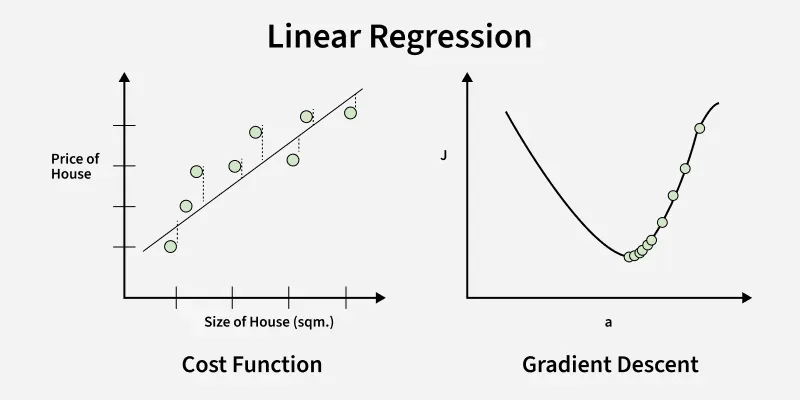
**Gradient Descent in Linear Regression:**

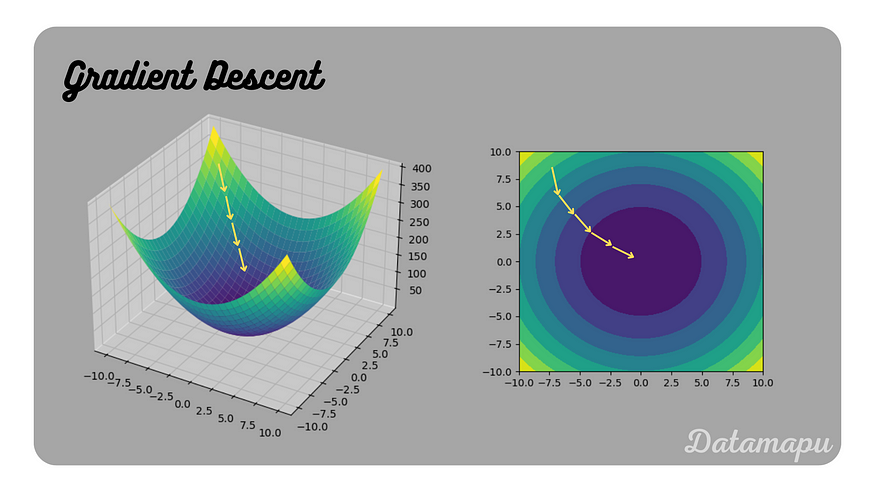
[**https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/**](https://www.geeksforgeeks.org/gradient-descent-in-linear-regression/)

**Gradient descent** is a optimization algorithm used in **linear regression** to find the best fit line to the data. It works by gradually by adjusting the line’s slope and intercept to reduce the difference between actual and predicted values. This process helps the model make accurate predictions by minimizing errors step by step. In this article we will see more about Gradient Descent and its core concepts in detail.



Gradient Descent in Linear Regression

Above image shows two graphs, left one plots house prices against size to show errors measured by the **cost function** while right one shows how **gradient descent** moves downhill on the cost curve to minimize error by updating parameters step by step.



**What is Gradient Descent?**

Gradient descent is an optimization algorithm used in **machine learning** to minimize the cost function by iteratively adjusting parameters in the direction of the negative gradient, aiming to find the optimal set of parameters.

The cost function represents the discrepancy between the predicted output of the model and the actual output. Gradient descent aims to find the parameters that minimize this discrepancy and improve the model’s performance.

The algorithm operates by calculating the gradient of the cost function, which indicates the direction and magnitude of the steepest ascent. However, since the objective is to minimize the cost function, gradient descent moves in the opposite direction of the gradient, known as the negative gradient direction.

By iteratively updating the model’s parameters in the negative gradient direction, gradient descent gradually converges towards the optimal set of parameters that yields the lowest cost. The learning rate, a hyperparameter, determines the step size taken in each iteration, influencing the speed and stability of convergence.

Gradient descent can be applied to various machine learning algorithms, including **linear regression**, **logistic regression**, **neural networks**, and **support vector machines**. It provides a general framework for optimizing models by iteratively refining their parameters based on the cost function.

**Example of Gradient Descent Algorithm**

Let’s say you are playing a game in which the players are at the top of a mountain and asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?

Take a moment to think about this before you read on.

The best way is to observe the ground and find where the land descends. From that position, step in the descending direction and iterate this process until we reach the lowest point.

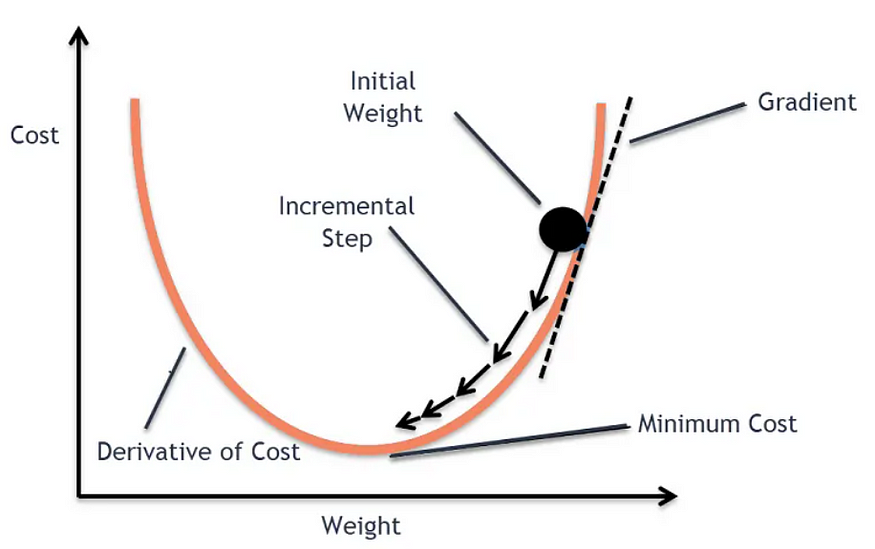
Finding the lowest point in a hilly landscape.



Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

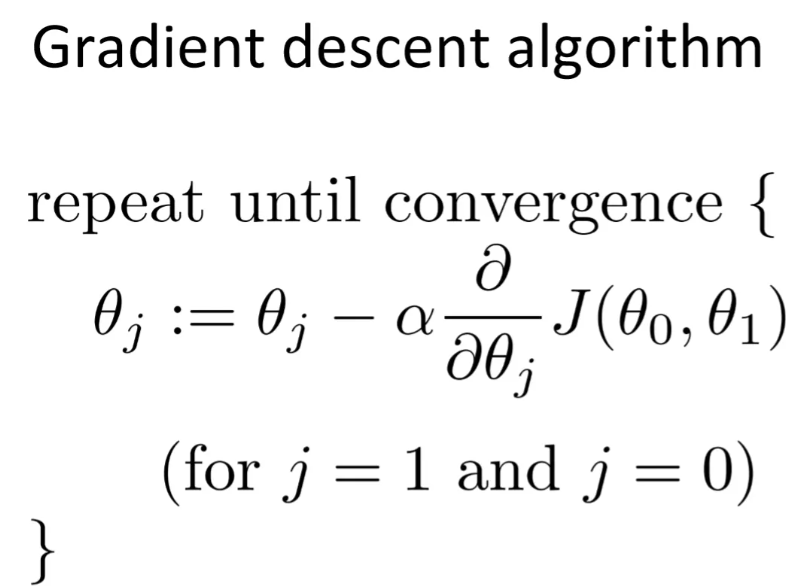
To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called**Gradient Ascent.**

Gradient descent was originally proposed by**CAUCHY**in 1847. It is also known as the steepest descent.



The goal of the gradient descent algorithm is to minimize the given function (say, cost function). To achieve this goal, it performs two steps iteratively:

1. **Compute the gradient**(slope), the first-order derivative of the function at that point
2. **Make a step (move) in the direction opposite to the gradient**. The opposite direction of the slope increases from the current point by alpha times the gradient at that point



Alpha is called**Learning rate**— a tuning parameter in the optimization process. It decides the length of the steps.

**How Does Gradient Descent Work?**

1. The algorithm optimizes to minimize the model’s cost function.
2. The cost function measures how well the model fits the training data and defines the difference between the predicted and actual values.
3. The cost function’s gradient is the derivative with respect to the model’s parameters and points in the direction of the steepest ascent.
4. The algorithm starts with an initial set of parameters and updates them in small steps to minimize the cost function.
5. In each iteration of the algorithm, it computes the gradient of the cost function with respect to each parameter.
6. The gradient tells us the direction of the steepest ascent, and by moving in the opposite direction, we can find the direction of the steepest descent.
7. The learning rate controls the step size, which determines how quickly the algorithm moves towards the minimum.
8. The process is repeated until the cost function converges to a minimum. Therefore indicating that the model has reached the optimal set of parameters.
9. Different variations of gradient descent include batch gradient descent, stochastic gradient descent, and mini-batch gradient descent, each with advantages and limitations.
10. Efficient implementation of gradient descent is essential for performing well in machine learning tasks. The choice of the learning rate and the number of iterations can significantly impact the algorithm’s performance.

