

Simple Linear Regression

- Supervised ML Algorithm
- Output column values will be numerical

3 Types of Linear Regression

1. Simple LR

- 1 Input, 1 Output column
- eg. CGPA vs Package

2. Multiple LR

- More than 1 Input, 1 Output column
- Input= CGPA, Gender, Location, Output = Package

3. Polynomial LR

- This allows fitting curved lines to data instead of just straight lines, still using a linear model but with transformed input features
- It allows for capturing non-linear relationships by adding polynomial terms (e.g., x^2 , x^3) to the linear model, which enables it to fit more complex curves.

Simple Linear Regression

- We have this data

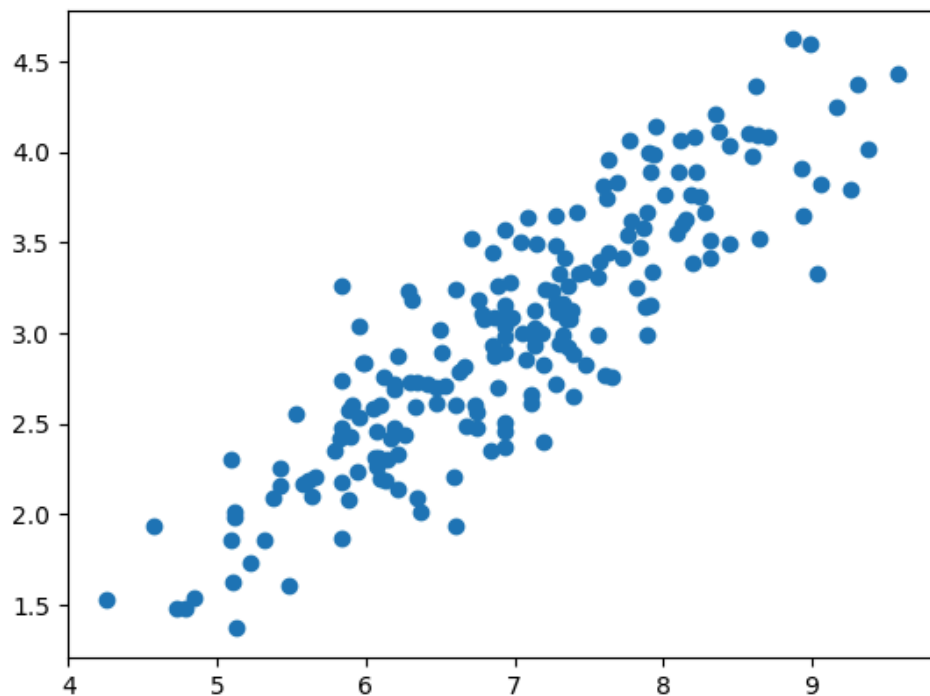
```
df = pd.read_csv('placement.csv')
```

```
df.head()
```

	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

- We want to build a model in which, when we give it cgpa, it will give us the package

```
plt.scatter(df.cgpa, df.package)
```



- LR draws a best fit line which passes through most of the points.

Apply Simple LR on the above data

When training a machine learning model, we must split our dataset into two parts:

- 1 **Training Set** – Used to train the model (learn patterns).
- 2 **Testing Set** – Used to evaluate the model (check accuracy).

- First, separate x & y

```
X= df.iloc[:,0:1]
```

	cgpa
0	6.89
1	5.12
2	7.82
3	7.42
4	6.94

```
Y = df.iloc[:,1]
```

0	3.26
1	1.98
2	3.25
3	3.67
4	3.57
...	
195	2.46
196	2.57
197	3.24
198	2.86

The syntax `df.iloc[:, 0:1]` is selecting:

- **Rows:** `:` indicates **all rows**.
- **Columns:** `0:1` refers to the **first column** (index 0) but with a slice, which means it selects from index 0 to index 1 **excluding 1** (a slice is half-open, meaning it includes the start but not the end index).
- `Y = df.iloc[:, 1]`: This selects all rows (`:`) and the second column (`1`) from the DataFrame `df`.
 - The `1` syntax selects the second column **as a 1D array** (just the values).
 - **Columns:** `1` refers to the **second column** (index 1).

We divide data in Training & Test

train_test_split

```
from sklearn.model_selection import train_test_split
```

```
x_train, x_test, y_train, y_test = train_test_split(X, Y, test_size=0.2, random_state=2)
```

`X` → Feature data

`Y` → Target data

`test_size=0.2` → 20% Sampling

`random_state=2` → To reproduce data

- `x_train` : The features (input data) for the **training set**.
- `x_test` : The features (input data) for the **testing set**.
- `y_train` : The target variable (output) for the **training set**.
- `y_test` : The target variable (output) for the **testing set**.

