([1,2,3])

(target variable)

<u>hypothesis</u> ~~$h_\theta(x) = \theta_0 + \theta_1 x$~~

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3 + \cdots + \theta_n x_n$

$x^{(i)}$ : input feature of $i^{th}$ trainning example

$$x^{(2)} = \begin{bmatrix} 1 \\ 3 \\ 4 \end{bmatrix}$$

$x_j^{(i)}$ : value of feature $j^{th}$ in $i^{th}$ trainning example.

$x_3^{(2)} = 4$

for convinence of notation define $x_0 = 1$ or $x_0^{(i)} = 1$

$h_\theta(x) = \theta_0 \underset{\mapsto 1}{x_0} + \theta_1 x_1 + \theta_2 x_2 + \theta_3 x_3$

$$X = \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

# add column of ones to array 'X'.

X = np.c_[np.ones(X.shape[0]), X]

- np.ones(X.shape[0]) : creates an array of ones with the same nos of rows as the original array 'X'.
  X.shape[0] represents nos of rows in array 'X'.
- np.c_ : concatenate array of ones with original array X.

o/p -> array([[1, 1, 2, 3],
[1, 1, 3, 4],
[1, 1, 4, 5]])

$$X^2 = \begin{bmatrix} 1 \\ 1 \\ 3 \\ 4 \end{bmatrix}$$

(X) → Now → 3x4 dimensional

$$\theta = \begin{bmatrix} \theta_0 \\ \theta_1 \\ \theta_2 \\ \vdots \\ \theta_n \end{bmatrix} \in \mathbb{R}^{n+1}$$

$h_\theta(x) = \theta^T x$

$$[\theta_0\ \theta_1\ \theta_2\ \dots\ \theta_n] \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ \vdots \\ x_n \end{bmatrix}$$

$(n+1) \times 1$ matrix

```
def compute_cost(X, y, theta):
    m = len(y)
    h = np.dot(X, theta)
    J = (1/(2*m)) * np.sum((h-y)**2)
    return J
```

```
h = np.dot(X, theta)
```

o/p → array([[ 1θ, 1θ, 2θ, 3θ ],
[ 1θ, 1θ, 3θ, 4θ ],
[ 1θ, 1θ, 4θ, 5θ ]])

Cost func: $J(\underbrace{\theta_0, \theta_1, \theta_2, \dots, \theta_n}_{(n+1)\text{ dimensional vector}}) = \frac{1}{2m} \sum_{i=1}^{m} h_\theta(x^{(i)}) - y^{(i)})^2$

or

$J(\theta)$

```
theta = np.zeros(X.shape[1])
```
↳ array([0,0,0,0])

- The length of theta is set to the numbers of columns in updated matrix `X`.

## Gradient Descent

repeat {

$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$ ($\theta_0, \theta_1, \theta_2, \dots, \theta_n$)

(simultaneously update for every $j = 0, 1, 2, \dots, n$).

$\theta_0 := \theta_0 - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$

$\rightarrow \frac{\partial}{\partial \theta_0} \left[ \frac{1}{2m} \times 2 \sum_{i=1}^{m} [(\theta_0 x_0^{(i)} + \theta_1 x_1^{(i)} + \theta_2 x_2^{(i)} + \theta_3 x_3^{(i)}) - y^{(i)})] (x_0^{(i)}) \right]$ ($x_0^{(i)} = 1$)

$\rightarrow \frac{1}{m} \sum_{i=1}^{m} [(h_\theta(x^{(i)}) - y^{(i)})]^2$

similarly,

$\theta_1 := \theta_1 - \alpha \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)})^2 x_1^{(i)}$

$\theta_2$ & $\theta_3$ also calculate.

```
def gradient_descent(X, y, theta, alpha, num_iterations,
                     print_cost=True, print_every=100):
    m = len(y)
    J_history = []
    for i in range(num_iterations):
        h = np.dot(X, theta)
        errors = h - y
        gradient = (1/m) * np.dot(X.T, errors)
        theta = theta - alpha * gradient
        J = compute_cost(X, y, theta)
        J_history.append(J)
```

if print_cost and i % print_every == 0:

print("Cost after iteration {i}:

return theta, J_history

h = np.dot(X, theta) → initially = 0 → array([0, 0, 0])

errors = h - y → array([-1, -2, -3])

gradient = (1/m) * np.dot(X.T, errors)

initially $X_0 \rightarrow \begin{bmatrix} 1 \\ 1 \\ 1 \\ 1 \end{bmatrix}$ [-1 -2 -3] ← $\theta_0$

as in loop we get others $\theta_1^T x_4$, $\theta_2^T x_2$ and many

model performance evaluation :

