

Linear regression

☑ Trắc nghiệm 57 📁 Thực hành 5 ⬇ PDF bài

Một căn chung cư 70 m², cách trung tâm 8 km, rao bán 4,2 tỷ. Giá đó cao hay thấp? Nếu có giá của 200 căn đã bán quanh đó, ta có thể ước lượng: căn rộng hơn thì đắt hơn, xa trung tâm thì rẻ hơn. Linear regression (hồi quy tuyến tính) viết cách ước lượng ấy thành công thức và tìm hệ số từ dữ liệu. Đây là model đơn giản nhất của học máy, và các model sau, từ logistic regression tới mạng neural, đều dùng lại ý của nó.

MỤC TIÊU

- Hiểu model tuyến tính và ý nghĩa từng hệ số.
- Tìm hệ số bằng công thức nghiệm và bằng gradient descent.
- Biết vì sao learning rate quá lớn làm loss tăng vọt, và vì sao cần chuẩn hóa đặc trưng.
- Dùng scikit-learn và tự cài lại bằng NumPy.

Dữ liệu và mô hình

Ta bắt đầu từ dữ liệu. Để không phải tải dữ liệu từ Internet, ta tự sinh 200 căn hộ theo một công thức định giá có sẵn: $\text{giá} = 0.05 \cdot \text{diện tích} - 0.15 \cdot \text{khoảng cách} + 1$ (tỷ đồng), cộng thêm một chút nhiễu. Như vậy ta biết trước đáp án và kiểm được model học đúng hay sai.

Python

```
import numpy as np

rng = np.random.default_rng(0)
n = 200
area = rng.uniform(30, 120, n) # diện tích, m2
dist = rng.uniform(1, 15, n)   # khoảng cách đến trung tâm, km
y = 0.05 * area - 0.15 * dist + 1.0 + rng.normal(0, 0.3, n) # giá, tỷ đồng

X = np.column_stack([area, dist])
```

```
print(X.shape, y.shape)
print(X[:3].round(1))
print(y[:3].round(2))
```

KẾT QUẢ

```
(200, 2) (200,)
[[87.3  5.5]
 [54.3  3.6]
 [33.7 10.4]]
[4.48 3.  1.28]
```

Ý tưởng của mô hình: mỗi đặc trưng đóng góp vào giá một lượng tỉ lệ với nó, rồi cộng tất cả lại. Viết thành công thức, với mỗi mẫu có d đặc trưng $\mathbf{x} = (x_1, \dots, x_d)$ (ở ví dụ này $d = 2$: diện tích và khoảng cách), mô hình dự đoán

$$\hat{y} = w_1x_1 + w_2x_2 + \dots + w_dx_d + b = \mathbf{w}^\top \mathbf{x} + b.$$

$\mathbf{w}$ là vector trọng số (weights), b là hệ số tự do (bias, intercept). b là giá trị dự đoán khi mọi đặc trưng bằng 0, và cũng được học như trọng số; model có d trọng số cộng 1 bias, tức $d + 1$ tham số. Một nơ-ron ở chương 3 cũng tính đúng tổng có trọng số này: đầu vào 2 và 3, trọng số 1 và 2, bias 0 cho $2 \cdot 1 + 3 \cdot 2 = 8$.

Đọc w_j như sau: nếu x_j tăng thêm một đơn vị **trong khi mọi đặc trưng khác giữ nguyên**, thì dự đoán tăng thêm w_j . Điều kiện “giữ nguyên” rất quan trọng; cuối bài sẽ quay lại điểm này.

Khi tính cho cả n mẫu một lúc, ta xếp các mẫu thành các hàng của ma trận X , ở đây là 200 hàng, 2 cột:

$$\hat{\mathbf{y}} = X\mathbf{w} + b.$$

$X\mathbf{w}$ là một vector, còn b là một số: phép cộng ở đây là cộng b vào từng phần tử, vì căn nào cũng được cộng cùng một khoản b . NumPy cũng tự làm đúng như vậy.

Trước khi viết code, nên ghi rõ shape của từng đại lượng. Trong C++ kích thước mảng được khai báo tường minh; NumPy thì không yêu cầu, nên người viết phải tự kiểm soát.

Ký hiệu	Ý nghĩa	Shape
X	Ma trận đặc trưng, mỗi hàng một mẫu	(n, d)
$\mathbf{w}$	Trọng số	$(d,)$
b	Bias	số thực
$\hat{y} = X\mathbf{w} + b$	Dự đoán	$(n,)$
y	Giá trị thật	$(n,)$

Cách viết $(d,)$ là NumPy đang nói “mảng một chiều có d phần tử”. Cả khóa này ta thống nhất: vector là mảng một chiều $(n,)$, không dùng mảng cột $(n, 1)$. Lý do được giải thích ở phần bẫy cài đặt.

