



Linear and Logistic Regression from Scratch

From-scratch code | Mean-squared-error and log-loss gradient descent in NumPy, checked against least squares, with the learning-rate limit, shape bugs and three extensions (ridge, mini-batch SGD, softmax regression) solved and run

#FromScratch #NumPy #Regression #GradientDescent

The idea and the maths in one line

Regression by gradient descent

In one sentence

Start with weights at zero, measure how wrong the predictions are, nudge every weight in the direction that lowers the error, repeat.

Practitioner

Linear: $\hat{y} = Xw + b$ with MSE. Logistic: $p = \sigma(Xw + b)$ with log-loss. Both gradients have the same form $X^T(\text{prediction} - \text{target}) / N$.

- the factor 2 of MSE is the only difference
- the sigmoid's derivative cancels inside the log-loss gradient

Expert

Both losses are convex in (w, b) : gradient descent with a small enough η reaches the unique minimum, so a closed-form solver is a perfect reference for the linear case.

- η limit = 2 / largest Hessian eigenvalue
- log-loss has no closed form: check with scikit-learn or a second method

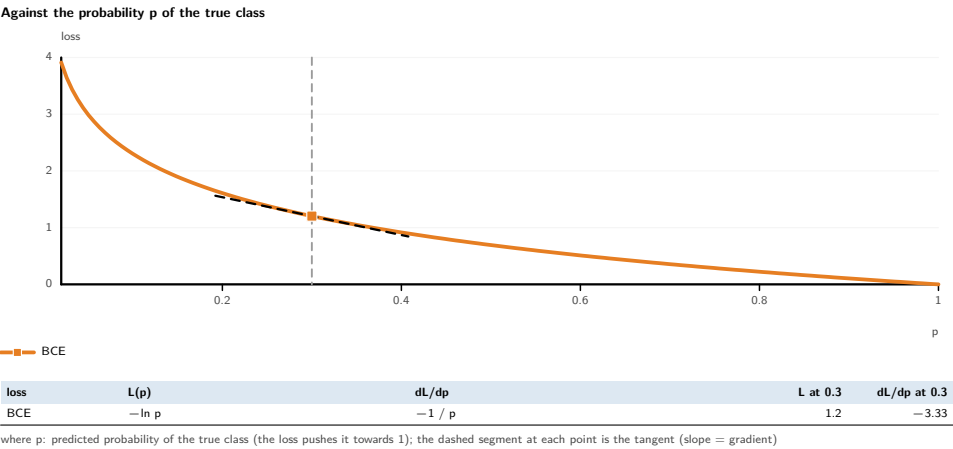
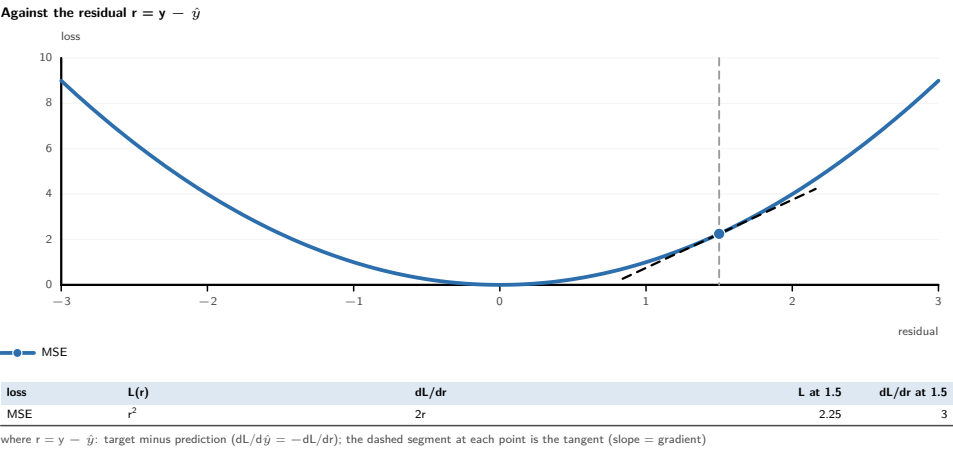
Analogy

Like tuning a radio: listen to the noise (loss), turn each knob a little in the direction that reduces it (gradient), stop when it is clear.