Loss function $L(y, h_\theta(x))$

$L$(Actual value, Predicted value)

Examples :

(assuming loss for underpredicting = overpredicting)

Absolute Error → $L(y, h_\theta(x))$

$\Rightarrow |(y - h_\theta(x))|$

Mean Absolute Error → $L(y, h_\theta(x))$

$\Rightarrow (y - h_\theta(x))^2$

★ Trainning Error computation :

① Define a loss function $L(y, h_\theta(x))$

ie squared error, absolute error

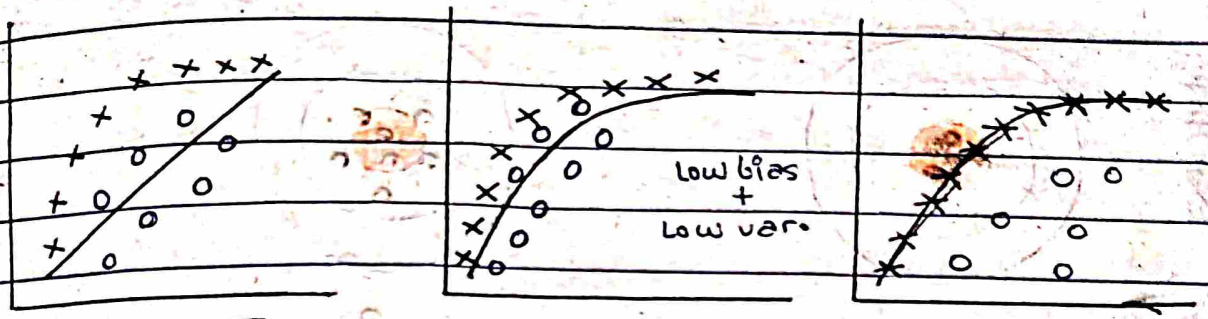
② Trainning error = avg. loss on house in trainning set

$$\Rightarrow \frac{1}{n} \sum_{i=1}^{n} L(y^{(i)} - h_\theta(x^{(i)}))$$

# Bias - Variance - Tradeoff



Low testing error




Under-fitting → $\theta_0 + \theta_1 x_1$

Appropriate-fitting → $\theta_0 + \theta_1 x_1 + \theta_2 x_1^2$

Over-fitting → $\theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_1^3$

× : trainning data
○ : testing data

- Trainning error is high (High bias)
- poor testing on unseen data (low variance)

- Trainning error is less (Low biased)
- Excellent perform on trainning set but poor perform -ance with unseen data. (High variance)

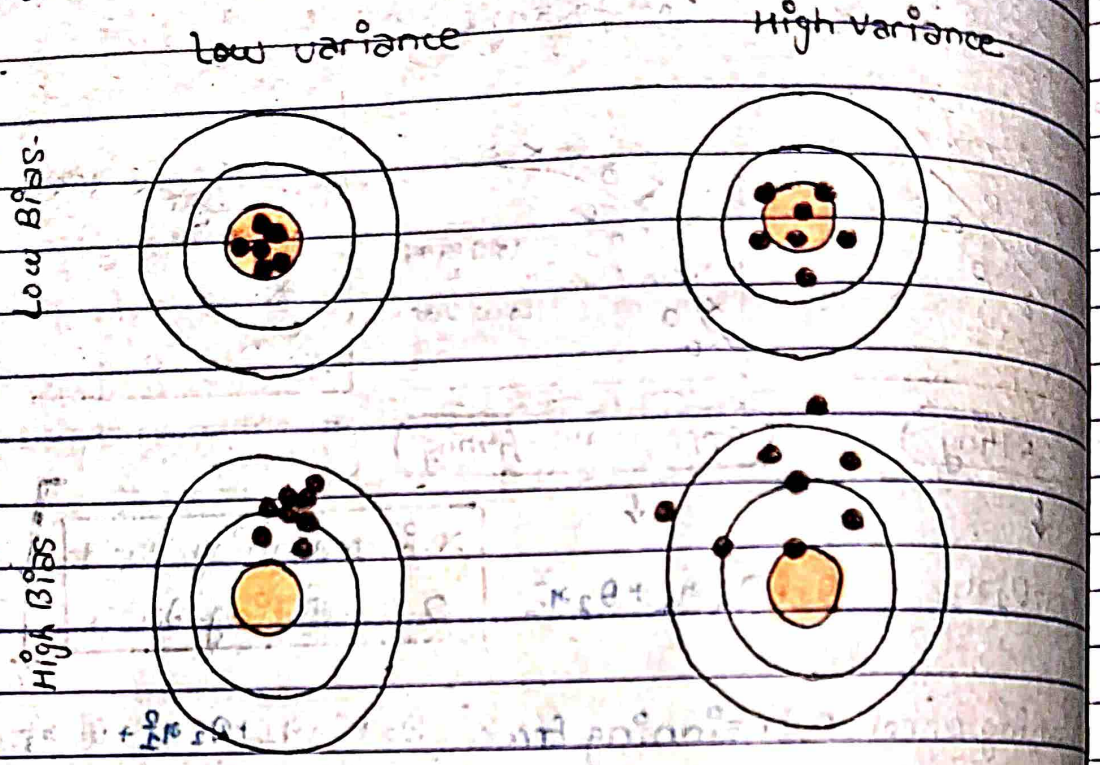
- Trainning error is zero (low biased)