HỎI NHANH

X có shape $(200, 2)$, $\mathbf{w}$ có shape $(2,)$. $X @ \mathbf{w} + b$ có shape gì?

A $(200,)$

B $(2,)$

C $(200, 2)$

D $(200, 1)$

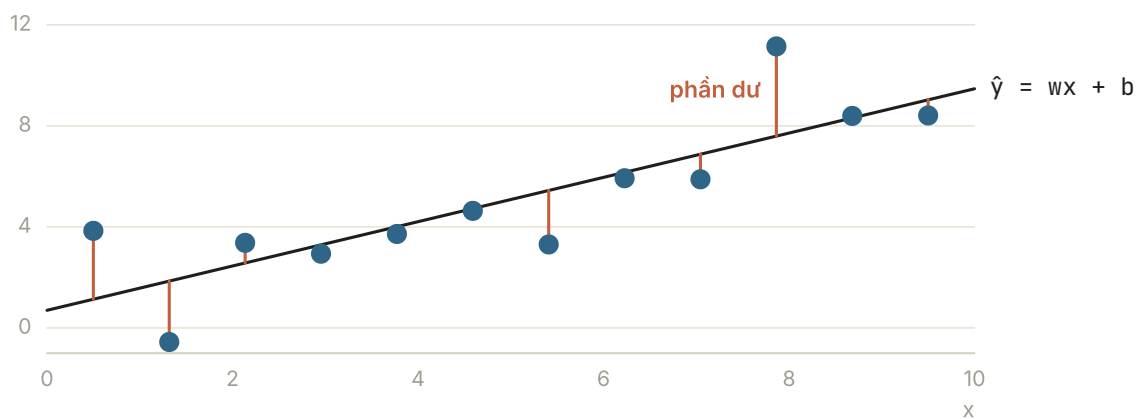
Hàm mất mát

Muốn chọn $\mathbf{w}$ và b “tốt nhất” thì trước hết phải nói rõ thế nào là tốt. Với mỗi căn, ta đo độ lệch giữa dự đoán và giá thật, gọi là phần dư (residual) $r_i = \hat{y}_i - y_i$. Rồi lấy trung bình bình phương các phần dư, gọi là sai số bình phương trung bình (mean squared error):

$$J(\mathbf{w}, b) = \frac{1}{n} \sum_{i=1}^n (\mathbf{w}^\top \mathbf{x}_i + b - y_i)^2.$$

J là **loss** của bài này: một con số đo dự đoán đang lệch đáp án bao nhiêu; học là tìm w, b làm loss nhỏ nhất. Bài hồi quy đo độ lệch giữa hai con số, như MSE; bài phân loại đo xác suất model dành cho nhãn đúng, như cross-entropy ở bài 2.3.

Vì sao bình phương mà không dùng trị tuyệt đối? Trị tuyệt đối cũng dùng được, và hay được chọn khi dữ liệu có nhiều điểm bất thường. Nhưng bình phương có hai ưu điểm. Nó trơn, có đạo hàm tại mọi điểm, nên có công thức nghiệm và dễ tối ưu bằng gradient. Nó phạt lỗi lớn nặng hơn lỗi nhỏ: sai 10 bị phạt 100, gấp bốn lần sai 5. Cái giá là MSE rất nhạy với điểm ngoại lai (outlier): một căn bị nhập nhầm giá có thể kéo lệch cả đường thẳng.



Phần dư là đoạn thẳng đứng từ điểm dữ liệu tới đường dự đoán.

Viết hai hàm nhỏ, rồi thử mô hình đơn giản nhất có thể: đoán mọi căn đều có giá bằng giá trung bình.

Python

```
def predict(X, w, b):  
    return X @ w + b  
  
def mse(y_hat, y):  
    return np.mean((y_hat - y) ** 2)  
  
print(mse(np.full(n, y.mean()), y)) # đoán mọi căn đều có giá trung bình  
print(y.var())
```

KẾT QUẢ

```
2.34147721596206  
2.34147721596206
```

Hai con số bằng nhau, và đây không phải trùng hợp: phương sai được định nghĩa đúng là trung bình bình phương độ lệch so với giá trị trung bình, tức là MSE của mô hình đoán trung bình. Con số này là mốc so sánh: mô hình nào có MSE không thấp hơn mốc này thì chưa học được gì. Người ta đo mức thắng bằng hệ số xác định $R^2 = 1 - \text{MSE} / \text{Var}(y)$: bằng 0 là ngang với đoán theo trung bình, bằng 1 là đoán trúng hết.