Same model family, two losses: the code changes in three lines

Model	Loss	Gradient used in the code
Linear	$L = \frac{1}{N} \sum_i (\hat{y}_i - y_i)^2$	$\nabla_w L = \frac{2}{N} X^{\text{top}}(\hat{y} - y); \nabla_b L = \frac{2}{N} \sum_i (\hat{y}_i - y_i)$
Logistic	$L = -\frac{1}{N} \sum_i [y_i \ln p_i + (1 - y_i) \ln(1 - p_i)]$	$\nabla_w L = \frac{1}{N} X^{\text{top}}(p - y); \nabla_b L = \frac{1}{N} \sum_i (p_i - y_i)$

where X = feature matrix, one row per example (shape (N, D)); w, b = weight vector and bias $((D,)$ and scalar); y = targets: real numbers (linear) or 0/1 labels (logistic) $((N,))$; $\hat{y}, p$ = predictions $Xw + b$ and probabilities $\sigma(Xw + b)$ $((N,))$; N = number of training examples (count)
What the two programs implement. Theory: machine-learning sheets 02 (linear regression) and 03 (logistic regression); maths: optimisation and convexity



The two losses at one point: MSE grows quadratically with the residual; log-loss explodes as the probability of the true class goes to 0

⌘ Program 1: linear regression

linreg.py (24 lines, NumPy only) with its output
import numpy as np

```
rng = np.random.default_rng(0)
N, D = 200, 3
X = rng.normal(size=(N, D))          # (N, D) features
true_w = np.array([2.0, -1.0, 0.5])
y = X @ true_w + 4.0 + 0.1 * rng.normal(size=N) # (N,) targets
```

```
w, b, lr = np.zeros(D), 0.0, 0.1
for epoch in range(201):
```

```
    y_hat = X @ w + b                # (N,) predictions
    err = y_hat - y                  # (N,) residuals
    loss = (err ** 2).mean()          # MSE
    grad_w = 2 / N * X.T @ err       # (D,)
    grad_b = 2 / N * err.sum()
    w -= lr * grad_w
    b -= lr * grad_b
    if epoch % 50 == 0:
        print(f"epoch {epoch:3d} mse {loss:.4f}")
```

```
Xb = np.c_[X, np.ones(N)]           # add bias column
w_ne = np.linalg.lstsq(Xb, y, rcond=None)[0]
print("GD   w, b:", w.round(3), round(b, 3))
print("lstsq w, b:", w_ne[:3].round(3), w_ne[3].round(3))
# expected output (NumPy 1.26):
# epoch   0 mse 19.7272
# epoch  50 mse 0.0100
# epoch 100 mse 0.0100
# epoch 150 mse 0.0100
# epoch 200 mse 0.0100
# GD   w, b: [ 1.991 -1.01  0.505] 3.996
# lstsq w, b: [ 1.991 -1.01  0.505] 3.996
```

Lines	What it does	Why
3-7	200 rows of 3 Gaussian features; y from known weights (2, -1, 0.5), bias 4 and noise $\sigma = 0.1$	known truth lets you see if the fit recovers it
9	weights and bias start at zero, $\eta = 0.1$	MSE is convex: any start works
11-13	forward pass, residuals, MSE	loss printed before the update of that epoch
14-15	gradients $2X^T \text{err} / N$ and $2 \sum \text{err} / N$	X^T sums each feature's contribution over the rows
16-17	in-place update against the gradient	-- keeps the same array
21-24	exact least-squares solution with a column of ones for the bias	reference: GD must agree to 3 decimals

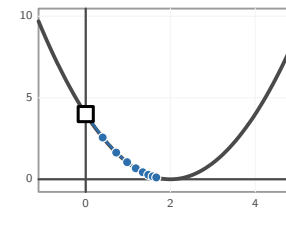
Line by line

📘 INFO

The final MSE 0.0100 equals the noise variance $0.1^2 = 0.01$: the model is as good as the data allows. The recovered weights differ from (2, -1, 0.5, 4) by the noise, and GD matches lstsq exactly.

