Linear Regression

```
import numpy as np

class LinearRegression:
    def __init__(self, lr: float = 0.01, n_iters: int = 100
0):
        """
        lr      : learning rate (step size)
        n_iters : number of gradient descent iterations
        """
        self.lr = lr
        self.n_iters = n_iters
        self.weights = None  # shape: (num_features,)
        self.bias = None     # scalar

    def fit(self, X, y):
        """
        Train the linear regression model using batch gradien
t descent

        X : shape (N, f)  -> N samples, f features
        y : shape (N,)    -> target values
        """
        num_samples, num_features = X.shape

        # Initialize parameters
        self.weights = np.random.randn(num_features)
        self.bias = 0.0
```

```
        for i in range(self.n_iters):

            # -------- Forward Pass --------
            # y_pred = Xw + b
            y_pred = np.dot(X, self.weights) + self.bias

            # -------- Compute Loss (Optional, for monitorin
g) --------
            # Mean Squared Error
            loss = (1 / (2 * num_samples)) * np.sum((y_pred -
y) ** 2)

            # -------- Backward Pass (Gradients) --------
            # Gradient of loss w.r.t weights
            dw = (1 / num_samples) * np.dot(X.T, (y_pred -
y))

            # Gradient of loss w.r.t bias
            db = (1 / num_samples) * np.sum(y_pred - y)

            # -------- Parameter Update --------
            self.weights -= self.lr * dw
            self.bias -= self.lr * db

            # Optional: print loss
            if i % 100 == 0:
                print(f"Iteration {i}, Loss: {loss:.4f}")

        return self

    def predict(self, X):
        """
        Predict using the learned linear model
        """
        return np.dot(X, self.weights) + self.bias
```

1️⃣ First: Big Picture of This Code

This class is implementing **linear regression trained using gradient descent**.

Conceptually, every iteration does:

1. **Forward pass** → make predictions
2. **Compute gradients of the loss** (implicitly)
3. **Update parameters**

⚠️ **The cost (loss) is not explicitly computed**, but its **gradient is**.

That's why you don't "see" cost.

2️⃣ `__init__` — Hyperparameters

```
def__init__(self, lr=0.01, n_iters=1000):
```

- `lr` → learning rate (step size)
- `n_iters` → number of gradient descent steps

📌 These do **not** depend on data — they control optimization.

3️⃣ Parameter Initialization

```
num_samples, num_features = X.shape
self.weights = np.random.rand(num_features)
self.bias =0
```

- `weights` = slope(s) → shape `[f]`
- `bias` = intercept → scalar

📌 This is equivalent to:

$$\hat{y} = Xw + b$$

Random init breaks symmetry (important later for NN).

4️⃣ Forward Pass (Prediction)

```
y_pred = np.dot(X,self.weights) + self.bias
```

This is exactly:

$$\hat{y} = Xw + b$$

✔️ **This is the forward pass**

No learning happens here — just evaluation.

5️⃣ Where Is the Cost Function? 👀

You're right: **it's not explicitly written**.

But the code **assumes Mean Squared Error (MSE)**:

$$J(w,b) = \frac{1}{2N}\sum(\hat{y} - y)^2$$

Instead of computing `J`, the code jumps **directly to its gradients**.

6️⃣ Backpropagation (Gradient Computation)

```
dw = (1 / num_samples) * np.dot(X.T, y_pred - y)
db = (1 / num_samples) * np.sum(y_pred - y)
```

These are **analytical gradients of MSE loss**.

Let's connect them to math:

Gradient w.r.t weights:

$$\frac{\partial J}{\partial w} = \frac{1}{N} X^T (\hat{y} - y)$$

Gradient w.r.t bias:

$$\frac{\partial J}{\partial b} = \frac{1}{N} \sum (\hat{y} - y)$$

📌 This *is* backprop — just very simple because the model is linear.