HỎI NHANH

Đổi đơn vị giá từ tỷ đồng sang triệu đồng, tức nhân mọi y với 1000. MSE thay đổi thế nào?

- A Không đổi
- B Tăng 1000 lần
- C Tăng 10^6 lần
- D Giảm 1000 lần

Nghiệm đóng

Với MSE, bài toán tìm $\mathbf{w}$ và b có lời giải chính xác bằng công thức, gọi là nghiệm đóng. Trước tiên là một kỹ thuật sẽ dùng lại nhiều lần: thêm vào X một cột toàn số 1. Gọi ma trận mới là $\tilde{X} = [X \ \mathbf{1}]$ (shape $(n, d+1)$) và gộp $\boldsymbol{\theta} = (\mathbf{w}, b)$; khi đó $\hat{\mathbf{y}} = \tilde{X}\boldsymbol{\theta}$, và b chỉ còn là một trọng số như mọi trọng số khác, nhân với cột số 1.

J là hàm bậc hai lồi theo $\boldsymbol{\theta}$, hình dung như một cái bát. Gradient, ký hiệu ∇J , là vector gồm đạo hàm của J theo từng tham số; nó chỉ hướng mà J tăng nhanh nhất. Đáy bát là nơi gradient bằng 0. Nếu chưa quen đạo hàm theo vector, hãy nghĩ như với một biến: $(a\theta - y)^2$ có đạo hàm $2a(a\theta - y)$. Bản ma trận dưới đây chỉ là thay a bằng $\tilde{X}$ và thêm chuyển vị cho khớp chiều:

$$\nabla J(\boldsymbol{\theta}) = \frac{2}{n} \tilde{X}^\top (\tilde{X}\boldsymbol{\theta} - \mathbf{y}) = \mathbf{0} \iff \tilde{X}^\top \tilde{X} \boldsymbol{\theta} = \tilde{X}^\top \mathbf{y}.$$

Đây là hệ phương trình chuẩn (normal equation), gồm $d + 1$ phương trình với $d + 1$ ẩn. Nếu $\tilde{X}^\top \tilde{X}$ khả nghịch thì nghiệm là $\theta^* = (\tilde{X}^\top \tilde{X})^{-1} \tilde{X}^\top \mathbf{y}$. Code dưới giải hệ này bằng `np.linalg.solve`, không tính nghịch đảo. (Trong thực tế, `np.linalg.lstsq(Xb, y, rcond=None)` giải thẳng bài least squares và ổn định hơn về số.)

Python

```
Xb = np.column_stack([X, np.ones(n)])          # thêm cột 1, shape (200, 3)
theta = np.linalg.solve(Xb.T @ Xb, Xb.T @ y)    # giải hệ phương trình chuẩn
w_cf, b_cf = theta[:2], theta[2]
print(w_cf.round(4), round(b_cf, 4))
print(mse(predict(X, w_cf, b_cf), y))
print(predict(np.array([70, 8]), w_cf, b_cf))    # căn hộ ở đầu bài
```

KẾT QUẢ

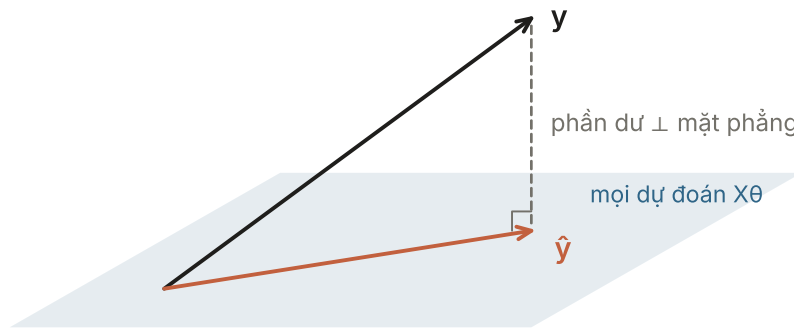
```
[ 0.0499 -0.1453] 0.9809
0.0880761518040597
3.309815169606735
```

Ba con số đầu, 0.0499, -0.1453 và 0.9809, rất gần với hệ số (0.05, -0.15 , 1) trong công thức sinh dữ liệu. Mô hình đã khôi phục gần đúng công thức đó chỉ từ 200 ví dụ.

MSE còn lại là 0.088, xấp xỉ phương sai của nhiễu ($0.3^2 = 0.09$): phần còn lại chính là nhiễu, không model nào đoán được (bài 1.2).

Còn căn hộ ở đầu bài? Model định giá khoảng 3,3 tỷ. Giá rao 4,2 tỷ cao hơn 0,9 tỷ, gấp ba lần độ lệch chuẩn của nhiễu, nên khó coi là giá rẻ.