Variable	What It Represents?	What It Contains?
<code>x_train</code>	Training Features (Inputs)	80% of <code>X</code> (Independent Variables)
<code>x_test</code>	Testing Features (Inputs)	20% of <code>X</code> (Independent Variables)
<code>y_train</code>	Training Labels (Outputs)	80% of <code>Y</code> (Target Variables)
<code>y_test</code>	Testing Labels (Outputs)	20% of <code>Y</code> (Target Variables)

Why Do We Split the Data?

- **Training Set:** Used to teach the model. The model learns patterns from this data.
- **Testing Set:** Used to evaluate the model. Since the model has never seen this data before, it tests how well the model generalizes to new, unseen data.

Example

Let's say you have a dataset of 1000 houses (`X` has 1000 rows of features, and `Y` has 1000 house prices).

- After splitting with `test_size=0.2` :
 - **Training Set:** 800 houses (`x_train` and `y_train`).
 - **Testing Set:** 200 houses (`x_test` and `y_test`).

Import linear Regression

```
from sklearn.linear_model import LinearRegression
```

Make an Object

```
lr = LinearRegression()
```

Fit the data in the object

```
lr.fit(x_train, y_train)
```

- Now, we will give cgpa to our model and it will predict the package.

```
lr.predict(x_test.iloc[0].values.reshape(1,1))
```

Output: array([3.89111601])

`.iloc[0]` → used to select the **first row**

`.values` → This converts the first row (or element) into a NumPy array.

- This step removes any Pandas-specific structures (like DataFrame or Series objects) and converts them to a simple array, which is what the `predict()` method expects.

`.reshape(1, 1)` reshapes the 1D array into a 2D array with 1 row and 1 column.

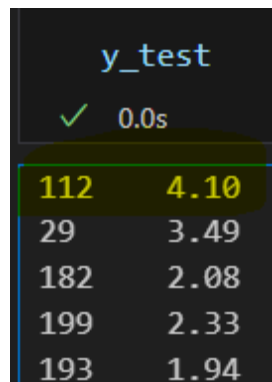
- This is needed because the model (`lr.predict()`) expects the input to be a **2D array** (even if it's just a single feature for a single prediction).

Finally,

`lr.predict()` is used to make a prediction using the trained **linear regression model** (`lr`).