$\eta = 0.1$

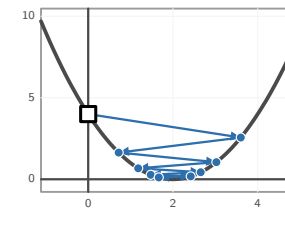
converges



$x_0 = 0, x_1 = 0.4, x_2 = 0.72, x_3 = 0.976, \dots, x_8 = 1.66$
 $|\Delta x|$ shrinks: $0.4 \rightarrow 0.0839$

$\eta = 0.9$

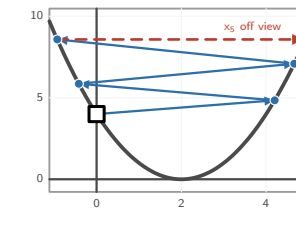
oscillates



$x_0 = 0, x_1 = 3.6, x_2 = 0.72, x_3 = 3.02, \dots, x_8 = 1.66$
 Δx flips sign, $|\Delta x|$ $3.6 \rightarrow 0.755$

$\eta = 1.05$

diverges



$x_0 = 0, x_1 = 4.2, x_2 = -0.42, x_3 = 4.66, \dots, x_8 = -2.29$
 $|\Delta x|$ grows: $4.2 \rightarrow 8.18$

Update rule: $x_{n+1} = x_n - \eta \cdot f'(x_n)$, on $f(x) = (x - 2)^2$

First step with $\eta = 0.1$: $x_1 = 0 - 0.1 \cdot f'(0) = 0 - 0.1 \cdot (-4) = 0.4$

Near the minimum $x^* \approx 2$, $f''(x^*) \approx 2$: steps shrink only if $\eta < 2 / f''(x^*) = 1$; above $\eta = 1 / f''(x^*) = 0.5$ they overshoot and oscillate

Verdict: converges = steps shrink without changing direction; oscillates = steps flip sign (overshoot); diverges = steps grow

Where: x_n : parameter value after n steps; η : learning rate (step size); f : the loss to minimise; $f'(x_n)$: its slope at x_n , by central difference $[f(x+h) - f(x-h)] / (2h)$; $\Delta x = x_{n+1} - x_n$: one step; x^* : the minimum in view; $f''(x^*)$: curvature there

Why η matters. On the 1-D slice $f(w) = (w - 2)^2$, $f'' = 2$, so η must stay below $2 / 2 = 1$. On the real data the largest Hessian eigenvalue of $2X^T X / N$ is 2.207, so the limit is 0.906: $\eta = 0.9$ still converges (mse 0.0103 at epoch 100), $\eta = 1.05$ diverges (mse 8.2e9 at epoch 50)

⌘ Program 2: logistic regression

logreg.py (27 lines) with its output
import numpy as np

```
rng = np.random.default_rng(1)
n = 100
X0 = rng.normal(loc=[-1, -1], size=(n, 2)) # class 0 blob
X1 = rng.normal(loc=[1, 1], size=(n, 2))    # class 1 blob
X = np.vstack([X0, X1])                    # (200, 2)
y = np.r_[np.zeros(n), np.ones(n)]         # (200,)
```

```
def sigmoid(z):
```

```
    return 1 / (1 + np.exp(-z))
```

```
w, b, lr = np.zeros(2), 0.0, 0.5
for epoch in range(301):
    p = sigmoid(X @ w + b)                # (200,) P(y=1|x)
    eps = 1e-12
    loss = -np.mean(y * np.log(p + eps)
                    + (1 - y) * np.log(1 - p + eps))
    grad_z = (p - y) / len(y)              # dL/dz, (200,)
    w -= lr * X.T @ grad_z
```

```

b -= lr * grad_z.sum()
if epoch % 100 == 0:
    acc = ((p > 0.5) == y).mean()
    print(f"epoch {epoch:3d}  logloss {loss:.4f}  acc {acc:.3f}")
print("w:", w.round(3), "b:", round(b, 3))
x_new = np.array([0.5, 0.5])
print("P(y=1 | x=[0.5, 0.5]) =", sigmoid(x_new @ w + b).round(3))
# expected output:
# epoch   0  logloss 0.6931  acc 0.500
# epoch 100  logloss 0.0976  acc 0.965

# epoch 200  logloss 0.0881  acc 0.965
# epoch 300  logloss 0.0849  acc 0.965
# w:  [3.052 3.12 ] b:  0.319
# P(y=1 | x=[0.5, 0.5]) = 0.968

```

sigmoid($X @ w + b$) — Turns the linear score into a probability; at $w = 0$ every $p = 0.5$, so the first log-loss is $\ln 2 = 0.6931$.

eps = 1e-12 — Keeps $\log(0) = -\infty$ away when a probability saturates at exactly 0 or 1 in float64.

grad_z = (p - y) / N — dL/dz for log-loss with a sigmoid: the σ' term cancels, so the gradient has the same shape as MSE's without the 2.