Phương trình chuẩn có một cách hiểu hình học. Coi $\mathbf{y}$ là một vector trong không gian n chiều; các cột của $\tilde{X}$ sinh ra một “mặt phẳng” trong không gian đó, và mọi dự đoán $\tilde{X}\theta$ có thể có đều nằm trên mặt phẳng ấy. Least squares chọn đúng cái bóng của $\mathbf{y}$ khi chiếu vuông góc xuống. Khó hình dung n chiều thì cứ nghĩ $n = 3$: $\mathbf{y}$ là một mũi tên trong phòng, các dự đoán nằm trên mặt sàn, và nghiệm là bóng của mũi tên khi đèn chiếu thẳng từ trên xuống. Phương trình chuẩn thật ra chỉ nói một điều: phần dư vuông góc với mọi cột. Vì một trong các cột là cột số 1, nên tổng các phần dư luôn bằng 0. Chỗ đoán cao và chỗ đoán thấp luôn bù trừ nhau.



Least squares chiếu y vuông góc xuống mặt phẳng các dự đoán.

Với một đặc trưng, hệ phương trình chuẩn rút gọn thành công thức tính được trên giấy:

$$w = \frac{\sum (x_i - \bar{x})(y_i - \bar{y})}{\sum (x_i - \bar{x})^2}, \quad b = \bar{y} - w\bar{x}.$$

Lấy ba điểm $(1, 2), (2, 3), (3, 5)$: $\bar{x} = 2, \bar{y} = 10/3$. Tử số bằng $(-1)(-\frac{4}{3}) + 0 + (1)(\frac{5}{3}) = 3$, mẫu số bằng 2. Vậy $w = 1.5, b = \frac{10}{3} - 3 = \frac{1}{3}$, đường hồi quy là $\hat{y} = 1.5x + \frac{1}{3}$ với $J = 0.056$.

Nghiệm đóng không phải lúc nào cũng dùng được. Khi có đặc trưng là tổ hợp tuyến tính của các đặc trưng khác (đa cộng tuyến, multicollinearity), hoặc khi số đặc trưng nhiều hơn số mẫu, $\tilde{X}^\top \tilde{X}$ không khả nghịch và bài toán có vô số nghiệm. `np.linalg.lstsq` hay `np.linalg.pinv` vẫn trả về một nghiệm (trong vô số nghiệm, nó chọn vector θ ngắn nhất), còn cách chữa tận gốc là regularization ở bài 2.2. Ngoài ra, chi phí của nó là $O(nd^2 + d^3)$: vài trăm đặc trưng thì tức thì, vài chục nghìn đặc trưng thì quá chậm.

HỎI NHANH

Thêm vào X một cột “diện tích tính bằng dm^2 ”, đúng bằng 100 lần cột diện tích. Nghiệm đóng sẽ ra sao?

- A Chính xác hơn, vì có thêm một cột thông tin
- B Không đổi gì, cột mới chỉ nhận hệ số 0
- C Ma trận suy biến, hệ số có thể rất lớn và trái dấu
- D Chỉ chạy chậm hơn một chút vì thêm một cột

Gradient descent

Nghiệm đóng chỉ có với MSE, và phải giải một hệ $(d + 1) \times (d + 1)$. Sang các bài sau, loss phức tạp hơn, dữ liệu lớn hơn, sẽ không còn công thức nào để giải ngay nữa. Khi đó cần một phương pháp tổng quát: gradient descent, thuật toán dùng để huấn luyện mọi mạng neural.

Hình dung một người đứng trên sườn núi trong sương mù, không thấy đáy thung lũng ở đâu, chỉ cảm nhận được độ dốc dưới chân. Cách hợp lý là bước một bước nhỏ về phía dốc xuống, cảm nhận lại, rồi bước tiếp. Gradient chỉ hướng dốc lên nhanh nhất, nên ta đi ngược nó.

Ở đây tách riêng b để thấy rõ từng thành phần. Đặt $r_i = \mathbf{w}^\top \mathbf{x}_i + b - y_i$; theo quy tắc đạo hàm hàm hợp (chain rule) $\partial(r_i^2)/\partial \mathbf{w} = 2r_i \mathbf{x}_i$, cộng lại rồi chia n :

$$\frac{\partial J}{\partial \mathbf{w}} = \frac{2}{n} \sum_{i=1}^n r_i \mathbf{x}_i = \frac{2}{n} X^\top \mathbf{r}, \quad \frac{\partial J}{\partial b} = \frac{2}{n} \sum_{i=1}^n r_i.$$

