**Linear algebra**

**1. Vectors**

A **vector** is simply a 1D array (list of numbers).

python

CopyEdit

import numpy as np

v = np.array([2, 4, 6])

print(v)

**Operations:**

python

CopyEdit

# Scalar multiplication

print(2 \* v) # Output: [4, 8, 12]

# Vector addition

v2 = np.array([1, 1, 1])

print(v + v2) # Output: [3, 5, 7]

# Dot product (v • v2)

print(np.dot(v, v2)) # Output: 12

**🧱 2. Matrices**

A **matrix** is a 2D array of numbers (rows and columns):

python

CopyEdit

A = np.array([[1, 2],

[3, 4]])

print(A)

**Operations:**

python

CopyEdit

# Matrix transpose

print(A.T)

# Matrix multiplication

B = np.array([[5, 6],

[7, 8]])

print(np.dot(A, B))

# Element-wise multiplication

print(A \* B)

**📐 3. Matrix-Vector Multiplication**

This is common in machine learning (e.g., in linear regression):

Let X=[1234],θ=[0.51]X = \begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}, \theta = \begin{bmatrix} 0.5 \\ 1 \end{bmatrix}X=[13​24​],θ=[0.51​]

python

CopyEdit

X = np.array([[1, 2],

[3, 4]])

theta = np.array([0.5, 1])

result = np.dot(X, theta)

print(result) # Output: [2.5, 5.5]

**🧠 4. Solving Linear Equations**

Given AX=BAX = BAX=B, solve for X:

python

CopyEdit

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

B = np.array([8, 13])

# Solve AX = B

X = np.linalg.solve(A, B)

print(X) # Output: [3. 2.]

**📏 5. Norms and Magnitude**

Useful in measuring **distance** or **error**.

python

CopyEdit

v = np.array([3, 4])

# Euclidean norm (||v||)

print(np.linalg.norm(v)) # Output: 5.0

**💡 Why Learn This?**

Linear algebra powers:

* Machine learning models (like regression)
* Neural networks
* Dimensionality reduction (PCA)
* Computer vision, NLP, and more

**Calculus**

**1. Symbolic Differentiation using SymPy**

python

CopyEdit

import sympy as sp

x = sp.Symbol('x')

f = x\*\*2 + 3\*x + 5

# Derivative

df = sp.diff(f, x)

print("Derivative:", df)

📤 Output:

makefile

CopyEdit

Derivative: 2\*x + 3

You can evaluate this derivative at a point:

python

CopyEdit

df\_val = df.subs(x, 2)

print(df\_val) # Output: 7

**🔁 2. Numerical Derivative (Approximate)**

python

CopyEdit

import numpy as np

def f(x):

return x\*\*2 + 3\*x + 5

def derivative(f, x, h=1e-5):

return (f(x + h) - f(x - h)) / (2\*h)

print(derivative(f, 2)) # Approximate value near 7

✅ This is central in machine learning when calculating gradients.

**∫ 3. Symbolic Integration using SymPy**

python

CopyEdit

f = x\*\*2 + 3\*x + 5

# Indefinite integral

F = sp.integrate(f, x)

print("Integral:", F)

📤 Output:

markdown

CopyEdit

Integral: x\*\*3/3 + 3\*x\*\*2/2 + 5\*x

**Definite Integral:**

python

CopyEdit

area = sp.integrate(f, (x, 0, 2))

print("Definite Integral from 0 to 2:", area)

**🧪 4. Numerical Integration using scipy.integrate**

python

CopyEdit

from scipy.integrate import quad

def f(x):

return x\*\*2 + 3\*x + 5

result, \_ = quad(f, 0, 2)

print("Numerical integration from 0 to 2:", result)

**📈 Application: Gradient Descent**

In machine learning, we use **derivatives** (gradients) to minimize loss functions.

If J(θ)=(θx−y)2J(\theta) = (\theta x - y)^2J(θ)=(θx−y)2, then we need:

python

CopyEdit

# Derivative of loss function J with respect to theta

J = (x\*theta - y)\*\*2

dJ = sp.diff(J, theta)

**📌 Summary**

| **Concept** | **Tool Used** | **Purpose** |
| --- | --- | --- |
| Derivative | sympy.diff() | Symbolic differentiation |
| Numerical Derivative | Custom or NumPy | Approximated slope |
| Integral | sympy.integrate() | Exact area under curve |
| Numerical Integration | scipy.integrate.quad() | Approximate definite integral |

**Gradient descent for linear regression**

**Explanation of the Code**

python

def gradient\_descent(X, y, theta, learning\_rate, iterations):

* **X**: Feature matrix (shape: m × n)
* **y**: Target values (shape: m × 1)
* **theta**: Initial weights (including bias term if used)
* **learning\_rate**: Step size (alpha)
* **iterations**: Number of steps to perform

**m = len(y)**

* Gets the number of training examples mmm

**The Loop: for \_ in range(iterations):**

We repeat the gradient update step multiple times to reduce the cost.