Output: array([3.89111601]) → **First student's package will be 3.89**

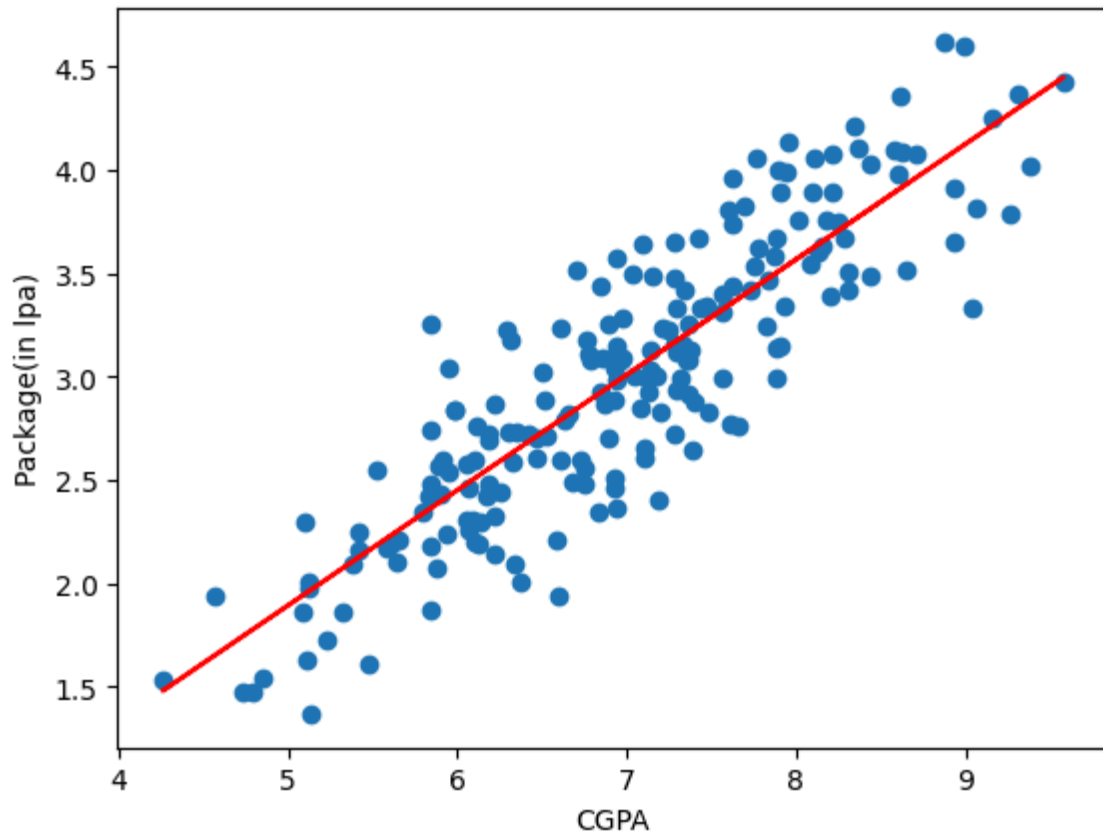
Actual package → 4.10



The screenshot shows a Jupyter Notebook terminal with a dark background. At the top, it says 'y_test' in blue. Below that, a green checkmark and '0.0s' indicate a successful execution. A table of data is displayed with yellow text on a dark background. The first row is highlighted with a green box, showing the student ID '112' and the predicted package value '4.10'. The subsequent rows show other student IDs and their predicted package values.

	y_test
✓ 0.0s	
112	4.10
29	3.49
182	2.08
199	2.33
193	1.94

```
plt.scatter(df['cgpa'],df['package'])  
plt.plot(x_train,lr.predict(x_train),color='red')  
plt.xlabel('CGPA')  
plt.ylabel('Package(in lpa)')
```



`plt.scatter(df['cgpa'], df['package'])` → Plots the original data

`plt.plot(x_train, lr.predict(x_train), color='red')`

- This line overlays the **predicted values from a linear regression model** (`lr`) onto the scatter plot.
- `x_train` : These are the **feature values** (CGPA values) used in the training set.
- `lr.predict(x_train)` : This generates the predicted `package` values for the training data based on the linear regression model (`lr`).
 - The model uses the `x_train` data (CGPA values) to predict the corresponding `package` values.

Training data vs Predicted Data 🙌

Slope of the line

`m= lr.coef_` → `array([0.55795197])`

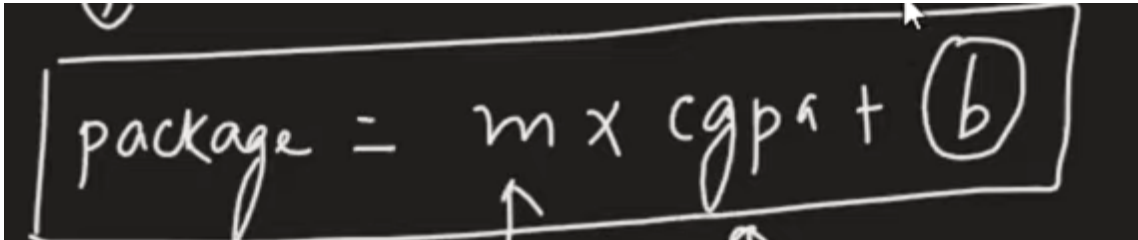
Y Intercept

`b= lr.intercept_` → `-0.8961119222429144`

Line equation → $y = mx + b$

Here, **y = Package**

x= cgpa



A photograph of a blackboard with the linear regression equation $\text{package} = m \times \text{cgpa} + b$ written in white chalk. The variable 'package' is on the left, followed by an equals sign, then 'm' multiplied by 'cgpa', plus a circled 'b'. Arrows point from the variables to their corresponding parts in the equation: an arrow from 'package' to the left side, an arrow from 'm' to the slope coefficient, an arrow from 'cgpa' to the feature variable, and an arrow from 'b' to the intercept term.

Closed-Form vs. Non-Closed-Form Solutions in Machine Learning

- 1 Closed-Form Solution** → A **direct formula** gives the exact answer.
- 2 Non-Closed-Form Solution (Iterative Methods)** → Uses **repeated calculations** (iterations) to get an approximate answer.

What is a Closed-Form Solution?

- A **closed-form solution** is a formula that directly calculates the result **in one step** without needing iterations.
- Called **OLS (Ordinary least Squares)**
- **We don't perform differentiation/integration**

- **Example in Machine Learning:**

- The **Normal Equation** for **Linear Regression** is a **closed-form** solution.