→ If the algorithm is too simple (hypothesis with linear equation) then it may be on high bias and low variance condition and thus is error prone

→ If algorithm fits too complex (hypothesis with high degree equation) then it may be on high variance and low bias.

→ In the latter condition the new entries will not perform well, Well, there's something b/w both of these conditions known as Bias-Variance Tradeoff

There are four possible combination of bias and variance —




(1) Low bias, Low variance : The combination of low bias & low variance shows an ideal ml model. However, it is not possible practically.

(2) Low Bias, High Variance : model predictions are inconsistent and accurate on average. This occur when a model learns with a large nos of parameters and hence leads to an overfitting.

(3) High Bias, Low Variance : predictions are consistent but inaccurate on average. This case occur when model learns with a ~~large~~ (few) nos of parameters and hence leads to underfitting.

(4) High bias, High Variance : With high variance & high bias prediction are inconsistent and inaccurate on average.

## ★ Regularization in Machine Learning :

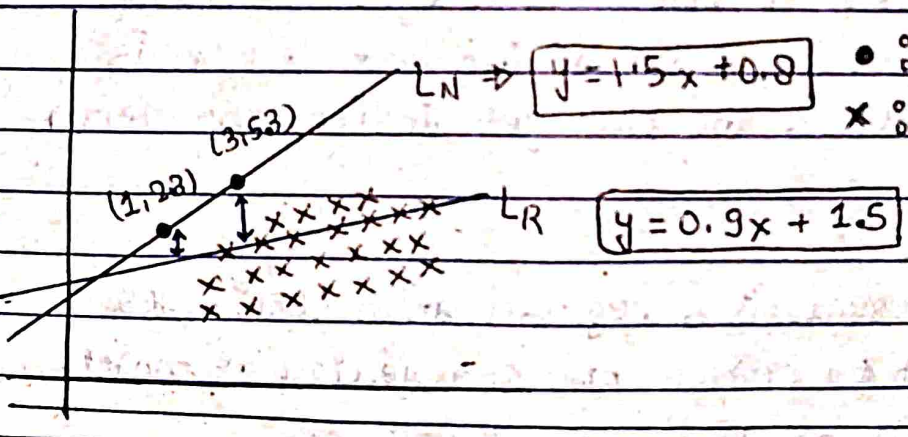
While developing ML models you must have encountered a situation in which the trainning accuracy of model is low but the testing accuracy is too low. This is the case of Overfitting.

Regularization helps us to solve the problem of overfitting which reduces error by fitting the function appropriately on the given trainning set.

### Regularization techniques →

1. Lasso regularization - L1 regularization
2. Ridge regularization - L2 regularization
3. Elastic-Net regularization - L1 & L2 regularization

### → Ridge Regularization



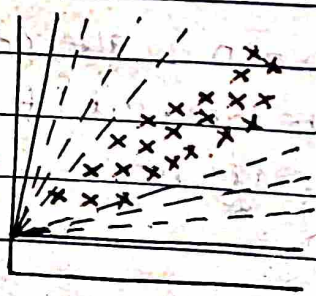

**Note :**

→ Oversitting ($m \to \infty$)

very much depend on 'm'.