**Challenges of Gradient Descent Algorithm**

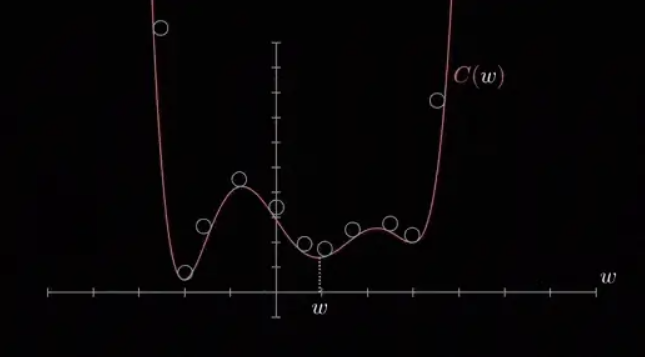
While gradient descent is a powerful optimization algorithm, it can also present some challenges affecting its performance. Some of these challenges include:

1. Local Optima: Gradient descent can converge to local optima instead of the global optimum, especially if the cost function has multiple peaks and valleys.
2. Learning Rate Selection: The choice of learning rate can significantly impact the performance of gradient descent. If the learning rate is too high, the algorithm may overshoot the minimum, and if it is too low, the algorithm may take too long to converge.
3. Overfitting: Gradient descent can overfit the training data if the model is too complex or the learning rate is too high. This can lead to poor generalization performance on new data.
4. Convergence Rate: The convergence rate of gradient descent can be slow for large datasets or high-dimensional spaces, making the algorithm computationally expensive.
5. Saddle Points: In high-dimensional spaces, saddle points can cause the gradient of the cost function to get stuck in a plateau, preventing gradient descent from converging to a minimum.

Researchers have developed several variations of gradient descent algorithms to overcome these challenges, such as adaptive learning rate, momentum-based, and second-order methods. Additionally, choosing the right regularization method, model architecture, and hyperparameters can also help improve the performance of the gradient descent algorithm.

**Local Minima**

The cost function may consist of many minimum points. Depending on the initial point (i.e., initial parameters(theta)) and the learning rate, the gradient may settle on any minima. Therefore, the optimization may converge to different starting points and learning rates.



In machine learning, local minima and global minima are two important concepts related to the optimization of loss functions. A loss function is a function that measures the error between a model’s predictions and the ground truth. The goal of machine learning is to find a model that minimizes the loss function.

A local minimum is a point in the parameter space where the loss function is minimized in a local neighborhood. A global minimum is a point in the parameter space where the loss function is minimized globally.



In general, it is not possible to find the global minimum of a loss function analytically. Instead, machine learning algorithms use iterative optimization methods to find a local minimum.

One common iterative optimization method is gradient descent. Gradient descent starts at a random point in the parameter space and then iteratively updates the parameters in the direction of the negative gradient of the loss function. The negative gradient points in the direction of the steepest descent, so gradient descent will eventually converge to a local minimum.

However, there is no guarantee that the local minimum found by gradient descent is the global minimum. In fact, it is possible that gradient descent will get stuck in a local minimum that is not the global minimum.

There are a number of techniques that can be used to avoid getting stuck in local minima. One technique is to use a different optimization algorithm, such as simulated annealing or genetic algorithms. Another technique is to use a regularization term in the loss function. Regularization terms penalize the model for having large parameters, which can help to prevent the model from overfitting the training data and getting stuck in a local minimum.

**Why Use Gradient Descent for Linear Regression?**

[Linear regression](https://www.geeksforgeeks.org/ml-linear-regression/) finds the **best-fit line** for a dataset by minimizing the **error** between the actual and predicted values. This error is measured using the [cost function](https://www.geeksforgeeks.org/what-is-the-cost-function-in-linear-regression/) usually Mean Squared Error (MSE). The goal is to find the model parameters i.e. the **slope m** and the **intercept b** that minimize this cost function.

For simple linear regression, we can use formulas like [Normal Equation](https://www.geeksforgeeks.org/ml-normal-equation-in-linear-regression/) to find parameters directly. However for **large datasets** or **high-dimensional data** these methods become computationally expensive due to:

* Large matrix computations.
* Memory limitations.

In models like [polynomial regression](https://www.geeksforgeeks.org/python-implementation-of-polynomial-regression/), the cost function becomes highly complex and non-linear, so analytical solutions are not available. That’s where **gradient descent** plays an important role even for:

* Large datasets.
* Complex, high-dimensional problems.
* Gradient decent perform much better if data are in same scale

**scikit-learn (sklearn)** does support **Gradient Descent**, but not directly in the LinearRegression class. Instead:

**✅ scikit-learn's LinearRegression:**

* Uses the **Ordinary Least Squares (OLS)** analytical solution.
* It **does not use Gradient Descent**.

**✅ For Gradient Descent in sklearn:**

You can use:

* sklearn.linear\_model.SGDRegressor

This class uses **Stochastic Gradient Descent (SGD)** for regression.

**✅ Code Example: Linear Regression Using Gradient Descent (SGDRegressor)**

python

CopyEdit

from sklearn.linear\_model import SGDRegressor

from sklearn.model\_selection import train\_test\_split

from sklearn.metrics import mean\_squared\_error, r2\_score

import numpy as np

import matplotlib.pyplot as plt

# 1. Sample dataset

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

y = np.array([2, 4, 5, 4, 5])

# 2. Train-test split

X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)

# 3. Define SGD Regressor

model = SGDRegressor(max\_iter=1000, learning\_rate='invscaling', eta0=0.01)

# 4. Fit model

model.fit(X\_train, y\_train)

# 5. Predict

y\_pred = model.predict(X\_test)

# 6. Evaluation

mse = mean\_squared\_error(y\_test, y\_pred)

r2 = r2\_score(y\_test, y\_pred)

print(f"Model Coefficients (theta1): {model.coef\_[0]:.4f}")

print(f"Model Intercept (theta0): {model.intercept\_[0]:.4f}")

print(f"Mean Squared Error: {mse:.4f}")

print(f"R² Score: {r2:.4f}")

# 7. Plot

plt.scatter(X, y, color='blue', label='Original Data')

plt.plot(X, model.predict(X), color='red', label='Fitted Line (SGD)')

plt.xlabel("X")

plt.ylabel("y")

plt.title("Linear Regression with Gradient Descent (SGD)")

plt.legend()

plt.grid(True)

plt.show()

**🔍 Notes:**

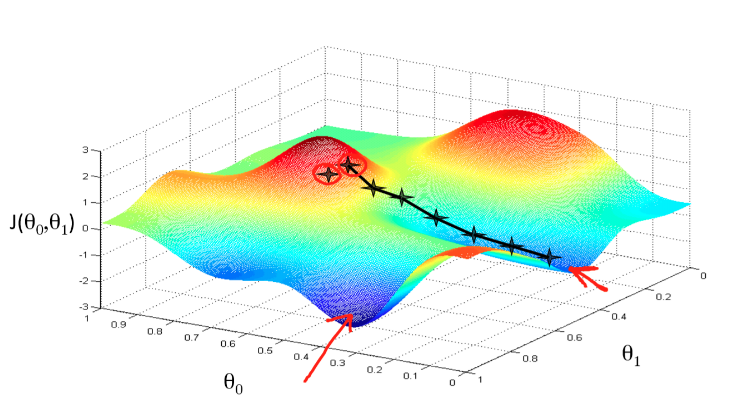
* SGDRegressor uses **stochastic gradient descent**, which is suitable for large datasets.
* eta0 is the **initial learning rate**.
* learning\_rate='invscaling' changes the learning rate over time.
* You can use loss='squared\_error' (default) for standard linear regression.

**Internal working explanation**

[**https://medium.com/geekculture/mathematics-behind-gradient-descent-f2a49a0b714f**](https://medium.com/geekculture/mathematics-behind-gradient-descent-f2a49a0b714f)

[**https://rumn.medium.com/understanding-optimization-in-ml-with-gradient-descent-implement-sgd-regressor-from-scratch-4e11dac74c9**](https://rumn.medium.com/understanding-optimization-in-ml-with-gradient-descent-implement-sgd-regressor-from-scratch-4e11dac74c9)

The gradient gives the direction of the maximum change and the magnitude indicates the maximum rate of change. The gradient always points in the direction of the steepest increase in the objective function.



J(θ0, θ1) is the cost function, and θ0 and θ1 are the variable or parameters. The idea here is to find the optimal value for these parameters to minimise the cost function is called gradient descent algorithm. Source: [[2]](https://hackernoon.com/gradient-descent-aynk-7cbe95a778da)

If we update variables or parameters of some cost function in the direction of the negative gradient in an iterative manner to reach the minimum of some cost function is called gradient descent algorithm.

Let’s explain this concept in the context of linear regression.