- Formula:

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

Here, θ (theta) is the best-fit parameter that minimizes the error.

◆ **Advantages of Closed-Form Solutions:**

- ✓ **Exact Solution** – No approximation.
- ✓ **Fast** for small datasets.

◆ **Disadvantages of Closed-Form Solutions:**

- ✗ **Computationally Expensive** for large datasets (matrix inversion is slow).
- ✗ **Not always possible** (some problems don't have a closed-form solution).

2 What is a Non-Closed-Form Solution (Iterative Solution)?

A **non-closed-form solution** does not have a direct formula. Instead, it **iteratively** updates values to find the best solution.

- A **non-closed-form solution** is when you can't find an exact formula to solve the problem. I

📌 **Example in Machine Learning:**

- **Gradient Descent** for **Linear Regression** and **Deep Learning**
- It starts with random values and updates them step by step:

$$\theta = \theta - \alpha \cdot \nabla J(\theta)$$

Here, α (alpha) is the learning rate, and $\nabla J(\theta)$ is the gradient (slope).

◆ Advantages of Non-Closed-Form Solutions:

- ✓ **Works for large datasets** – No need for matrix inversion.
- ✓ **Can be applied to complex models** like deep learning.

◆ Disadvantages of Non-Closed-Form Solutions:

- ✗ **Slower** – Needs multiple iterations.
- ✗ **May not converge** – Bad choice of learning rate can make it unstable.

🚀 Key Differences in a Table

Feature	Closed-Form Solution	Non-Closed-Form (Iterative)
Definition	Direct formula gives exact answer	Step-by-step approximation
Example	Normal Equation in Linear Regression	Gradient Descent
Speed	Fast for small datasets	Slower but scalable
Accuracy	Exact solution	Approximate solution
Computational Cost	Expensive for large datasets	Efficient for large datasets
Use Cases	Small datasets with simple equations	Large datasets, deep learning

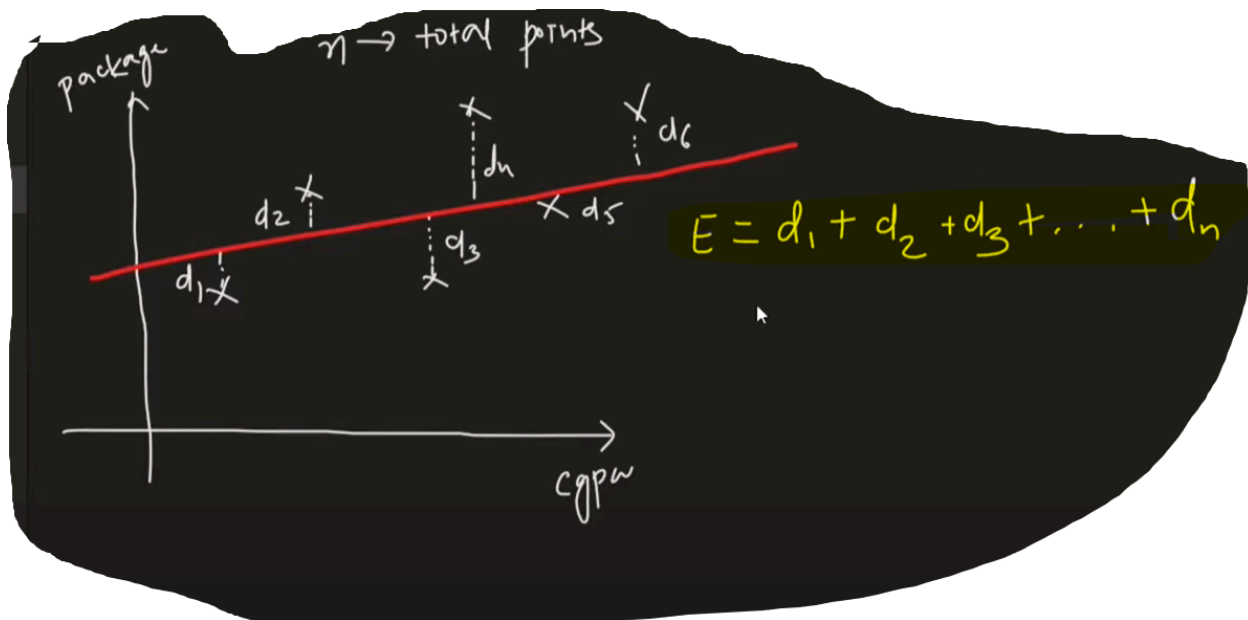
📌 Which One to Use?