$y = mx + b$

m : scale factor to each input value

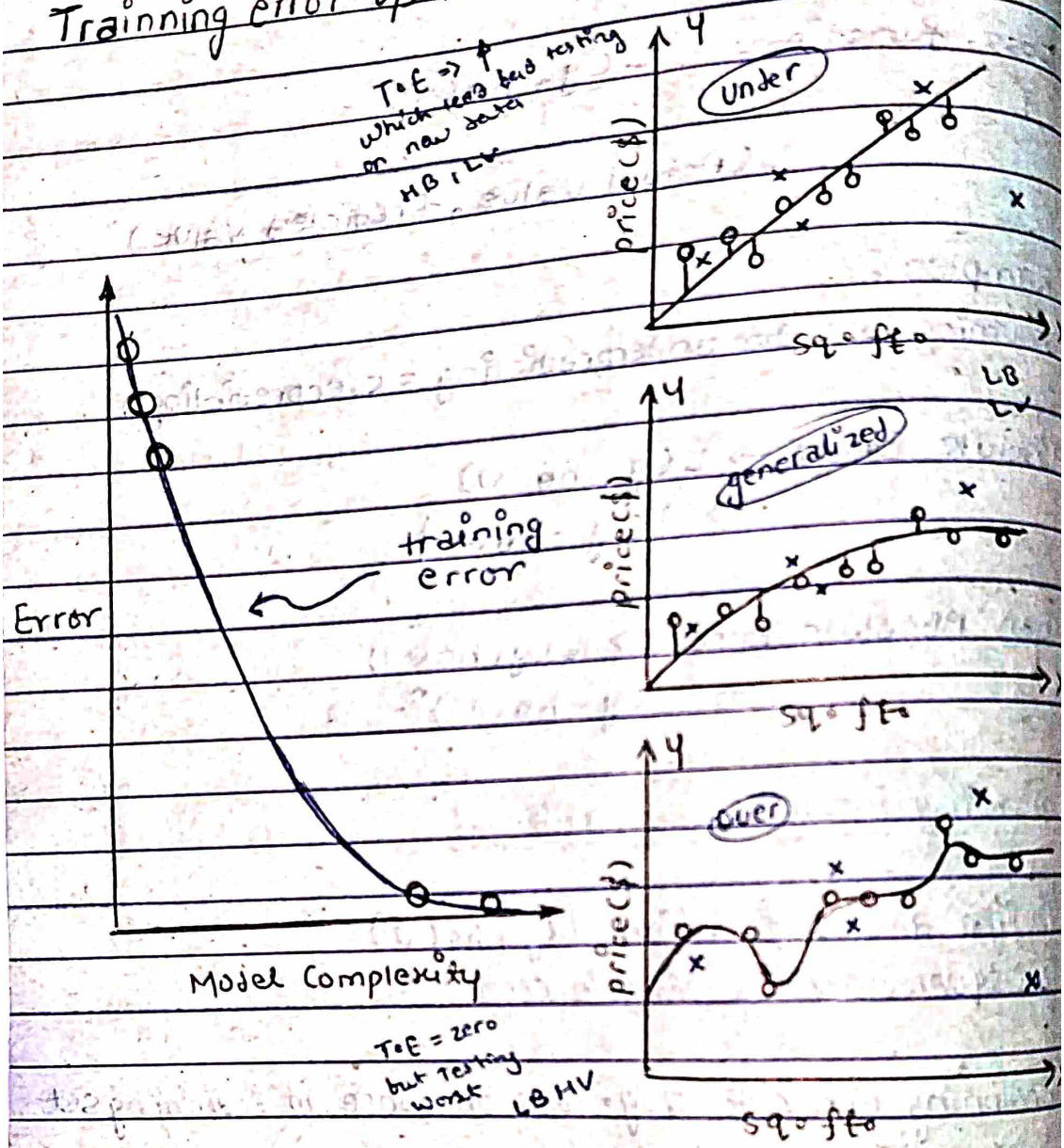


→ Undersitting ($m \to 0$)

target value is not so depending on 'm'

means, Overfitting कम करने के लिए slope 'm' को थोड़ा कम रखो।

# Trainning error v/s model Complexity :

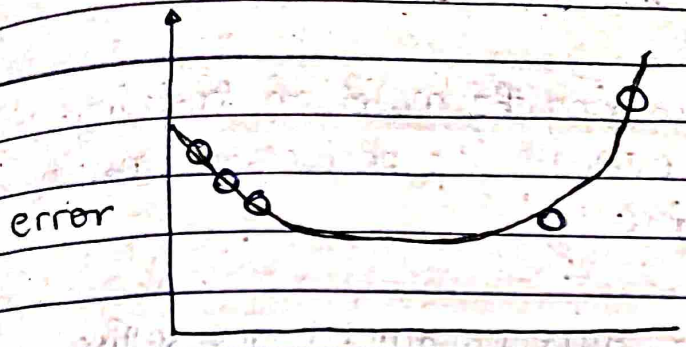



- Is trainning error a good measure of predective performance ?