7️⃣ Parameter Update (Learning Happens Here)

```
self.weights =self.weights -self.lr * dw
self.bias =self.bias -self.lr * db
```

This is classic gradient descent:

$$w := w - \alpha \frac{\partial J}{\partial w}$$

📌 **This is where the model "learns".**

8️⃣ Why Cost Is Often Omitted in Code

Good ML insight here:

> You don't need the loss value to train — you need its gradient.

Loss is usually computed only to:

- Monitor training
- Debug
- Early stopping
- Logging

That's why many educational implementations skip it.

9️⃣ If You Want to Explicitly See the Cost

Add this inside the loop:

```
loss = (1 / (2 * num_samples)) * np.sum((y_pred - y) **2)

if i %100 ==0:
print(f"Iter {i}, Loss:{loss}")
```

Now you'll **see convergence**.

🔁 Mapping Code → ML Concepts (Very Important)

ML Concept	Code Line
Model	`y_pred = Xw + b`
Loss	Implicit MSE
Gradient	`dw`, `db`
Optimization	Gradient descent
Learning	Parameter updates

Linear Regression

Dataset

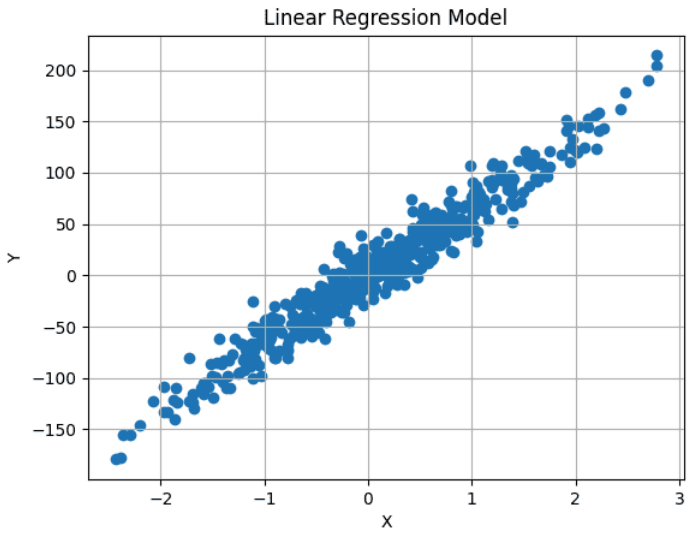
We generate a dummy dataset using Scikit-Learn methods.

The make_regression method provided by Scikit-Learn generates random linear regression datasets, with added Gaussian noise to add some randomness.

```
X, y = datasets.make_regression(n_samples=500, n_features=1, noise=15, random_state=4)
```

We generate 500 random values, each with 1 single feature. Therefore, X has shape (500, 1) and each of the 500 independent X values, has a corresponding y value. So, y also has shape (500,).

Visualized, the dataset looks as follows:



We aim to find a best-fit line that passes through the center of this data, minimizing the average difference between the predicted and original y values.

Intuition

The general equation for a linear line is:

$$y = m * X + b$$

X is numeric and single-valued.

Here, m and b represent the gradient and y-intercept (or bias).

These are unknowns, and varying values of these can generate different lines.

In machine learning, X is dependent on the data, and so are the y values.

- **We only have control over m and b, that act as our model parameters.**

We aim to find optimal values of these two parameters, that generate a line that minimizes the difference between predicted and actual y values.

This extends to the scenario where X is multi-dimensional. In that case, the number of m values will equal the number of dimensions in our data. For example, if our data has three different features, we will have three different m values, called **weights**.

The equation will now become:

$$y = w1 * X1 + w2 * X2 + w3 * X3 + b$$

This can then extend to any number of features.

But how do we know the optimal values of our bias and weight values?

Well, we don't. But we can iteratively find it out using Gradient Descent.

We start with random values and change them slightly for multiple steps until we get close to the optimal values.

First, let us initialize Linear Regression, and we will go over the optimization process in greater detail later.