- ✓ **Use Closed-Form if:**
 - The dataset is **small**.
 - The equation is **simple** (like Linear Regression).
 - You need **exact results** quickly.

✓ Use Non-Closed-Form if:

- The dataset is **large**.
- The model is **complex** (like deep learning).
- You can **afford iterative approximations**.

How to calculate m & b?



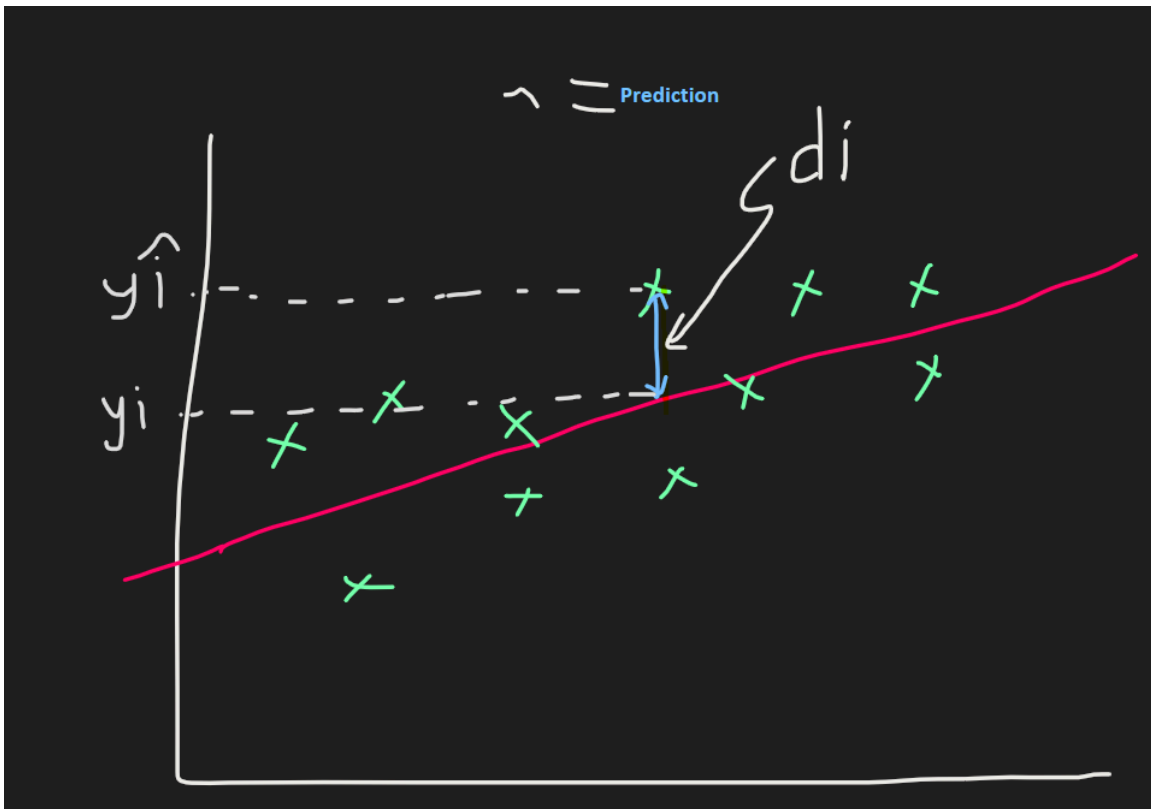
- Square the distances t:
 - Convert into +ve values
 - Penalize outliers
 - Apply differentiation in future

$$E = d_1^2 + d_2^2 + d_3^2 + \dots + d_n^2$$

Error function:

$$E = \sum_{i=1}^n d_i^2$$

← Error function (J)



$$E = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

y_i = Actual Package

\hat{y}_i = Package according to our model

Diff between 👉 = Error

Sum of the diff = Total error



Our aim is → To minimize E (Total Error)

- Avg error → $\times 1/n$

$$\hat{y}_i = m x_i + b$$
$$E(m, b) = \sum_{i=1}^n (y_i - m x_i - b)^2$$

How to find b?