Mỗi bước, ta cập nhật với tốc độ học (learning rate) η :

$$\mathbf{w} \leftarrow \mathbf{w} - \eta \frac{\partial J}{\partial \mathbf{w}}, \quad b \leftarrow b - \eta \frac{\partial J}{\partial b}.$$

Đọc công thức thành lời: phần dư r được “chia” về cho từng trọng số, đặc trưng nào góp phần gây lỗi nhiều thì trọng số của nó bị chỉnh nhiều. Khi đoán cao, r dương, gradient của b dương, b bị trừ bớt và dự đoán hạ xuống. Ví dụ với một tham số: $w = 2$, gradient bằng 4, $\eta = 0.1$ thì w mới $= 2 - 0.1 \cdot 4 = 1.6$. Mỗi bước ở đây dùng cả tập dữ liệu, nên thứ tự các mẫu không ảnh hưởng gì; thứ tự chỉ quan trọng khi mỗi bước chỉ dùng một phần dữ liệu (bài 3.5).

Trong code, ta gộp $\mathbf{w}$ và b vào một mảng theta (bias ở cuối, cùng thứ tự với nghiệm đúng) cho các hàm gọn hơn. Mỗi vòng lặp dùng toàn bộ dữ liệu để tính gradient một lần, và người ta gọi một vòng như thế là một epoch:

Python

```
def loss(theta, X, y):
    # theta = (w_1, ..., w_d, b)
    return mse(predict(X, theta[:-1], theta[-1]), y)

def grad(theta, X, y):
    weights = theta[:-1]
    bias = theta[-1]
    residual = predict(X, weights, bias) - y # (n,)
    grad_weights = 2 * (X.T @ residual) / len(y) # (d,)
    grad_bias = 2 * residual.mean() # một số
    return np.append(grad_weights, grad_bias)

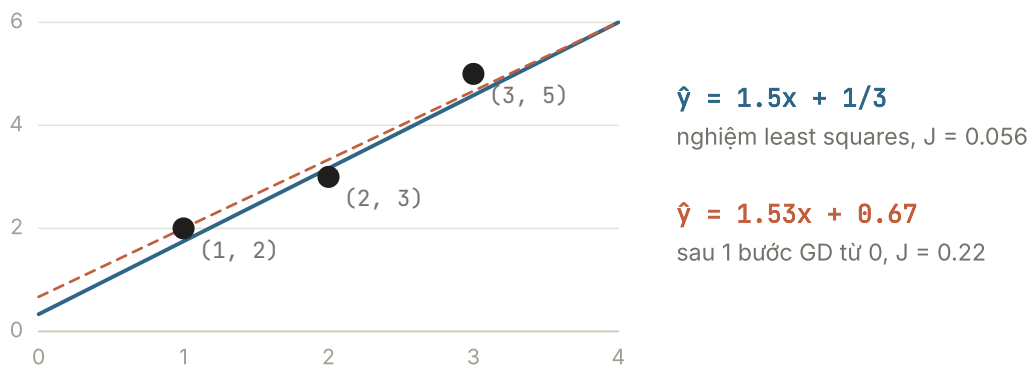
def gd(X, y, lr, epochs):
    theta = np.zeros(X.shape[1] + 1)
    history = []
    for _ in range(epochs):
        history.append(loss(theta, X, y))
        theta -= lr * grad(theta, X, y)
    return theta, history
```

Trước khi chạy, lưu ý rằng grad là hàm dễ sai nhất trong ba hàm trên: nhầm dấu, quên hệ số 2, gõ $X @ r$ thay vì $X.T @ r$. Đáng ngại là viết sai mà code vẫn chạy, loss vẫn có thể giảm, chỉ là giảm về sai chỗ. Mục kiểm tra gradient bên dưới trình bày cách phát hiện các lỗi này.

Thử một bước trên ba điểm (1, 2), (2, 3), (3, 5) của mục trước, xuất phát từ $w = b = 0$ với $\eta = 0.1$. Lúc này $\hat{y} = 0$, nên $r = (-2, -3, -5)$ và

$$\frac{\partial J}{\partial w} = \frac{2}{3} \sum r_i x_i = -\frac{46}{3}, \quad \frac{\partial J}{\partial b} = \frac{2}{3} \sum r_i = -\frac{20}{3}.$$

Sau một bước: $w = 1.53$, $b = 0.67$, mất mát giảm từ 12.67 xuống 0.22, đã gần nghiệm đúng $w = 1.5$, $b = \frac{1}{3}$. Một bước đã gần tới đích chỉ vì bài toán quá nhỏ và quá đẹp; dữ liệu thật thường cần hàng trăm, hàng nghìn bước.



Ba điểm: nghiệm đúng và đường sau một bước gradient descent.