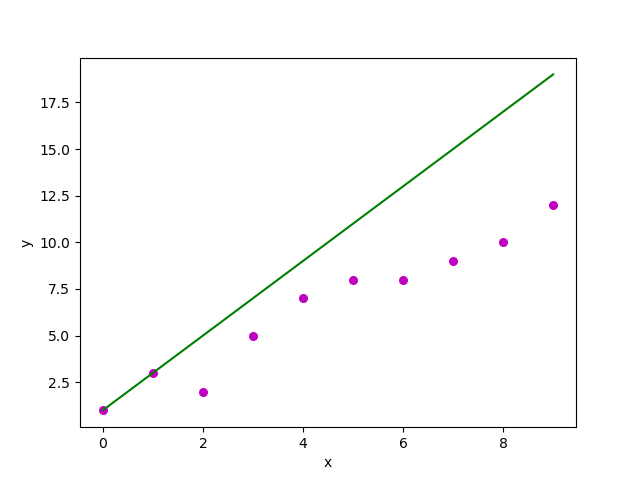
Overall, we need to train the regression model with historical data so that it can predict Y with high accuracy. Linear regression works by finding the coefficients of a line that best fit the historical data to predict y. We can represent a line with the equation Y= mX +b, where m and b are the coefficients or variables of the function. Our task is to use gradient descent approach to optimize the coefficients of the regression model such a way that it has less prediction error.

**Steps for the gradient descent**

*The below pseudo-code is a modified version from the source:*[*[4]*](https://www.kdnuggets.com/2017/04/simple-understand-gradient-descent-algorithm.html)

**1. Initialise the coefficients m and b with random values**

For example m = 1 and b =2, i.e a line equation is y = mx+b, then y = 1\*x+2. This can give a line that is not best fitted for the historical data, as shown below.

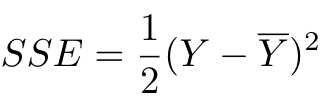


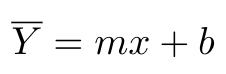
The line is not best fitted, and the idea here is to find the optimal values of the line’s coefficients that can best fit the historical data. Once we find the best line that fit the data, we can use this regression model for prediction.

**2. Compute the gradient.**

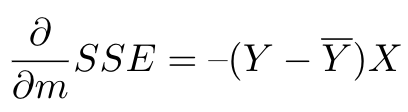
We use the Sum of Squared Errors (SSE) as our loss/ cost function to minimise the prediction error. In this case, the gradient of SSE is a partial derivative of SSE w.r.t m and partial derivative of SSE w.r.t b.

SSE equation is as follows:

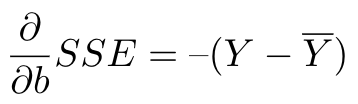




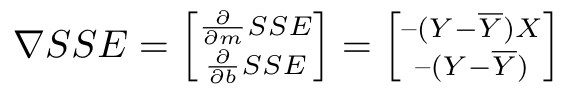
The partial derivative of SSE w.r.t m is:



The partial derivative of SSE w.r.t b is:



Finally, the gradient is made up of all the partial derivatives i.e. both partial derivatives of SSE as shown below:



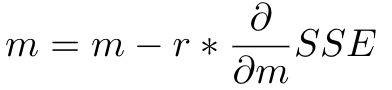
As we discussed, gradients give the direction of the movement of m and b w.r.t to SSE.

In the first step, we initialize m and b with random values. For the subsequent iterative process, m and b values are updated using step 3. In this step, we compute the partial derivative of SSE w.r.t m and partial derivative w.r.t b using the above equation for each data point in X. Finally, calculate the sum of all partial derivatives f w.r.t m and all partial derivatives f w.r.t b. In other words, we compute the gradient of SSE for the data X.

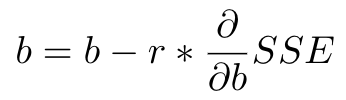
**3. Update coefficients in the direction of optimal m and b**

We can update the coefficients m and b using the gradient calculated from the above equations.

Update rule of coefficient m is:



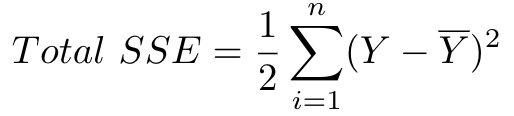
Update rule of coefficient b is:



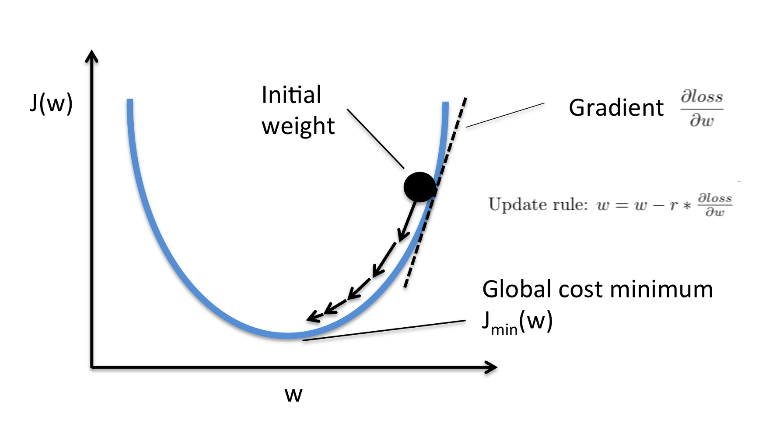
Here r is the learning rate.

**4. Use new m and b for prediction**

We use the data X with new m and b, computed in the above step, to draw the line that fit the data. We calculate the SSE of each data point in X to find out the total SSE, which is a sum of squared errors of data points, divided of 2. The total SSE indicates the error rate of the model prediction.

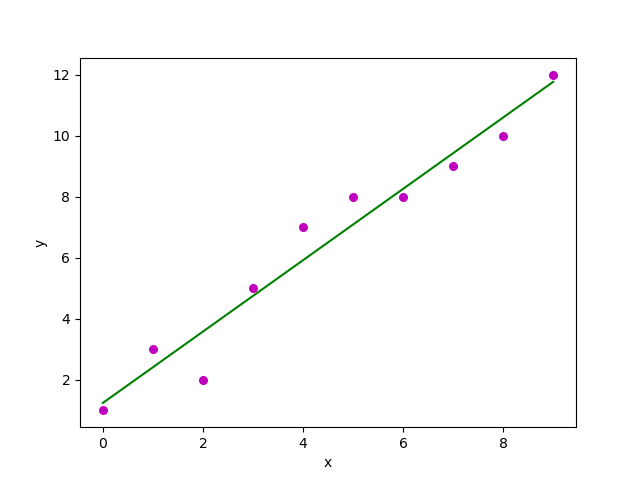


**5. Repeat steps 2, 3 and 4**



This diagram shows the overall steps of the gradient descent algorithm. Here, computing the gradient of loss function at a specific point in the graph and then update the weight w(that is the variable of loss function) with the gradient to reach the loss minimum value. This process repeats until reach the minimum. Source:[[5]](https://mc.ai/gradient-descent-and-its-types/)

We need to repeat the steps 2, 3 and 4 until optimal values for the coefficients m and b are found that reduces the SSE to a minimum value. The optimal values of m and b enable the model to predict the Y with the highest accuracy.



We found the best line that fit the data for our regression model using the gradient descent algorithm.

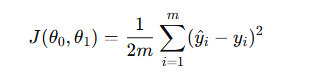
This is the overall intuitive explanation of the gradient descent algorithm. When you think about gradient descent algorithm for a neural network, the whole approach we explained here is the basis

In **gradient descent for linear regression**, the formulas for updating the model parameters θ0\theta\_0θ0​ (intercept) and θ1\theta\_1θ1​ (slope) are derived from **minimizing the loss function** — specifically, the **Mean Squared Error (MSE)**.

Let’s break this down step by step:

**🔶 1. Loss Function Used**

The **loss function** (also called cost function) used in linear regression with gradient descent is:



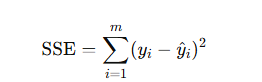
Where:

* y^​i​=θ0 ​+ θ1​xi​
* yi​ is the actual value.
* m is the number of training samples.
* This is the **Mean Squared Error (MSE)** cost function (scaled by (1/2)​ for convenience in derivatives).