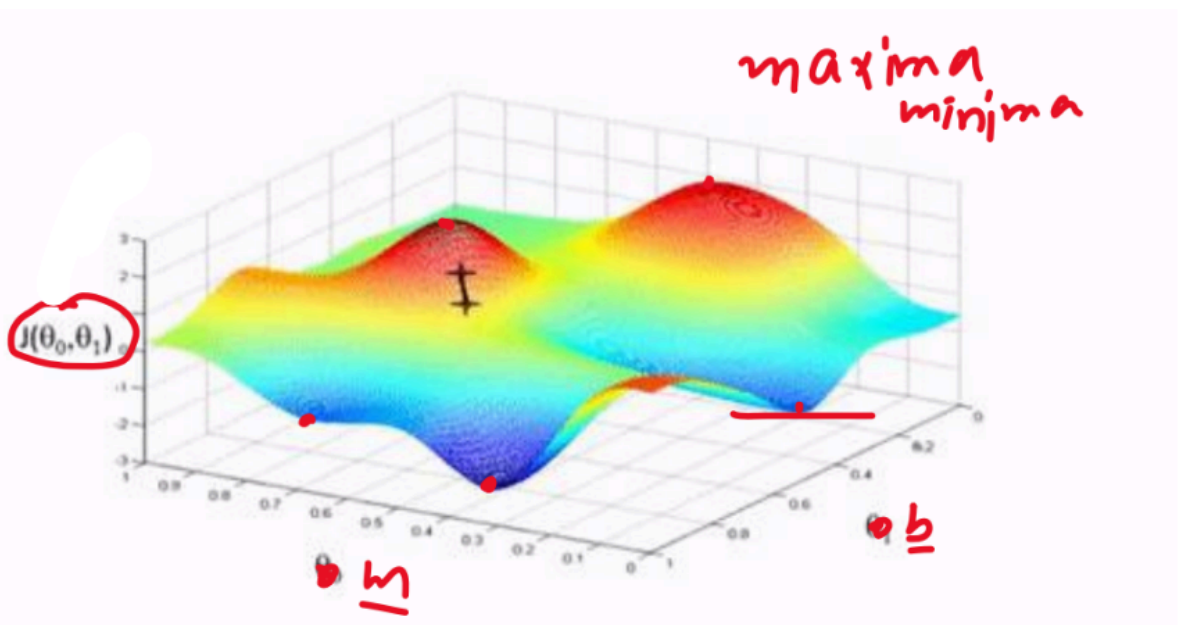
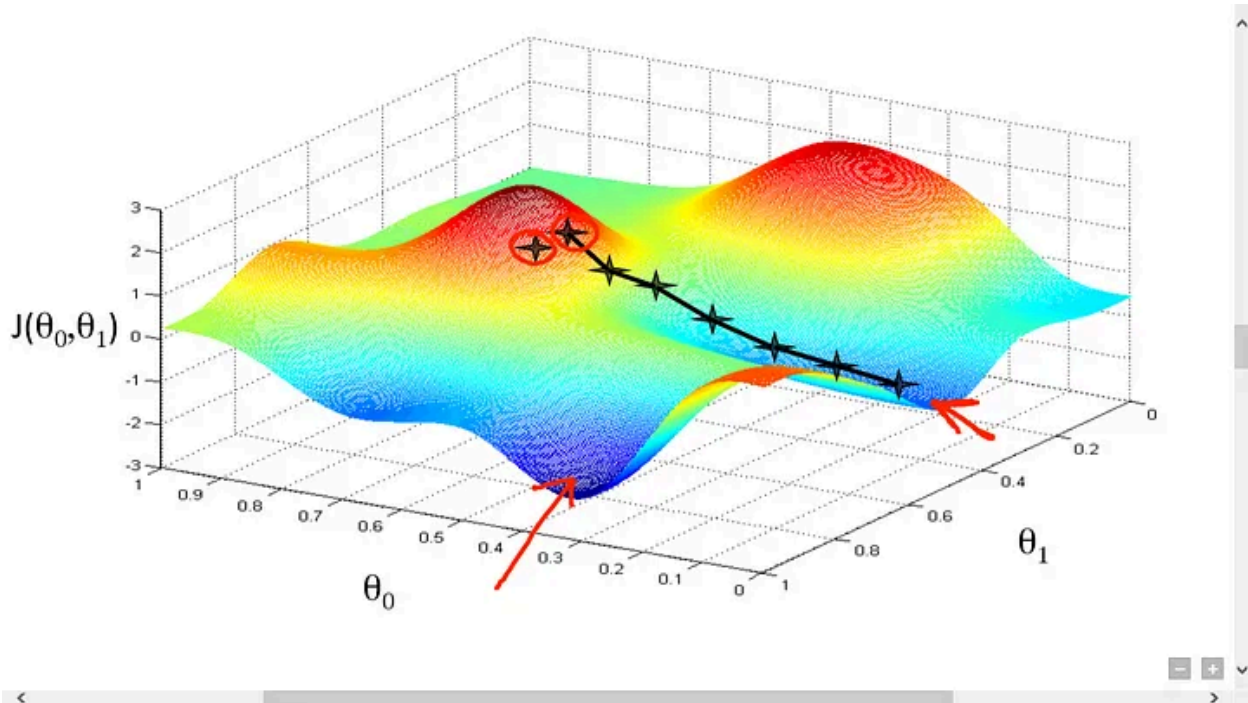
$$\hat{y}_i = m x_i + b$$
$$E(m, b) = \sum_{i=1}^n (y_i - m x_i - b)^2$$