**Step 1: Predictions**

python

CopyEdit

predictions = np.dot(X, theta)

This calculates:

y^=X⋅θ\hat{y} = X \cdot \thetay^​=X⋅θ

**Step 2: Errors**

python

CopyEdit

errors = predictions - y

Calculates:

errors=y^−y\text{errors} = \hat{y} - yerrors=y^​−y

This is how far off the predictions are from the actual values.

**Step 3: Gradient**

python

CopyEdit

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

This computes the **gradient of the cost function**. It tells us how to adjust the parameters.

Mathematically:

∂J∂θ=1mXT⋅(Xθ−y)\frac{\partial J}{\partial \theta} = \frac{1}{m} X^T \cdot (X\theta - y)∂θ∂J​=m1​XT⋅(Xθ−y)

**Step 4: Update Parameters**

python

CopyEdit

theta -= learning\_rate \* gradients

This is the actual gradient descent step:

θ:=θ−α⋅gradient\theta := \theta - \alpha \cdot \text{gradient}θ:=θ−α⋅gradient

**return theta**

After all iterations, return the optimized theta — the best-fit line’s parameters.

**🛠️ How to Write It Yourself**

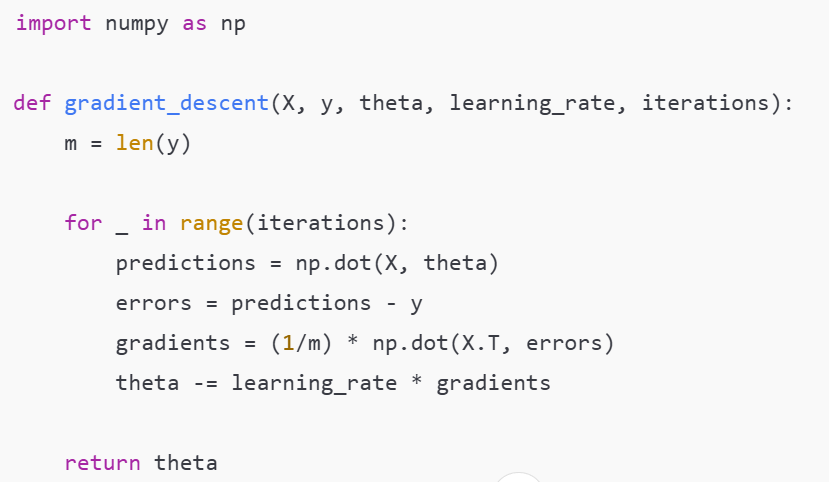
Once you understand the above math, writing your own is simple. Follow this structure:

1. **Initialize parameters**
2. **Loop for N iterations**:
   * Make predictions
   * Compute errors
   * Compute gradients
   * Update parameters
3. **Return final parameters**

**✅ Final Version You Can Write From Scratch:**

python

CopyEdit



**💡 What is theta?**

In **linear regression**, we try to find the best-fit line:

y^=θ0+θ1x\hat{y} = \theta\_0 + \theta\_1 xy^​=θ0​+θ1​x

Or in vector/matrix form (for multiple variables):

y^=X⋅θ\hat{y} = X \cdot \thetay^​=X⋅θ

Where:

* X = input features matrix
* θ (theta) = parameters (weights) of the model
* ŷ = predicted values

**🧠 Goal of Linear Regression**

We want to find the values of theta (parameters) that **minimize the error** between predicted (ŷ) and actual (y) values. This error is measured using **cost/loss function**:

J(θ)=12m∑i=1m(hθ(x(i))−y(i))2J(\theta) = \frac{1}{2m} \sum\_{i=1}^{m} (h\_\theta(x^{(i)}) - y^{(i)})^2J(θ)=2m1​i=1∑m​(hθ​(x(i))−y(i))2

This is also called **Mean Squared Error (MSE)**.

**🔁 Role of Gradient Descent**

**Gradient Descent** is an **optimization algorithm** used to find the values of theta that minimize the cost function J(θ)J(\theta)J(θ).