**🔶 2. Gradient Descent — Why Not SSE or SSR Directly?**

**✅ SSE:**

SSE is the **Sum of Squared Errors**:

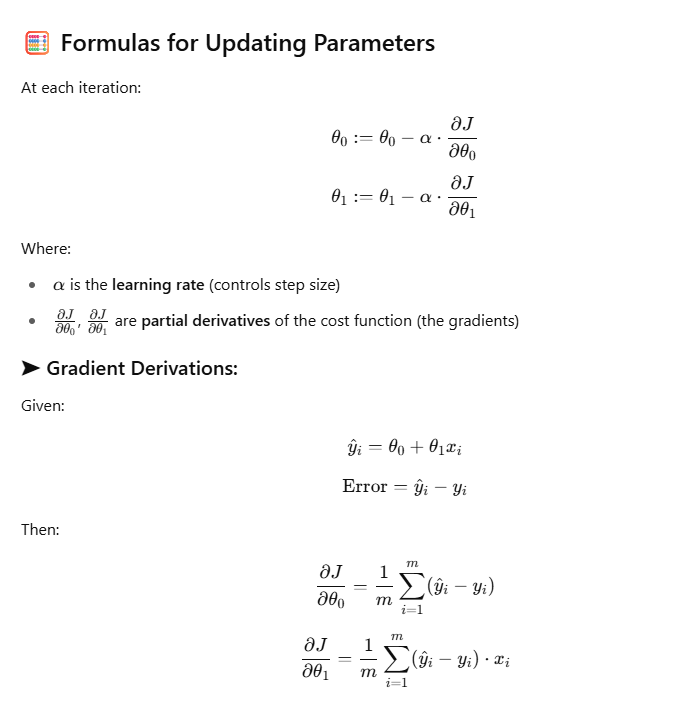


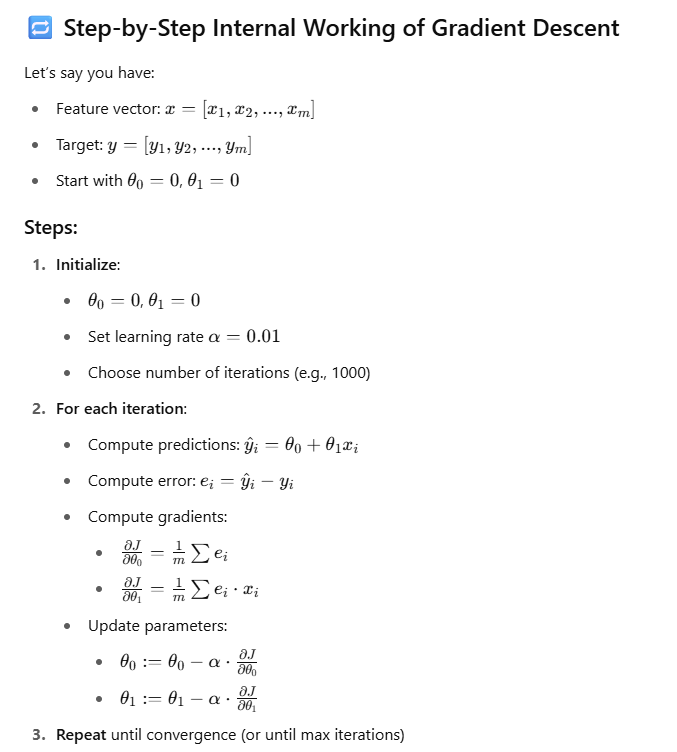
It is the **total error** we're trying to minimize.

✅ **The formula used in gradient descent is based on the derivative of SSE** (or MSE, which is SSE divided by mmm). So, SSE *is* being used — just indirectly, through **calculus**.

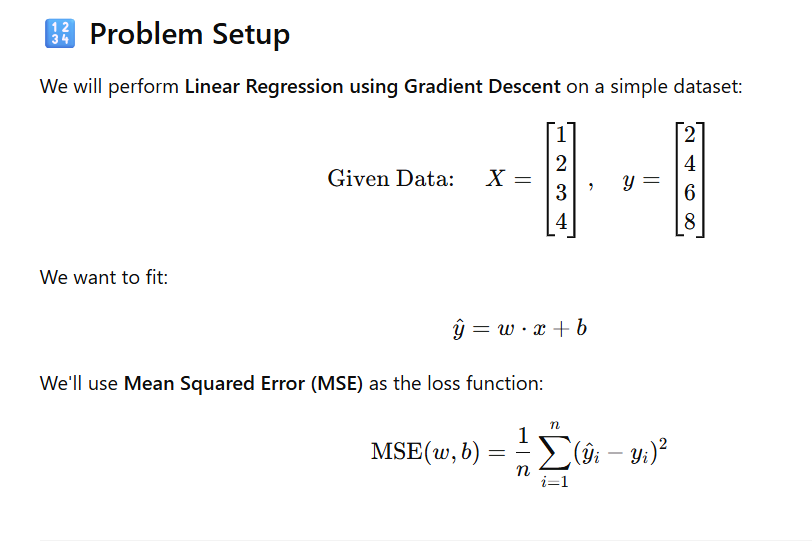
**❌ SSR:**

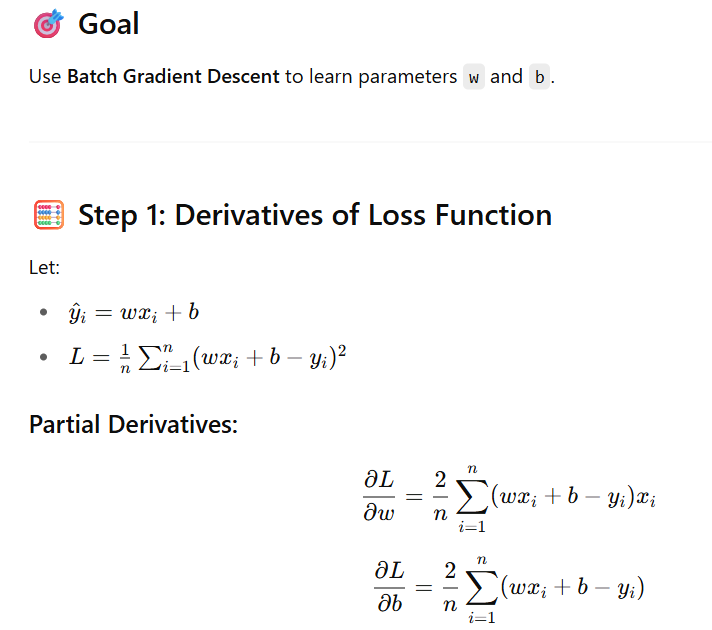
SSR (Sum of Squares due to Regression) measures the explained variance by the model, not the error. It's used more in **model evaluation (R² score)**, not in optimization.

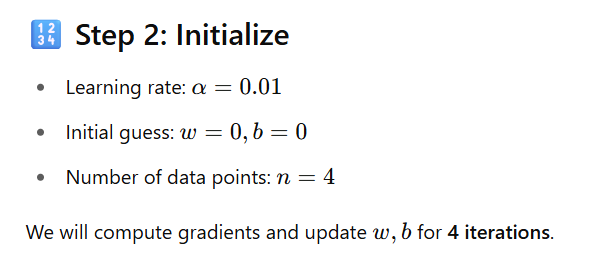


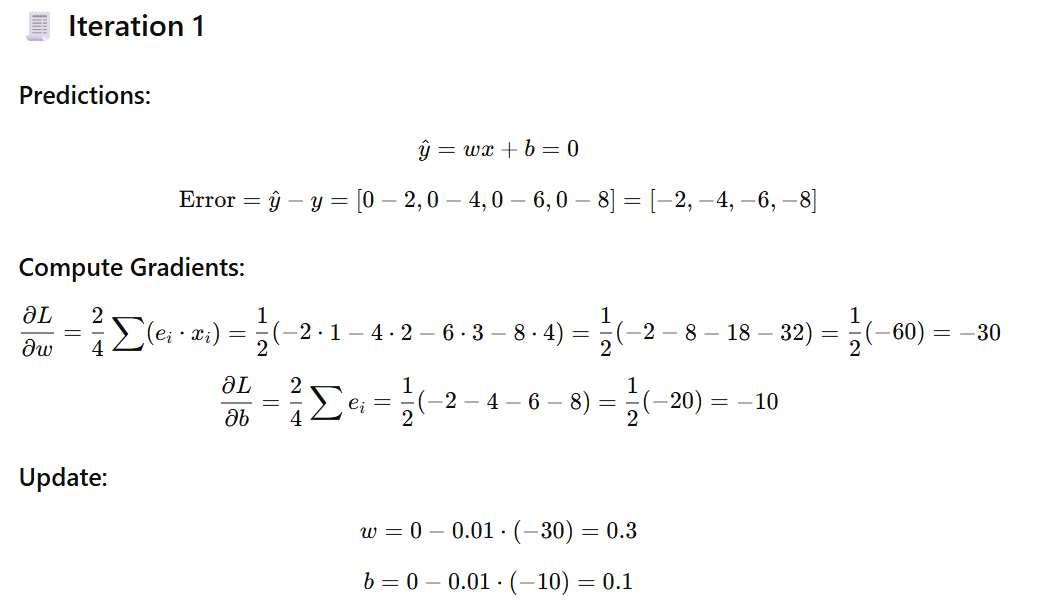


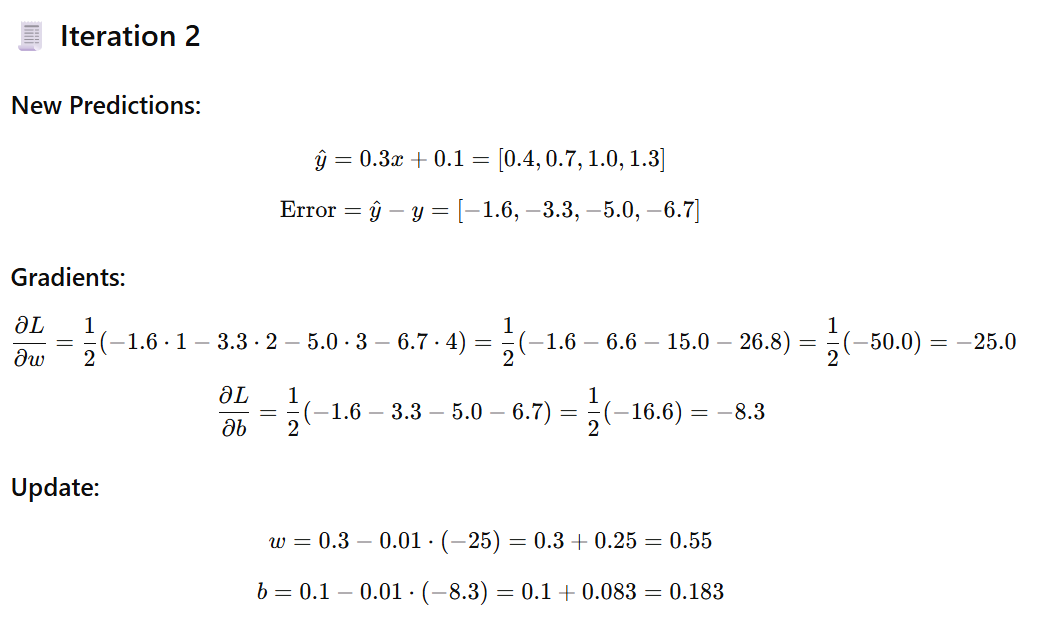
**Let’s Solve a new problem statement:**