(p > 0.5) == y — Accuracy at the 0.5 threshold; at epoch 0 every $p = 0.5$ is not > 0.5 , so all predictions are class 0 and accuracy is 0.500.

Check the last line by hand

FORMULA $p = 1 / (1 + \exp(-(w_1 \times x_1 + w_2 \times x_2 + b)))$

WHERE

- p = predicted probability of class 1 (0-1)
- w_1 = learned weight of feature 1 (per feature unit)
- w_2 = learned weight of feature 2 (per feature unit)
- x_1 = feature 1 of the new point (feature unit)
- x_2 = feature 2 of the new point (feature unit)
- b = learned bias (logit)

WORKED

```

w1 = 3.052
w2 = 3.12
x1 = 0.5
x2 = 0.5
b = 0.319

```

READ IT $z = 3.405$, so the point is 3.4 logits on the class-1 side of the boundary $w \cdot x + b = 0$: $p \approx 0.968$, as printed.

MISTAKES

- the log-loss keeps falling after accuracy stops moving: weights grow to make p more confident
- on perfectly separable data $\|w\|$ grows without bound; add an L2 penalty

Shapes and common bugs

Rules: 1. align the shapes on the right; 2. a missing leading dimension counts as 1; 3. in each dimension the sizes must be equal or one of them 1 (the 1 is stretched); otherwise the operation fails.

stable softmax: logits Z (300, 3) $\oplus$ $Z.\max(1, \text{keepdims=True})$ (300, 1) $\rightarrow$ (300, 3)

logits Z	300	3
$Z.\max(1, \text{keepdims=True})$	300	1
result	300	3
	equal	1 stretched

silent bug: y_hat (200,) $\oplus$ y as a column (200, 1) $\rightarrow$ (200, 200)

y_hat	(1)	200
y as a column	200	1
result	200	200
	missing $\rightarrow$ 1	1 stretched

missing keepdims: logits Z (300, 3) $\oplus$ $Z.\max(1)$ (300,) $\rightarrow$ $\times$ error

logits Z	300	3
$Z.\max(1)$	(1)	300
result	300	$\times$
	missing $\rightarrow$ 1	$\times$ 3 vs 300

$\times$ Cannot broadcast: dimension -1 (counted from the right): sizes 3 and 300 differ and neither is 1.

Three shape pairs from this sheet. The second is the silent killer: a target of shape (N, 1) against predictions of shape (N,) broadcasts to an (N, N) error matrix and the MSE is wrong without any error message

Loss becomes nan or inf after a few epochs

HIGH

LIKELY CAUSES

- Learning rate above 2 / λ_max
- Unscaled features (one column in thousands)



HOW TO CONFIRM

Print the loss every epoch; standardise and retry

```
X = (X - X.mean(0)) / X.std(0)
```

FIX

- Lower η by 3-10 $\times$
- Standardise every feature

MSE stuck far above the noise level, weights look random

MEDIUM

LIKELY CAUSES

- y has shape (N, 1): err becomes (N, N)
- Gradient sign flipped ($w \leftarrow lr * \text{grad}$)



HOW TO CONFIRM

Assert shapes before the loop

```
assert y.shape == (N,) and err.shape == (N,)
```

FIX

- $y = y.\text{ravel}()$
- Update with $-$ against the gradient

Log-loss prints nan, accuracy still fine

LOW

LIKELY CAUSES

- $\log(0)$ when p saturates to 0 or 1



HOW TO CONFIRM

Look for $p == 0$ or $p == 1$

```
print((p == 0).sum(), (p == 1).sum())
```

FIX

- Add eps inside the log
- Or compute the loss from logits: `np.logaddexp(0, -z)` form