**Conceptually:**

1. Start with a random theta (weights).
2. Calculate the **gradient** (slope) of the cost function.
3. Update theta in the **opposite direction** of the gradient:

θ:=θ−α⋅∂J(θ)∂θ\theta := \theta - \alpha \cdot \frac{\partial J(\theta)}{\partial \theta}θ:=θ−α⋅∂θ∂J(θ)​

Where:

* + α\alphaα = learning rate (step size)
  + ∂J∂θ\frac{\partial J}{\partial \theta}∂θ∂J​ = gradient of cost function w.r.t theta

1. Repeat (for n iterations).

**🧪 Example Intuition:**

Suppose you're trying to fit a line to this data:

| **x** | **y** |
| --- | --- |
| 1 | 2 |
| 2 | 3 |
| 3 | 4 |

We want to find the line y=θ0+θ1xy = \theta\_0 + \theta\_1 xy=θ0​+θ1​x that best fits it.

You start with random theta = [0, 0]. Gradient descent iteratively **adjusts theta** until the line matches the trend of the data — **reducing prediction error**.

**🛠️ Why Return theta?**

Because after running gradient descent:

✅ theta holds the **best-fit parameters**  
✅ You can now **make predictions** using:

y^=X⋅θ\hat{y} = X \cdot \thetay^​=X⋅θ

**✅ Summary**

| **Step** | **Purpose** |
| --- | --- |
| Define Cost Function | Measure error |
| Compute Gradient | Find direction to reduce error |

**Algorithm: SGD for Linear Regression**

**Goal:**

Minimize the loss:

L=1n∑i=1n(yi−(wxi+b))2L = \frac{1}{n} \sum\_{i=1}^n (y\_i - (wx\_i + b))^2L=n1​i=1∑n​(yi​−(wxi​+b))2

**Steps:**

1. **Initialize** weights w and bias b (e.g., to 0 or random).
2. For each epoch (number of full passes over the dataset):
   * Shuffle the data (optional for real applications).
   * For each data point (x\_i, y\_i):
     1. Predict: y^i=wxi+b\hat{y}\_i = wx\_i + by^​i​=wxi​+b
     2. Compute the error: ei=y^i−yie\_i = \hat{y}\_i - y\_iei​=y^​i​−yi​
     3. Compute gradients:
        + ∂L∂w=2xi⋅ei\frac{\partial L}{\partial w} = 2x\_i \cdot e\_i∂w∂L​=2xi​⋅ei​
        + ∂L∂b=2⋅ei\frac{\partial L}{\partial b} = 2 \cdot e\_i∂b∂L​=2⋅ei​
     4. Update:
        + w=w−η⋅∂L∂ww = w - \eta \cdot \frac{\partial L}{\partial w}w=w−η⋅∂w∂L​
        + b=b−η⋅∂L∂bb = b - \eta \cdot \frac{\partial L}{\partial b}b=b−η⋅∂b∂L​
3. Repeat until convergence.

**✏️ Example: One Epoch Trace with 2 Data Points**

Let's take a simple dataset:

ini

CopyEdit

x = [1, 2]

y = [5, 7]

We'll try to fit y=wx+by = wx + by=wx+b

**Initial Parameters:**

ini

CopyEdit

w = 0

b = 0

learning\_rate = 0.1

**➤ First Data Point (x = 1, y = 5)**

1. **Prediction:**

y^=w⋅x+b=0⋅1+0=0\hat{y} = w \cdot x + b = 0 \cdot 1 + 0 = 0y^​=w⋅x+b=0⋅1+0=0

1. **Error:**

e=y^−y=0−5=−5e = \hat{y} - y = 0 - 5 = -5e=y^​−y=0−5=−5

1. **Gradients:**

dw=2⋅x⋅e=2⋅1⋅(−5)=−10dw = 2 \cdot x \cdot e = 2 \cdot 1 \cdot (-5) = -10dw=2⋅x⋅e=2⋅1⋅(−5)=−10 db=2⋅e=2⋅(−5)=−10db = 2 \cdot e = 2 \cdot (-5) = -10db=2⋅e=2⋅(−5)=−10

1. **Update:**

w=w−lr⋅dw=0−0.1⋅(−10)=1w = w - lr \cdot dw = 0 - 0.1 \cdot (-10) = 1w=w−lr⋅dw=0−0.1⋅(−10)=1 b=b−lr⋅db=0−0.1⋅(−10)=1b = b - lr \cdot db = 0 - 0.1 \cdot (-10) = 1b=b−lr⋅db=0−0.1⋅(−10)=1