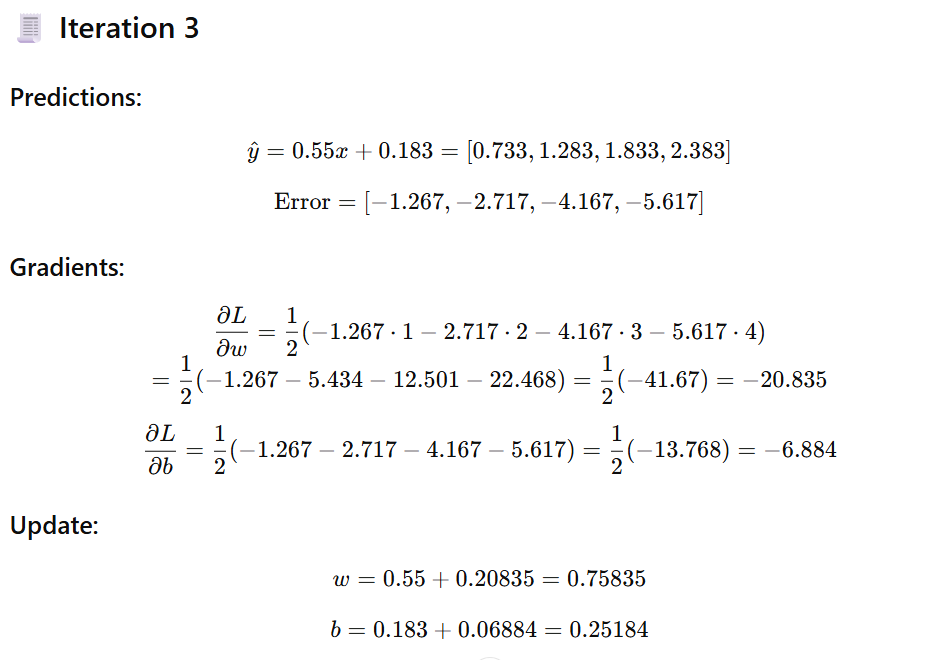


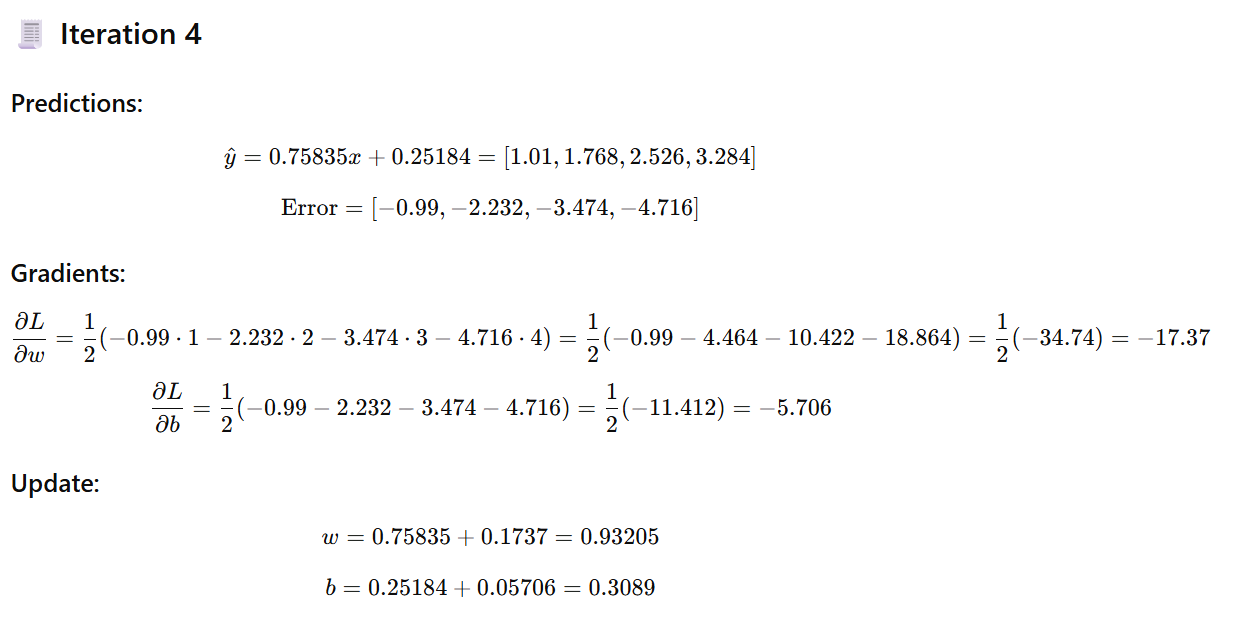


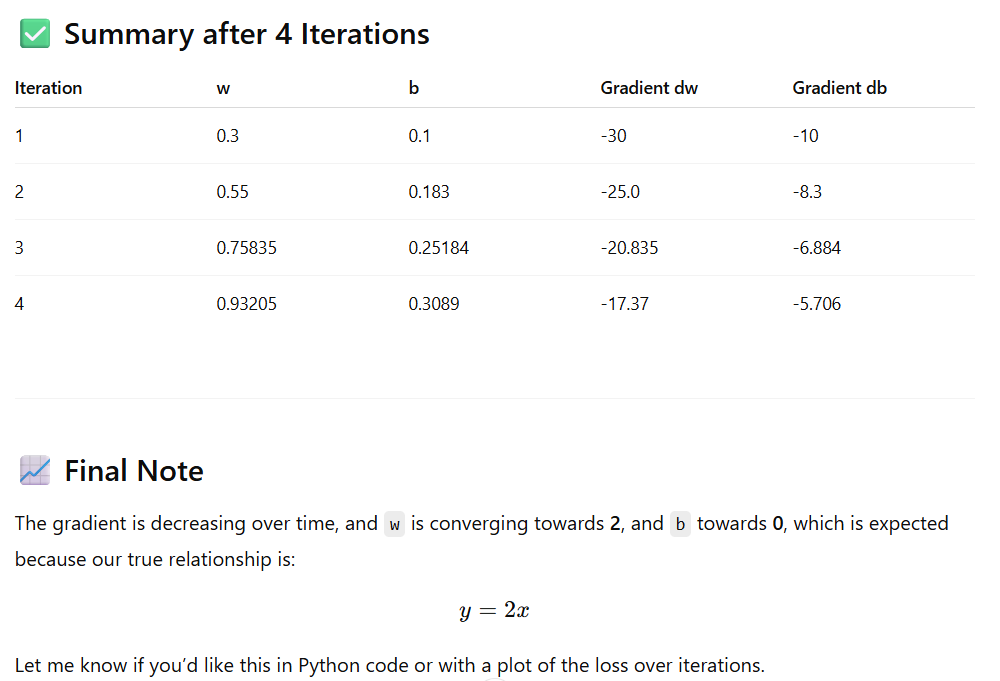












**Python code: -**

import matplotlib.pyplot as plt

import numpy as np

import matplotlib.animation as animation

# Sample dataset

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

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

m = len(x)

# Learning rate

alpha = 0.1

# Initialize theta

theta\_0 = 0

theta\_1 = 0

# Store theta history

theta\_history = [(theta\_0, theta\_1)]

# Perform 3 iterations of gradient descent

for \_ in range(3):

y\_pred = theta\_0 + theta\_1 \* x

error = y\_pred - y

d\_theta\_0 = (1 / m) \* np.sum(error)

d\_theta\_1 = (1 / m) \* np.sum(error \* x)

theta\_0 = theta\_0 - alpha \* d\_theta\_0

theta\_1 = theta\_1 - alpha \* d\_theta\_1

theta\_history.append((theta\_0, theta\_1))

# Prepare plot

fig, ax = plt.subplots()

line, = ax.plot([], [], 'r-', lw=2, label='Fitted Line')

scatter = ax.scatter(x, y, color='blue', label='Data Points')

title = ax.set\_title('')

ax.set\_xlim(0, 4)

ax.set\_ylim(0, 4)

ax.set\_xlabel("x")

ax.set\_ylabel("y")

ax.legend()

# Animation function

def animate(i):

theta\_0, theta\_1 = theta\_history[i]

y\_pred\_line = theta\_0 + theta\_1 \* x

line.set\_data(x, y\_pred\_line)

title.set\_text(f"Iteration {i}: y = {theta\_0:.2f} + {theta\_1:.2f}x")

return line, title

ani = animation.FuncAnimation(fig, animate, frames=len(theta\_history), interval=1000, blit=False, repeat=True)

plt.close(fig)

ani

**Final Notes**

* The result is a line y^​=θ0​+θ1​x that **best fits the data** (minimizes the MSE).
* Learning rate (α) is critical: too large → overshooting; too small → slow convergence.
* You can use **Batch**, **Stochastic**, or **Mini-batch Gradient Descent** depending on data size.