HỎI NHANH

Đặt $r = \hat{y} - y$. Dòng nào cập nhật bias đúng?

A `b -= lr * 2 * r.mean()`

B `b += lr * 2 * r.mean()`

C `b -= lr * 2 * r.sum()`

D `b -= lr * r`

Kiểm tra gradient

Làm sao biết hàm grad đúng? Cách làm giống viết code trâu để đối chiếu trong lập trình thi đấu. Đạo hàm là giới hạn của tỉ sai phân, nên chỉ cần gọi hàm loss là xấp xỉ được gradient, chậm nhưng khó sai. Với $\mathbf{e}_j$ là vector đơn vị thứ j :

$$\frac{\partial J}{\partial \theta_j} \approx \frac{J(\boldsymbol{\theta} + h\mathbf{e}_j) - J(\boldsymbol{\theta} - h\mathbf{e}_j)}{2h}.$$

Dịch về cả hai phía như thế (sai phân trung tâm, central difference) cho sai số $O(h^2)$: giảm h đi 10 lần thì sai số giảm 100 lần. Dịch một phía chỉ đạt $O(h)$, giảm 10 lần thì sai số cũng chỉ giảm 10 lần. Chọn h khoảng 10^{-5} là vừa; nhỏ quá thì phép trừ hai số gần bằng nhau lại sinh sai số làm tròn. Hai vector gradient được so bằng sai số tương đối, với $\|\cdot\|$ là độ dài vector:

$$\text{err} = \frac{\|\mathbf{g} - \mathbf{g}_{\text{num}}\|}{\|\mathbf{g}\| + \|\mathbf{g}_{\text{num}}\|}.$$

Sai số cỡ 10^{-7} trở xuống là gradient gần như chắc đúng; cỡ 10^{-2} trở lên thì công thức gradient nhiều khả năng sai.

Python

```
def numerical_grad(f, theta, h=1e-5):
    g = np.zeros_like(theta)
    for j in range(len(theta)):
        e = np.zeros_like(theta)
        e[j] = h
        g[j] = (f(theta + e) - f(theta - e)) / (2 * h)
    return g

def rel_error(a, b):
    return np.linalg.norm(a - b) / (np.linalg.norm(a) + np.linalg.norm(b))

theta0 = rng.normal(size=3)  # một điểm bất kỳ
f = lambda t: loss(t, X, y)
print(rel_error(grad(theta0, X, y), numerical_grad(f, theta0)))

def grad_buggy(theta, X, y):  # quên hệ số 2
    r = predict(X, theta[:-1], theta[-1]) - y
    return np.append(1 / len(y) * X.T @ r, r.mean())

print(rel_error(grad_buggy(theta0, X, y), numerical_grad(f, theta0)))
```

KẾT QUẢ

```
7.159601417180615e-13
0.33333333333335514
```

Bản đúng lệch cỡ 10^{-13} , bản quên hệ số 2 lệch đúng $1/3$. Cách kiểm này tốn $2(d + 1)$ lần tính loss nên chỉ dùng để kiểm trên dữ liệu nhỏ, không dùng để huấn luyện. Với linear regression, bước này có vẻ thừa; nhưng khi tự viết backpropagation ở Chương 3, đây là cách nhanh nhất để biết đạo hàm đúng hay sai.

HỎI NHANH

Vì sao ta dùng sai phân trung tâm $\frac{J(\theta+h)-J(\theta-h)}{2h}$ thay vì $\frac{J(\theta+h)-J(\theta)}{h}$?

- A Vì tính nhanh hơn
- B Vì sai số là $O(h^2)$ thay vì $O(h)$
- C Vì không phải chọn h
- D Vì sai phân một phía có thể ra số âm

Chọn learning rate

Gradient đã được kiểm tra, giờ chạy gradient descent. Câu hỏi đầu tiên là chọn learning rate bao nhiêu. Thử trên dữ liệu gốc với ba giá trị:

Python

```
for lr in [1e-3, 2e-4, 1e-4]:
    theta_gd, hist = gd(X, y, lr, 1000)
    print(f"lr = {lr:g}: loss sau 1000 vòng = {hist[-1]:.4g}")
print(theta_gd.round(3))                # lr = 1e-4
```

KẾT QUẢ

```
lr = 0.001: loss sau 1000 vòng = nan
lr = 0.0002: loss sau 1000 vòng = inf
lr = 0.0001: loss sau 1000 vòng = 0.1541
[ 0.057 -0.107  0.01 ]
```