**➤ Second Data Point (x = 2, y = 7)**

1. **Prediction:**

y^=1⋅2+1=3\hat{y} = 1 \cdot 2 + 1 = 3y^​=1⋅2+1=3

1. **Error:**

e=3−7=−4e = 3 - 7 = -4e=3−7=−4

1. **Gradients:**

dw=2⋅2⋅(−4)=−16dw = 2 \cdot 2 \cdot (-4) = -16dw=2⋅2⋅(−4)=−16 db=2⋅(−4)=−8db = 2 \cdot (-4) = -8db=2⋅(−4)=−8

1. **Update:**

w=1−0.1⋅(−16)=1+1.6=2.6w = 1 - 0.1 \cdot (-16) = 1 + 1.6 = 2.6w=1−0.1⋅(−16)=1+1.6=2.6 b=1−0.1⋅(−8)=1+0.8=1.8b = 1 - 0.1 \cdot (-8) = 1 + 0.8 = 1.8b=1−0.1⋅(−8)=1+0.8=1.8

**✅ Final after 1 epoch:**

* w = 2.6, b = 1.8
* This means the model is now: y=2.6x+1.8y = 2.6x + 1.8y=2.6x+1.8

You can repeat this for more epochs to get closer to the optimal fit.

**Bayes theorem**

**Conceptual Understanding**

**🔸 What is Bayes’ Theorem?**

Bayes' Theorem is a way of **reversing conditional probabilities**. It answers the question:

"Given that we observed **effect E**, what is the probability of **cause C**?"

Mathematically:

P(C∣E)=P(E∣C)⋅P(C)P(E)P(C|E) = \frac{P(E|C) \cdot P(C)}{P(E)}P(C∣E)=P(E)P(E∣C)⋅P(C)​

Where:

* P(C∣E)P(C|E)P(C∣E): Posterior — probability of cause **C** given evidence **E**
* P(E∣C)P(E|C)P(E∣C): Likelihood — how likely evidence **E** is if cause **C** is true
* P(C)P(C)P(C): Prior — probability of cause **C** before seeing the evidence
* P(E)P(E)P(E): Evidence — total probability of the evidence **E**

**🧠 Intuition**

Imagine a medical test for a disease:

* Only 1% of people have the disease: P(D)=0.01P(D) = 0.01P(D)=0.01
* Test is 99% accurate: P(Positive∣D)=0.99P(\text{Positive} | D) = 0.99P(Positive∣D)=0.99
* But it can also falsely say “positive” for 5% of healthy people: P(Positive∣¬D)=0.05P(\text{Positive} | \neg D) = 0.05P(Positive∣¬D)=0.05

Now, someone tests positive. What’s the **actual chance they have the disease**?

This is **exactly** where Bayes’ Theorem helps — it balances the prior chance with the strength of the evidence.

**📐 Mathematical Proof**

We start from the **definition of conditional probability**:

P(A∣B)=P(A∩B)P(B)(1)P(A|B) = \frac{P(A \cap B)}{P(B)} \quad \text{(1)}P(A∣B)=P(B)P(A∩B)​(1) P(B∣A)=P(A∩B)P(A)(2)P(B|A) = \frac{P(A \cap B)}{P(A)} \quad \text{(2)}P(B∣A)=P(A)P(A∩B)​(2)

From both:

P(A∩B)=P(A∣B)⋅P(B)=P(B∣A)⋅P(A)P(A \cap B) = P(A|B) \cdot P(B) = P(B|A) \cdot P(A)P(A∩B)=P(A∣B)⋅P(B)=P(B∣A)⋅P(A)

So, equating the right sides:

P(A∣B)⋅P(B)=P(B∣A)⋅P(A)P(A|B) \cdot P(B) = P(B|A) \cdot P(A)P(A∣B)⋅P(B)=P(B∣A)⋅P(A)

Divide both sides by P(B)P(B)P(B):

P(A∣B)=P(B∣A)⋅P(A)P(B)P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}P(A∣B)=P(B)P(B∣A)⋅P(A)​

That’s Bayes’ Theorem:

P(A∣B)=P(B∣A)⋅P(A)P(B)\boxed{P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}}P(A∣B)=P(B)P(B∣A)⋅P(A)​​

**💡 Real-World Applications**

**1. Medical Diagnosis**

Given a test result, estimate disease probability.

✅ Posterior = How likely is the disease now?  
🧪 Likelihood = How accurate is the test?  
📊 Prior = How common is the disease in the population?  
📈 Evidence = Probability of testing positive

**2. Spam Filtering (Naive Bayes Classifier)**

Given an email, decide whether it's spam.

* Features = words in email (e.g., “free”, “money”)
* Learn how often spam emails contain those words (likelihood)
* Learn how often emails are spam in general (prior)
* Apply Bayes to classify new email

**3. Weather Forecasting**

You can calculate the probability it will rain given that the sky is cloudy using past data (priors) and current sensor data (likelihoods).