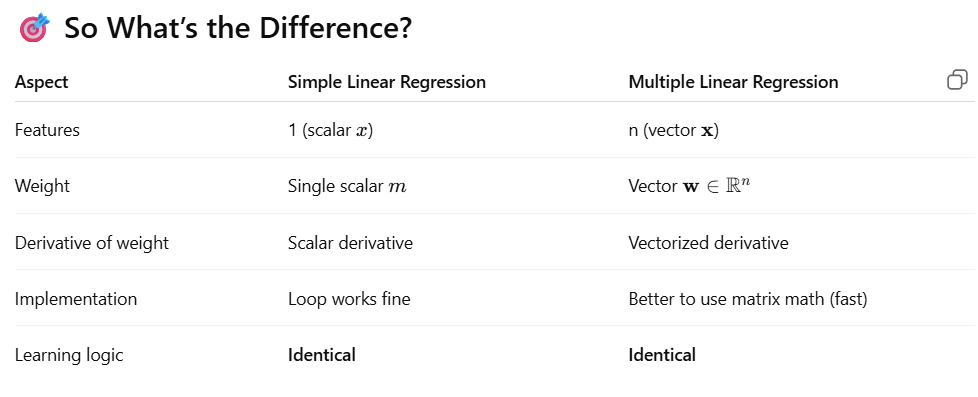
**Gradient Descent for multiple linear regression**

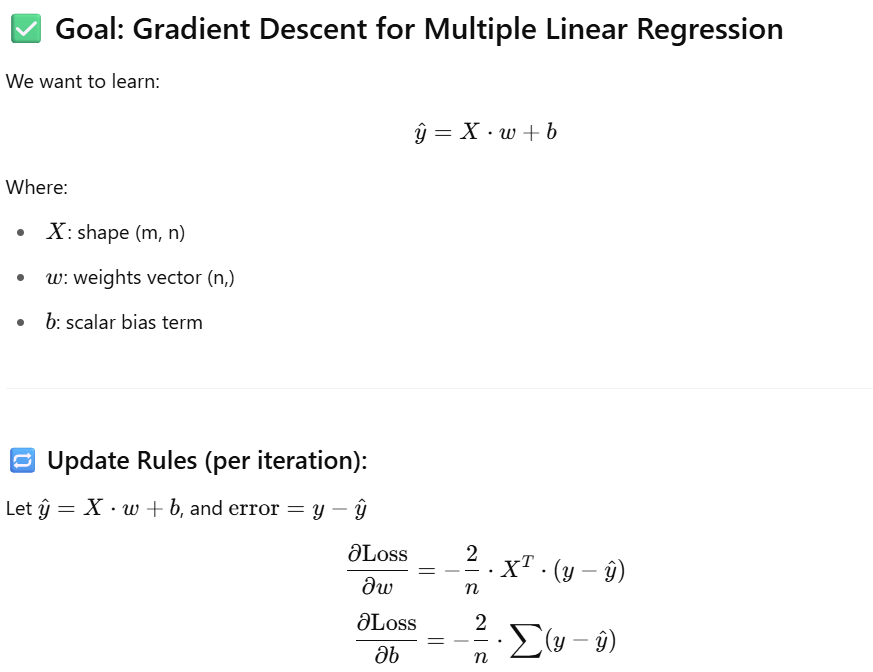
Gradient Descent works **the same way** for both:

* **Simple Linear Regression** (1 feature)
* **Multiple Linear Regression** (n features)

The **loss function** is the same, the **update rule** is the same — the only difference is:

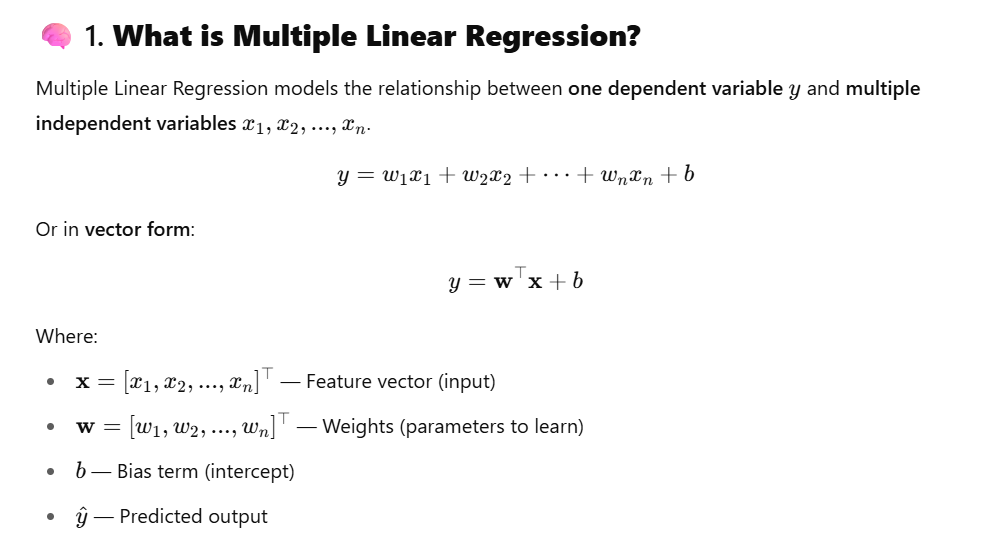
* In simple linear regression, you're updating **one weight** m
* In multiple linear regression, you're updating **a vector of weights** w
* There is another difference a bit in calculation approach