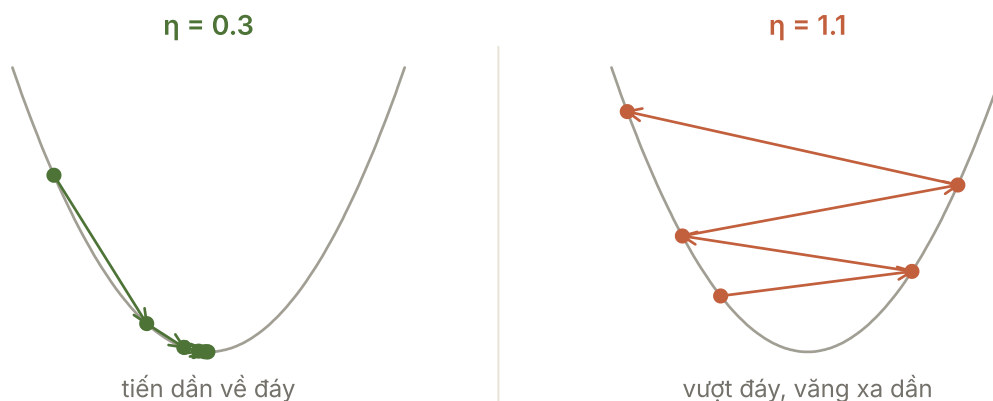
Hai learning rate đầu làm loss nổ tung: inf là vô cùng, còn nan (not a number) là kết quả của những phép như vô cùng trừ vô cùng. Khi chạy trên notebook, NumPy còn in cảnh

báo overflow (tràn số). Cái thứ ba không nổ, nhưng sau 1000 vòng loss vẫn còn 0.154, trong khi đáy là 0.088. Riêng b mới đi được 0.01 trên quãng đường tới 0.98.

Khoảng learning rate dùng được rất hẹp, nhưng có thể tính trước mà không cần thử. J là hàm bậc hai, nên ma trận Hessian (ma trận đạo hàm bậc hai) của nó là hằng số $H = \frac{2}{n} \tilde{X}^\top \tilde{X}$. Gọi sai số là hiệu $\theta - \theta^*$ giữa vị trí hiện tại và đáy. Phân tích sai số theo các vector riêng của H thì thấy: ứng với trị riêng λ , mỗi bước gradient descent nhân sai số với $(1 - \eta\lambda)$. Muốn mọi thành phần đều co lại, cần $|1 - \eta\lambda| < 1$ với mọi λ , tức là

$$\eta < \frac{2}{\lambda_{\max}}.$$

Nếu chưa học trị riêng, có thể hiểu λ là độ cong của cái bát theo một hướng nào đó: hướng nào cong gắt thì bước chân phải ngắn, không thì bước qua đáy rồi văng lên vách bên kia, mỗi lần một xa hơn. Kết luận $\eta < 2/\lambda_{\max}$ dùng được ngay; chứng minh chặt chẽ thuộc về đại số tuyến tính. Với $f(x) = x^2$ thì $\lambda = 2$ nên cần $\eta < 1$: $\eta = 1.5$ nhân x với $1 - 1.5 \cdot 2 = -2$ mỗi bước, từ -1 thành 2 rồi -4 . Loss lúc giảm lúc tăng, dao động mà không hội tụ là dấu hiệu learning rate quá lớn; quá nhỏ thì loss giảm đều nhưng rất chậm.



Bước vừa phải đi dần về đáy; bước quá lớn vượt đáy rồi văng xa dần.

Tính thử cho dữ liệu căn hộ:

Python

```
H = 2 / n * Xb.T @ Xb
eig = np.linalg.eigvalsh(H)          # trị riêng, xếp tăng dần
```

```
print(2 / eig[-1])          # learning rate lớn nhất còn hội tụ
print(eig[-1] / eig[0])    # số điều kiện
```

KẾT QUẢ

```
0.0001434720214166643
99638.12391867978
```

Ngưỡng là $1.43 \cdot 10^{-4}$. Learning rate $2 \cdot 10^{-4}$ vượt ngưỡng nên nổ, 10^{-4} nằm dưới nên hội tụ, khớp đúng với thí nghiệm.

Con số thứ hai là số điều kiện (condition number) $\kappa = \lambda_{\max} / \lambda_{\min} \approx 10^5$, và nó giải thích vì sao hội tụ chậm. Mặt loss lúc này là một thung lũng dài và hẹp: theo chiều này, độ cong lớn gấp khoảng trăm nghìn lần chiều kia. Bước chân phải đủ nhỏ để khỏi văng ra khỏi vách dốc, mà nhỏ như thế thì theo chiều thoải gần như đứng yên tại chỗ.

HỎI NHANH

Linear regression tối ưu MSE, loss qua các vòng là 15.9, 41.2, 260, 17 000, rồi NaN. Shape và công thức gradient đều đã kiểm. Ta nên thử gì trước?