1️⃣ What "Iterative Approach" Means (The Slow Way)

Suppose you have:

- `N` data points
- `f` features per point

Mathematically:

$$\hat{y}_i = \sum_{j=1}^{f} x_{ij} w_j + b$$

Naive Python code

```
y_pred = []

for i in range(N):
    prediction =0
    for j in range(f):
        prediction += X[i][j] * w[j]
    prediction += b
    y_pred.append(prediction)
```

This is:

- Correct ✅
- Easy to read ✅
- **Very slow** ❌ for large data

Why slow?

- Python loops are slow
- Each operation happens one-by-one
- No hardware optimization

2️⃣ Vectorized Approach (The Fast Way)

Instead of looping over **samples** and **features**, we express the whole computation as **one math operation**:

$$\hat{y} = Xw + b$$

NumPy version

```
y_pred = np.dot(X, w) + b
```

This does **the same thing**, but:

- No Python loops

- Uses optimized C / BLAS libraries
- Can run on SIMD / multi-core / GPU

3️⃣ Why This Works (Key Insight)

Shapes matter — let's look at them

Object	Shape
`X`	`(N, f)`
`w`	`(f,)`
`X @ w`	`(N,)`

Each row of `X` is one data point.

Matrix multiplication automatically computes:

$$\hat{y}_i = \sum_j x_{ij} w_j$$

📌 **The summation is "hidden" inside the matrix multiplication.**

4️⃣ Visual Intuition 🧠

Think of `X` as:

```
[x11 x12 x13]   → dot with [w1 w2 w3]
[x21 x22 x23]
[x31 x32 x33]
```

Each row independently produces **one prediction**.

You're saying:

> "Apply the same weights to every data point, all at once."

```
Element-wise multiply (per row, per feature)

[x11 x12 x13]   *   [w1 w2 w3]   =   [x11·w1  x12·w2  x13·w3]
[x21 x22 x23]       [w1 w2 w3]       [x21·w1  x22·w2  x23·w3]
[x31 x32 x33]       [w1 w2 w3]       [x31·w1  x32·w2  x33·w3]


------------------------------------------------------------


Dot product (row → scalar)

[x11 x12 x13]   ·   [w1 w2 w3]   =   x11·w1 + x12·w2 + x13·w3
[x21 x22 x23]   ·   [w1 w2 w3]   =   x21·w1 + x22·w2 + x23·w3
[x31 x32 x33]   ·   [w1 w2 w3]   =   x31·w1 + x32·w2 + x33·w3


------------------------------------------------------------

Matrix–vector dot (all rows at once)

┌ x11 x12 x13 ┐   ┌ w1 ┐   ┌ ŷ1 ┐
│ x21 x22 x23 │ · │ w2 │ = │ ŷ2 │
└ x31 x32 x33 ┘   └ w3 ┘   └ ŷ3 ┘

ŷ1 = x11·w1 + x12·w2 + x13·w3
ŷ2 = x21·w1 + x22·w2 + x23·w3
ŷ3 = x31·w1 + x32·w2 + x33·w3
```

BackPropagation

Model

$$\hat{y} = Xw + b$$

Loss (Mean Squared Error)

$$J(w, b) = \frac{1}{2N} \sum_{i=1}^{N} (\hat{y}_i - y_i)^2$$

Backpropagation (Gradients)

Gradient w.r.t. weights

$$\boxed{\frac{\partial J}{\partial w} = \frac{1}{N} X^\top (\hat{y} - y)}$$

Gradient w.r.t. bias

$$\boxed{\frac{\partial J}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)}$$

Parameter Update (Gradient Descent)

$$w \leftarrow w - \alpha \frac{\partial J}{\partial w}$$

$$b \leftarrow b - \alpha \frac{\partial J}{\partial b}$$

Expanded (Per-Sample Form)

$$\frac{\partial J}{\partial w_j} = \frac{1}{N} \sum_{i=1}^{N} x_{ij} (\hat{y}_i - y_i)$$

$$\frac{\partial J}{\partial b} = \frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i - y_i)$$

One-Line Lock-In 🧠

> Backprop in linear regression is just the gradient of MSE passed through a single linear layer.