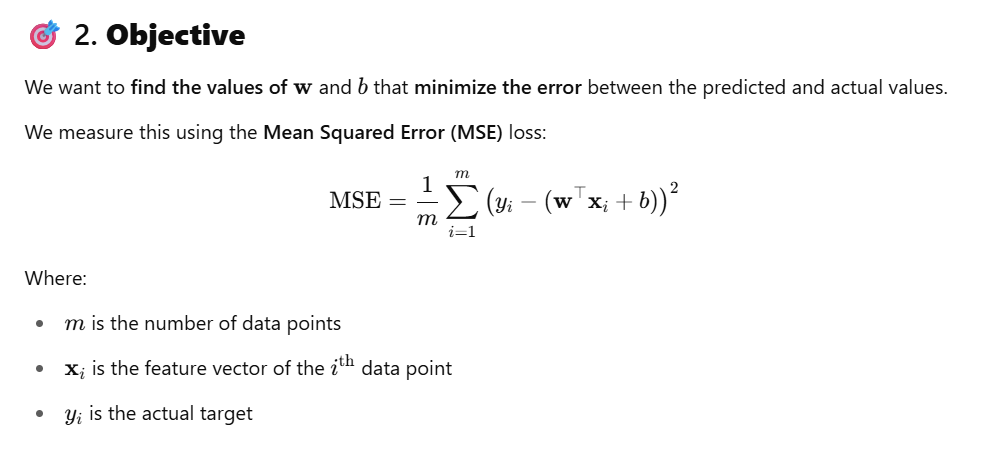


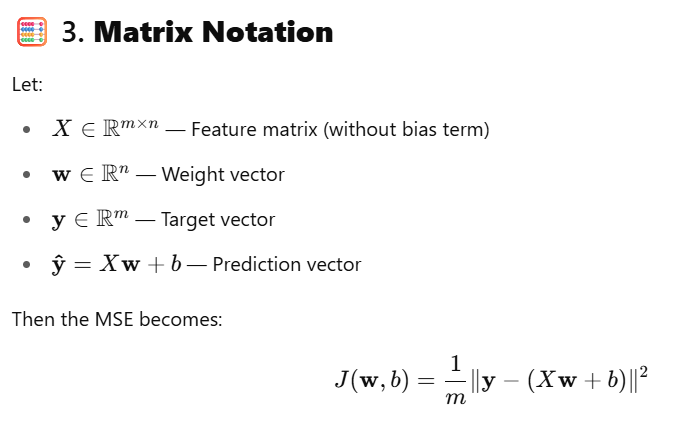


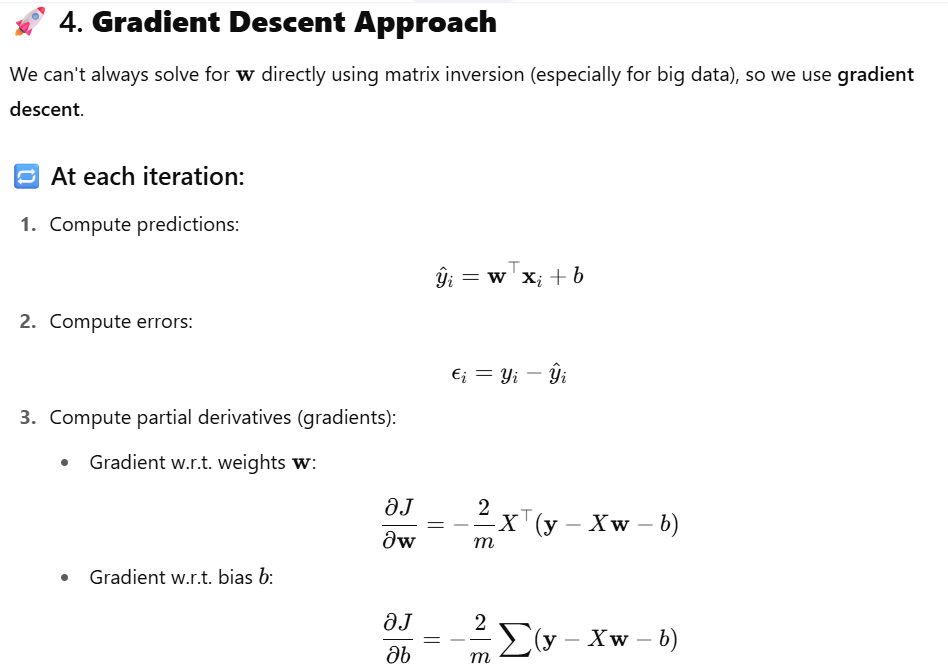
**Multiple Linear Regression — A Detailed Mathematical Intuition for Gradient Descent**

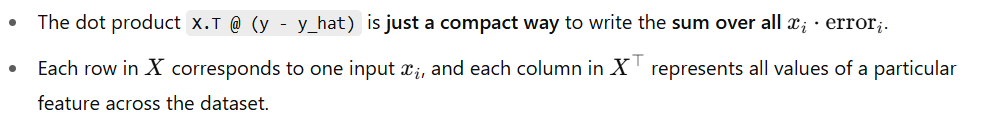
<https://medium.com/accredian/gradient-descent-multi-linear-regression-ml-from-scratch-3-8773421be2ce>