Extensions with solutions

```
1. Ridge (L2) penalty: append to linreg.py
lam = 0.1                                # L2 strength
w, b = np.zeros(D), 0.0
for epoch in range(500):
    err = X @ w + b - y
    w -= lr * (2 / N * X.T @ err + 2 * lam * w)  # + d(lam|w|^2)/dw
    b -= lr * 2 / N * err.sum()                 # bias not penalised
print("ridge w:", w.round(3))
# output: ridge w: [ 1.81 -0.921  0.444]
```

```
2. Mini-batch SGD (batch 32): append to linreg.py
w, b, bs = np.zeros(D), 0.0, 32
for epoch in range(20):
    idx = rng.permutation(N)                # reshuffle every epoch
    for s in range(0, N, bs):
        i = idx[s:s + bs]
        err = X[i] @ w + b - y[i]
        w -= lr * 2 / len(i) * X[i].T @ err
        b -= lr * 2 / len(i) * err.sum()
mse = ((X @ w + b - y) ** 2).mean()
print("SGD w:", w.round(3), "mse", mse.round(4))
# output: SGD w: [ 1.98 -1.017  0.523] mse 0.0106
```

```
3. Softmax regression, 3 classes: append to logreg.py
K = 3
C = np.array([[ -2, 0], [2, 0], [0, 2.5]])
Xs = np.vstack([rng.normal(c, 1, (100, 2)) for c in C]) # (300, 2)
ys = np.repeat(np.arange(K), 100)
W, bk = np.zeros((2, K)), np.zeros(K)
for epoch in range(300):
    Z = Xs @ W + bk                                # (300, K) logits
    Z -= Z.max(1, keepdims=True)                   # stable softmax
    P = np.exp(Z); P /= P.sum(1, keepdims=True)
    G = P.copy(); G[np.arange(300), ys] -= 1; G /= 300 # dL/dZ
    W -= 0.5 * Xs.T @ G; bk -= 0.5 * G.sum(0)
loss = -np.log(P[np.arange(300), ys]).mean()
print(f"softmax loss {loss:.4f} acc {(P.argmax(1) == ys).mean():.3f}")
# output: softmax loss 0.2361 acc 0.903
```

💡 TIP

Reading the extension outputs: ridge shrinks every weight by about 10 % ($1.991 \rightarrow 1.81$, $0.505 \rightarrow 0.444$), the price of lower variance; SGD reaches nearly the batch solution (mse 0.0106 vs 0.0100) after 20 epochs of 7 small steps; softmax regression on overlapping blobs plateaus near 90 % accuracy because the classes overlap, not because of a bug. The softmax gradient $P - \text{onehot}$ is the K-class version of $p - y$.

Myths, recap and self-test

YOU MAY HEAR

"Logistic regression is a regression model"

BUSTED

✓ ACTUALLY

It is a classifier

It regresses the log-odds, then a threshold on p gives the class.

YOU MAY HEAR

"Use MSE for 0/1 labels, it is simpler"

BUSTED

✓ ACTUALLY

Log-loss gives larger gradients when wrong

With a sigmoid, MSE's gradient vanishes for confident mistakes; a smaller η never fixes that.

Same gradient shape

$X^T(\text{prediction} - \text{target}) / N$

Check with lstsq

GD equals the exact solution

$\eta < 2 / \lambda_{\max}$
0.906 on this data

Shapes (N,) not (N, 1)

or the loss silently goes wrong

Initial log-loss $\ln 2$

0.6931 at $w = 0$

Softmax: $P - \text{onehot}$

K-class version of $p - y$

Q1 Write the MSE gradient for w in NumPy.

A $2 / N * X.T @ (X @ w + b - y)$: $X.T$ sums each feature's contribution over the N rows.

Q2 Why does the log-loss gradient have no σ' term?

A $dL/dp \times dp/dz = [(p - y) / (p(1 - p))] \times p(1 - p) = p - y$; the sigmoid derivative cancels.

Q3 What first log-loss do you expect at $w = 0$, $b = 0$?

A $\ln 2 = 0.6931$, because every $p = 0.5$ and $-\ln 0.5 = \ln 2$.

Q4 How do you know program 1 is correct beyond a falling loss?

A Its weights equal `np.linalg.lstsq`'s to 3 decimals and its MSE equals the noise variance 0.01.

Q5 What learning rate makes program 1 diverge, and why?

A Any η above $2 / 2.207 \approx 0.906$, the inverse of half the largest eigenvalue of the Hessian $2X^T X / N$.

Q6 What happens if y has shape $(N, 1)$?

A $y_{\text{hat}} - y$ broadcasts to (N, N) ; the loss and gradients are wrong without any error.

Q7 What does the ridge penalty add to the gradient?

A $2\lambda w$ (bias excluded); it shrinks the weights, here by about 10 % with $\lambda = 0.1$.

Q8 Why subtract the row maximum before $\exp$ in softmax?

A Avoids overflow of $\exp$ for large logits; softmax is unchanged by adding a constant to a row.