**4. Fault Diagnosis in Machines**

If a sensor triggers an alert, Bayes’ theorem helps calculate which component is most likely at fault based on prior failure rates and test patterns.

**Problem: Drug Test Case (Real-Life Example)**

Suppose a **new drug test** detects **drug users** with:

* 98% **true positive rate**: P(Positive∣User)=0.98P(\text{Positive} | \text{User}) = 0.98P(Positive∣User)=0.98
* 10% **false positive rate**: P(Positive∣Non-user)=0.10P(\text{Positive} | \text{Non-user}) = 0.10P(Positive∣Non-user)=0.10

In a population, only **5% of people** actually use the drug:

* P(User)=0.05P(\text{User}) = 0.05P(User)=0.05, so P(Non-user)=0.95P(\text{Non-user}) = 0.95P(Non-user)=0.95

**❓Question:**

If a person tests **positive**, what’s the probability they actually **use the drug**?

That is:

P(User∣Positive)=?P(\text{User}|\text{Positive}) = ?P(User∣Positive)=?

**✅ Step-by-Step Solution Using Bayes' Theorem**

Bayes’ Theorem:

P(User∣Positive)=P(Positive∣User)⋅P(User)P(Positive)P(\text{User}|\text{Positive}) = \frac{P(\text{Positive}|\text{User}) \cdot P(\text{User})}{P(\text{Positive})}P(User∣Positive)=P(Positive)P(Positive∣User)⋅P(User)​

**Step 1: Identify known values**

* P(Positive∣User)=0.98P(\text{Positive} | \text{User}) = 0.98P(Positive∣User)=0.98
* P(Positive∣Non-user)=0.10P(\text{Positive} | \text{Non-user}) = 0.10P(Positive∣Non-user)=0.10
* P(User)=0.05P(\text{User}) = 0.05P(User)=0.05
* P(Non-user)=0.95P(\text{Non-user}) = 0.95P(Non-user)=0.95

**Step 2: Compute total probability of testing positive (denominator)**

P(Positive)=P(Positive∣User)⋅P(User)+P(Positive∣Non-user)⋅P(Non-user)P(\text{Positive}) = P(\text{Positive}|\text{User}) \cdot P(\text{User}) + P(\text{Positive}|\text{Non-user}) \cdot P(\text{Non-user})P(Positive)=P(Positive∣User)⋅P(User)+P(Positive∣Non-user)⋅P(Non-user) =(0.98)(0.05)+(0.10)(0.95)=0.049+0.095=0.144= (0.98)(0.05) + (0.10)(0.95) = 0.049 + 0.095 = 0.144=(0.98)(0.05)+(0.10)(0.95)=0.049+0.095=0.144

**Step 3: Apply Bayes' Theorem**

P(User∣Positive)=0.98⋅0.050.144=0.0490.144≈0.340P(\text{User}|\text{Positive}) = \frac{0.98 \cdot 0.05}{0.144} = \frac{0.049}{0.144} \approx 0.340P(User∣Positive)=0.1440.98⋅0.05​=0.1440.049​≈0.340

**✅ Answer:**

The person has **only a 34% chance** of actually using the drug, even though they tested positive!

**🧠 Approach for Any Bayes Problem (Checklist)**

1. **Define the event of interest**  
   → What are you trying to find? (e.g., P(A∣B)P(A|B)P(A∣B): the chance of being a drug user given a positive test)
2. **Identify the "prior" probability**  
   → What's the baseline chance before the evidence? (e.g., P(A)=0.05P(A) = 0.05P(A)=0.05)
3. **Identify the "likelihood"**  
   → How likely is the evidence if the hypothesis is true? (e.g., P(B∣A)=0.98P(B|A) = 0.98P(B∣A)=0.98)
4. **Find total probability of evidence (denominator)**  
   → Consider **all possible ways** the evidence can occur.
5. **Plug into Bayes’ formula**

P(A∣B)=P(B∣A)⋅P(A)P(B)P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)}P(A∣B)=P(B)P(B∣A)⋅P(A)​

1. **Interpret the result**  
   → Does the result match your expectations? Is the conclusion useful in real life?

**🔗 Real-World Uses:**

* **Medical testing** (HIV, COVID): Probability of having a disease after a test
* **Machine fault prediction**: What's the chance of part failure after seeing a warning?
* **Finance**: What's the chance a customer will default given behavior?
* **AI & ML**: Naive Bayes classifiers for email spam or sentiment analysis