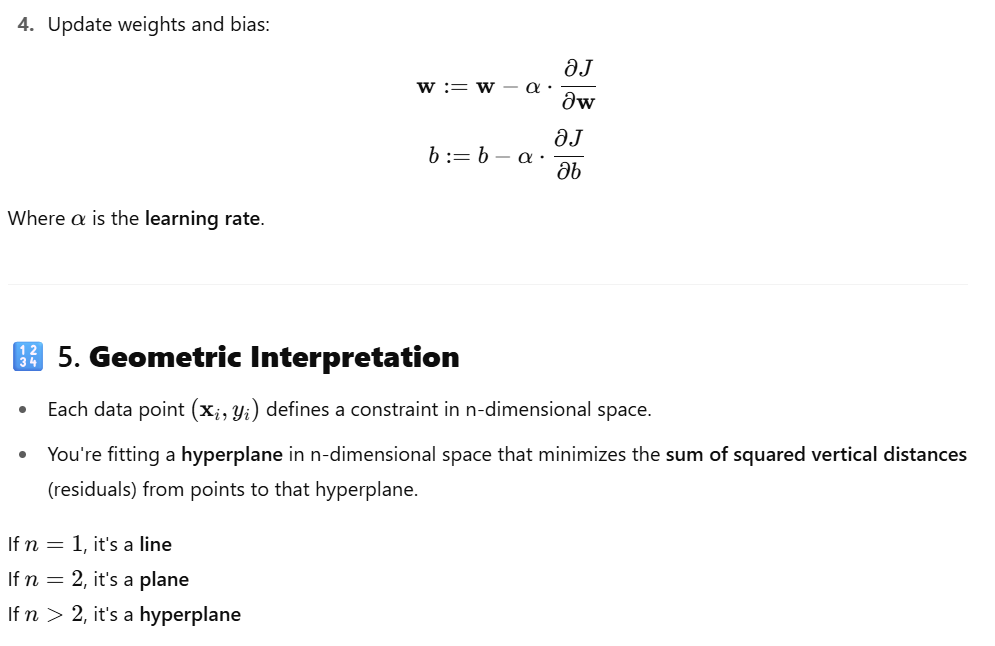


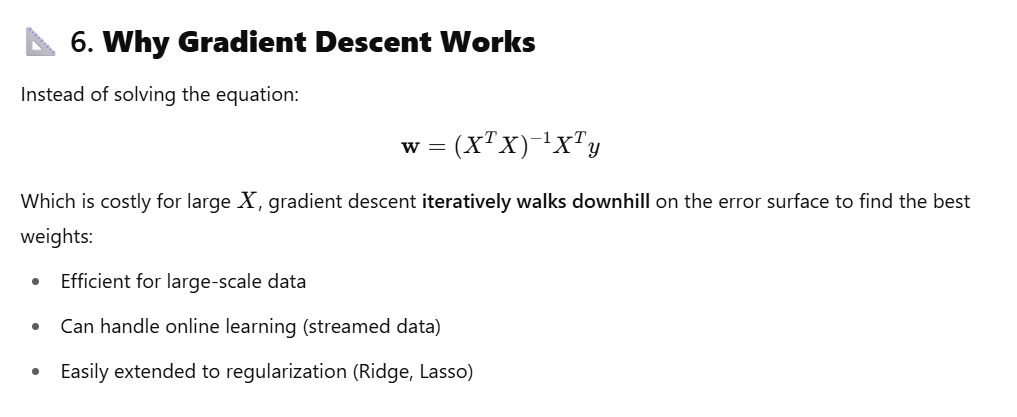


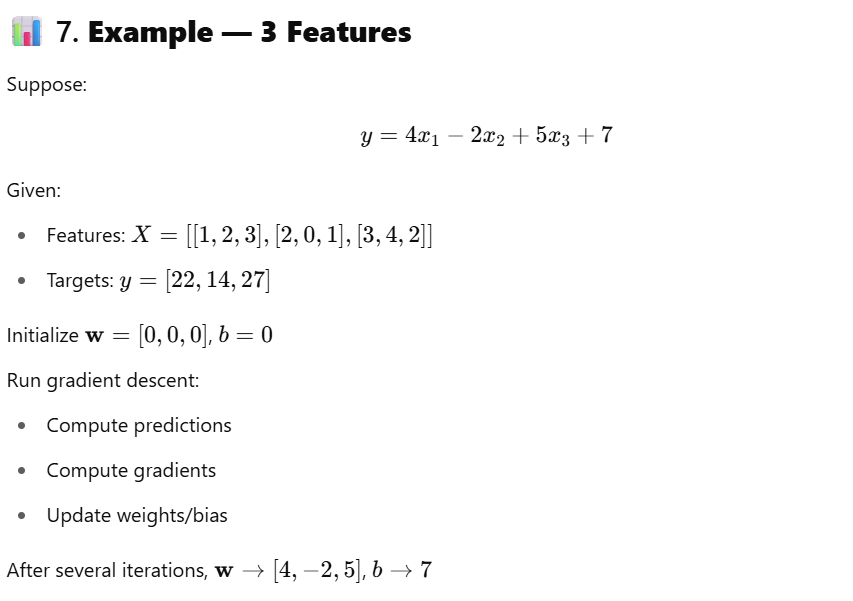




So X.T is same as xi



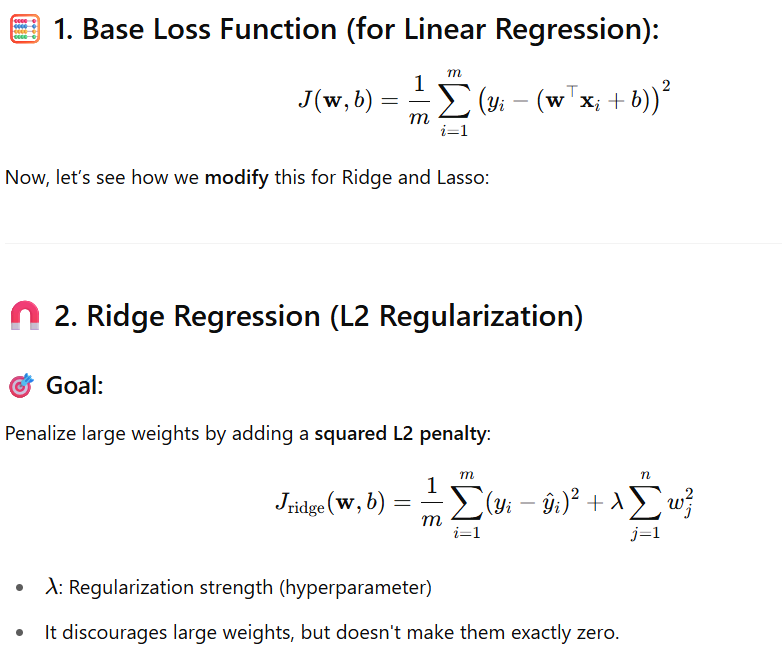


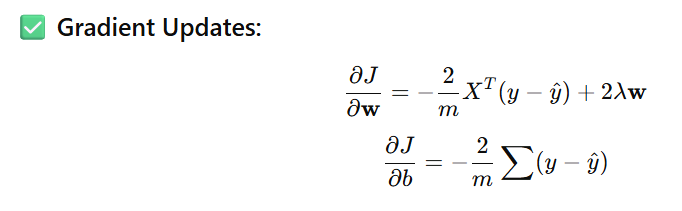


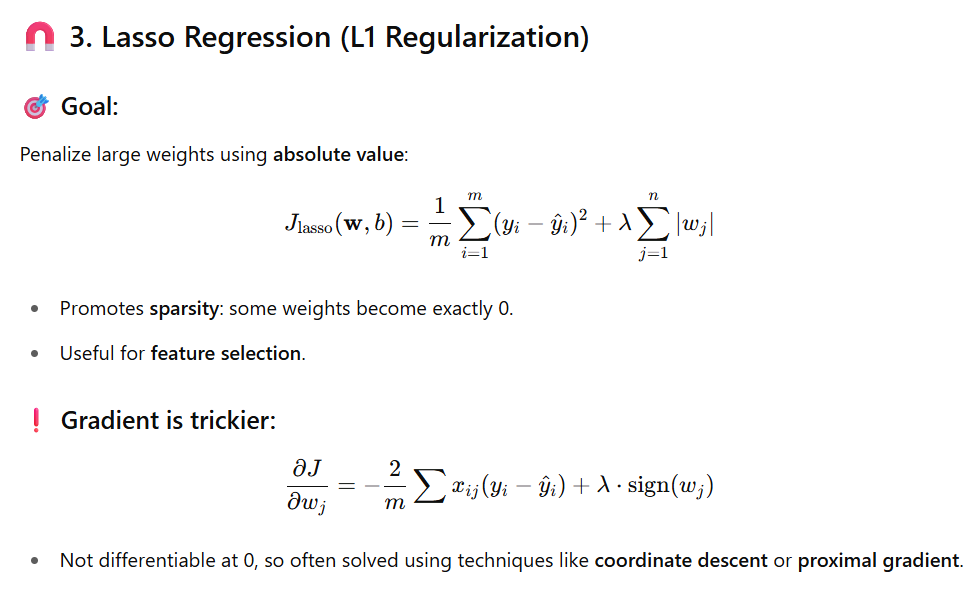
**Why Regularization?**

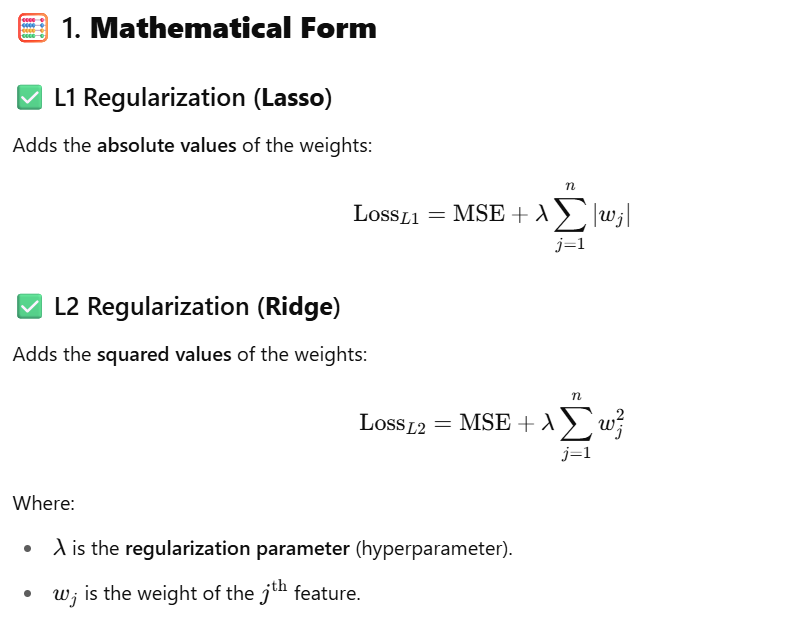
Regularization is used to **prevent overfitting** by penalizing large coefficients (weights).  
In real-world datasets, this is especially helpful when:

* You have **many correlated features**.
* You're fitting **noisy data**.
* You want to **shrink less useful weights toward zero**.



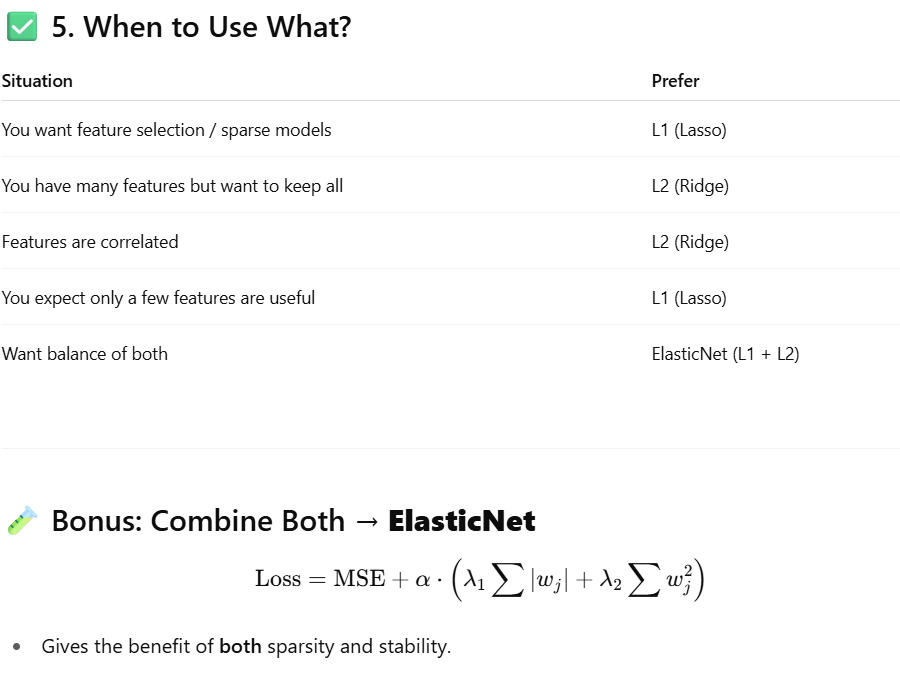






**2. Effect on Weights**

| **Property** | **L1 (Lasso)** | **L2 (Ridge)** |
| --- | --- | --- |
| Penalizes | Absolute values of weights | Squares of weights |
| Effect on weights | **Sparsity** – some become exactly **0** | Shrinks all weights toward **0**, but not exactly |
| Feature selection | ✅ Performs automatic feature selection | ❌ Keeps all features |
| When useful | When you have **many irrelevant features** | When features are correlated or all are important |
| Optimization | Less smooth (non-differentiable at 0) | Smooth and differentiable |
| Used in | Sparse models, interpretable results | Stable models, good generalization |



**4. Python Code: Ridge-Style Regularized Gradient Descent**

Here’s how you can extend our previous class to support **L2 regularization (Ridge)**:

class RidgeGDLinearRegressor:

def \_\_init\_\_(self, lr=0.01, epochs=1000, l2\_penalty=0.1):

self.lr = lr

self.epochs = epochs

self.l2\_penalty = l2\_penalty

self.weights = None

self.bias = 0

self.losses = []

def fit(self, X, y):

if isinstance(X, pd.DataFrame):

X = X.values

if isinstance(y, pd.Series):

y = y.values

m, n = X.shape

self.weights = np.zeros(n)

self.bias = 0

for epoch in range(self.epochs):

y\_pred = np.dot(X, self.weights) + self.bias

error = y - y\_pred

# Gradients with L2 regularization

dw = (-2 / m) \* np.dot(X.T, error) + 2 \* self.l2\_penalty \* self.weights

db = (-2 / m) \* np.sum(error)

# Update

self.weights -= self.lr \* dw

self.bias -= self.lr \* db

loss = (1 / m) \* np.sum(error \*\* 2) + self.l2\_penalty \* np.sum(self.weights \*\* 2)

self.losses.append(loss)

if epoch % 100 == 0:

print(f"Epoch {epoch} | Loss: {loss:.4f} | Weights[:3]: {self.weights[:3]}")

def predict(self, X):

if isinstance(X, pd.DataFrame):

X = X.values

return np.dot(X, self.weights) + self.bias

model = RidgeGDLinearRegressor(lr=0.1, epochs=1000, l2\_penalty=0.5)

model.fit(X\_train, y\_train)

predictions = model.predict(X\_test)

plt.plot(model.losses)

plt.title("Ridge Regression: Loss over Epochs")

plt.xlabel("Epoch")

plt.ylabel("Loss")

plt.grid(True)

plt.show()