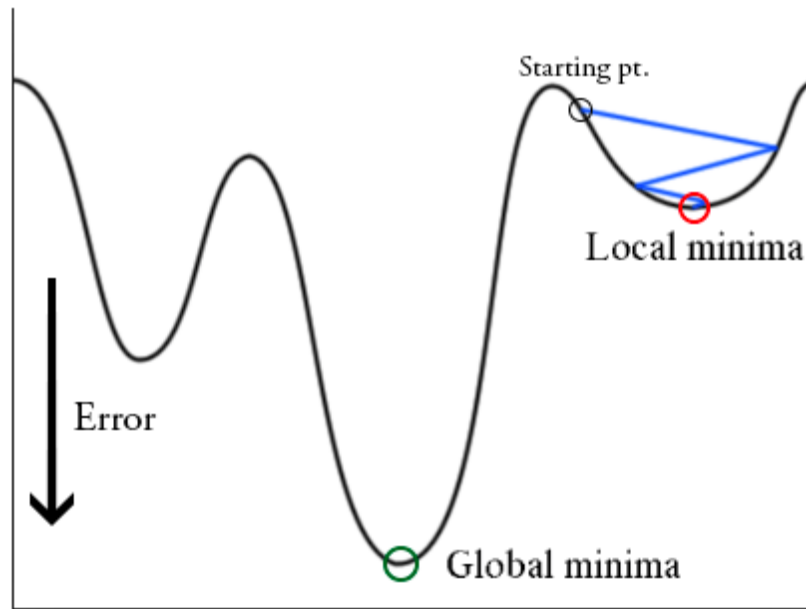


You can increase or decrease the error by manipulation the values of m & b

AIM → To find optimum values of m & b so that E would be minimum.

Gradient Descent





- At **global minima** → error is minimum or zero

How do we find the Global Minima?

- At minima → Slope = 0
- And to find out slope, we have to find out derivative and do =0
- So, we will find out derivative of the error function, both wrt m & b

Error Function:

$$E(m, b) = \sum_{i=1}^n (y_i - mx_i - b)^2$$

$$E(m) = \sum_{i=1}^n (y_i - mx_i)^2$$

$$E(b) = \sum_{i=1}^n (y_i - x_i - b)^2$$

- We have to use partial derivatives because we have m & b

$$\frac{\partial E}{\partial m} = 0, \quad \frac{\partial E}{\partial b} = 0$$

- We will get 2 equations
- & with the help of these 2 equations, we'll calculate the value of m & b
- If it were only 1:

$$\frac{dE}{dx} = 0$$

First solve this:

$$\frac{\partial E}{\partial b} = \frac{\partial}{\partial b} \sum_{i=1}^n (y_i - mx_i - b)^2 = 0$$

Then solve:

$$\frac{\partial E}{\partial m} = \sum \frac{\partial}{\partial m} (y_i - mx_i - \bar{y} + m\bar{x})^2 = 0$$

After all the complex calculations, we get the value of **m**:

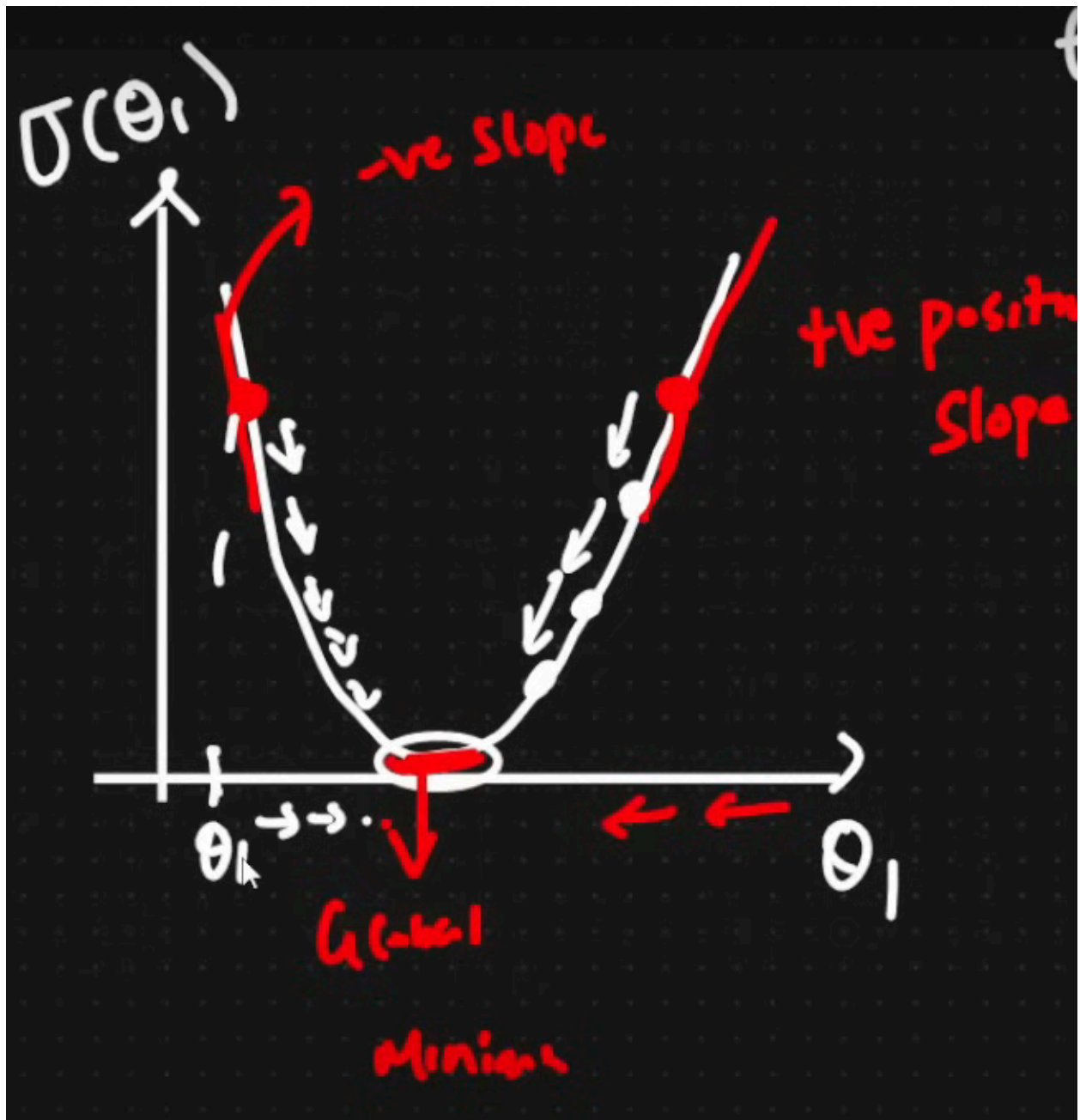
$$m = \frac{\sum_{i=1}^n (x_i - \bar{x})(y_i - \bar{y})}{\sum_{i=1}^n (x_i - \bar{x})^2}$$

Repeat convergence theorem

- The **Convergence Theorem** is a mathematical guarantee that an iterative process (like gradient descent) will eventually reach a solution (or get very close to it).
- In machine learning, it's often used to ensure that optimization algorithms will find the best model parameters.

Key Idea:

- If you keep improving your model step by step (iteration by iteration), you'll eventually reach the **optimal solution** (or a good approximation of it).



Learning rate \rightarrow Should be small so that the increment will be small.

Outline

- ① Start with θ_0 & θ_1
- ② Keep changing θ_0, θ_1 to reduce $J(\theta_0, \theta_1)$ until we reach near global Minima.
- ③ Convergence Theorem
}

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} (J(\theta_0, \theta_1)) \quad j=0 \text{ and } 1$$

}

Notations of Andrew Ng:

COST FUNCTION/SQUARED ERROR FUNCTION 📌

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

Parameters: θ_0, θ_1

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

Goal: minimize $J(\theta_0, \theta_1)$

Predicted Point $h_{\theta}(x)$

Actual value $y^{(i)}$

$J \rightarrow$ Error function