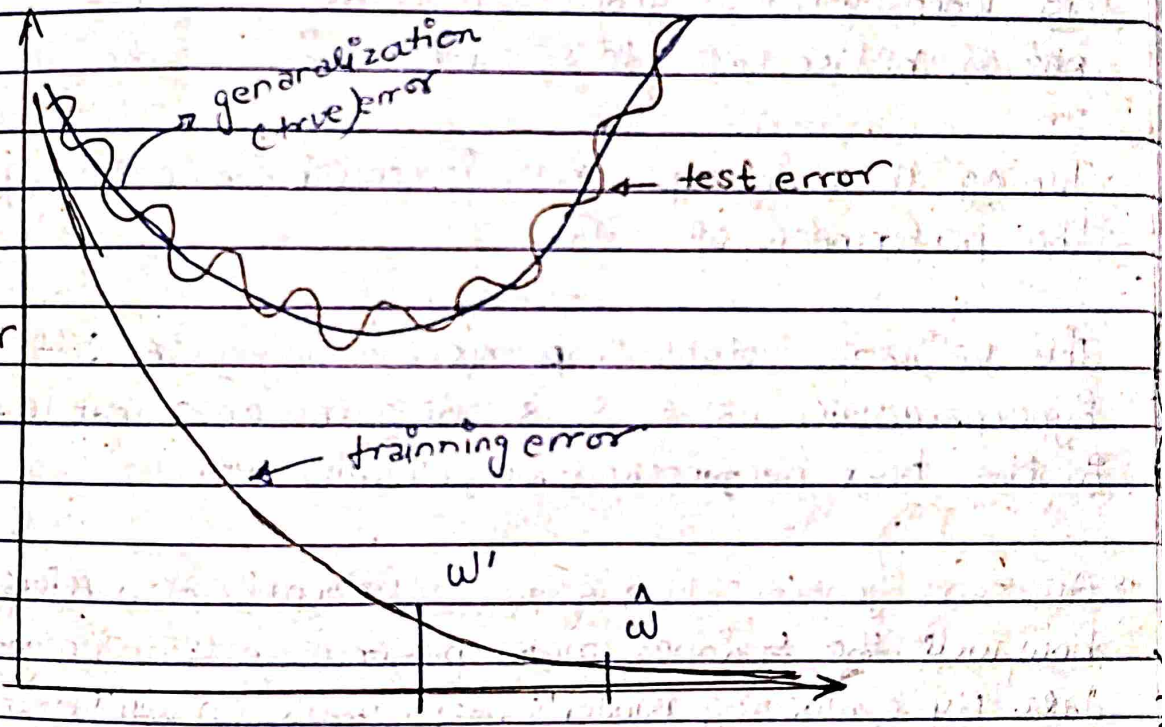
- Trainning Error is overly optimistic Because θ was fit to training data.
- small trainning error ⇏ good predictions
- unless Trainning data include everything you might ever see.

Generalization error

~~Training, Store and Test Error~~ v/s model complexity




model complexity




model complexity

Overfitting if : ~~if there exist a model with estimated parameter w~~

1. training error ($\hat{w}$) < training error ($w'$)
2. ~~true error ($\hat{w}$) > true error ($w'$)~~

regularization

$h_\theta(x) = \theta_0 + \theta_1 x_1 + \theta_2 x_1^2 + \theta_3 x_3^3 + \dots + \theta_n x_n$

$x_2, x_3 \rightarrow$ unimportant

① remove manually

② use model selection algo

③ make weff $\theta_j$ small ($\approx 0$) § regularization §.

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

repeat till conv §

$$\theta_j = \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta_0, \theta_1, \theta_2, \dots \theta_n)$$

§

update simultaneously
($j = 0, 1, 2, \dots n$)

$$\frac{\partial J(\theta)}{\partial \theta_0} \rightarrow \frac{2 \times 1}{2m} \sum_{i=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_0^{(i)}$$

$$\rightarrow \frac{1}{m} \sum_{i=1}^{m} (h_\theta(x^{(i)} - y^{(i)}) x_0^{(i)}$$

$$\theta_j = \theta_j - \alpha \left[ \frac{1}{m} \sum_{i=1}^{m} [h_\theta(x^{(i)} - y^{(i)}] x_j^{(i)} - \frac{\lambda}{m} \sum_{j=1}^{m} \theta_j \right]$$

$$\theta_j \left(1 - \alpha \frac{\lambda}{m}\right) - \alpha \frac{1}{m} \sum_{j=1}^{m} (h_\theta(x^{(i)}) - y^{(i)}) x_j^{(i)}$$