- A Tăng learning rate để vượt qua vùng loss cao
- B Giữ learning rate, chỉ đổi seed vài lần
- C Giảm learning rate, và kiểm thang đo của các feature
- D Tăng số vòng để loss có thời gian ổn định

Chuẩn hóa đặc trưng

Nguyên nhân làm thung lũng hẹp là thang đo. Diện tích tính bằng m^2 , cỡ vài chục đến hơn trăm; khoảng cách cỡ vài km; cột bias thì luôn bằng 1. Ba thang đo chênh nhau quá xa. Cách chữa là chuẩn hóa (standardization) mọi đặc trưng về trung bình 0, độ lệch chuẩn 1.

$$z_j = \frac{x_j - \mu_j}{\sigma_j}.$$

Python

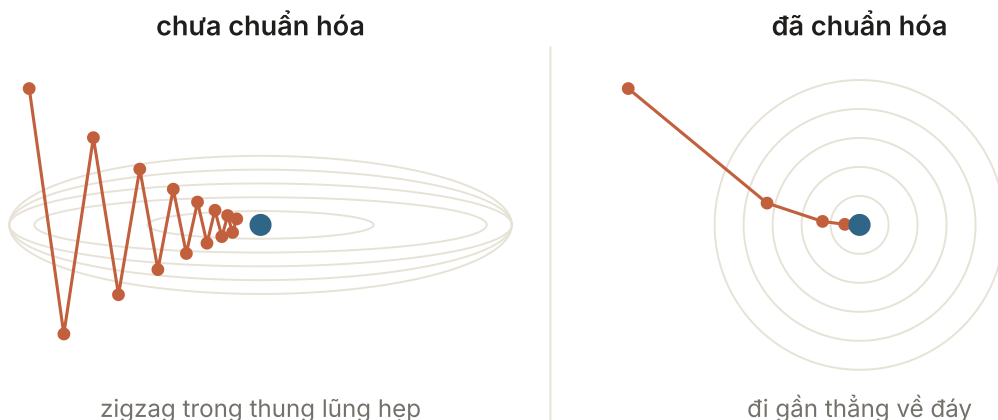
```
mu, sigma = X.mean(axis=0), X.std(axis=0)
Xs = (X - mu) / sigma
Xsb = np.column_stack([Xs, np.ones(n)])
eig_s = np.linalg.eigvalsh(2 / n * Xsb.T @ Xsb)
print(2 / eig_s[-1], eig_s[-1] / eig_s[0])

for lr in [0.01, 0.1, 0.5, 1.0]:
    theta_s, hist = gd(Xs, y, lr, 100)
    print(f"lr = {lr:g}: loss sau 100 vòng = {hist[-1]:.4g}")
```

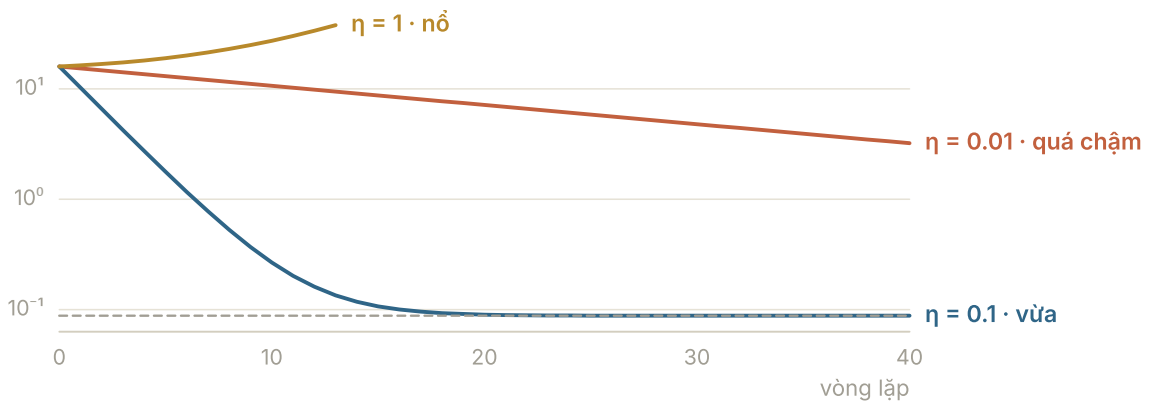
KẾT QUẢ

```
0.951931951010249 1.1063612539062566
lr = 0.01: loss sau 100 vòng = 0.3736
lr = 0.1: loss sau 100 vòng = 0.08808
lr = 0.5: loss sau 100 vòng = 0.08808
lr = 1: loss sau 100 vòng = 3.703e+08
```

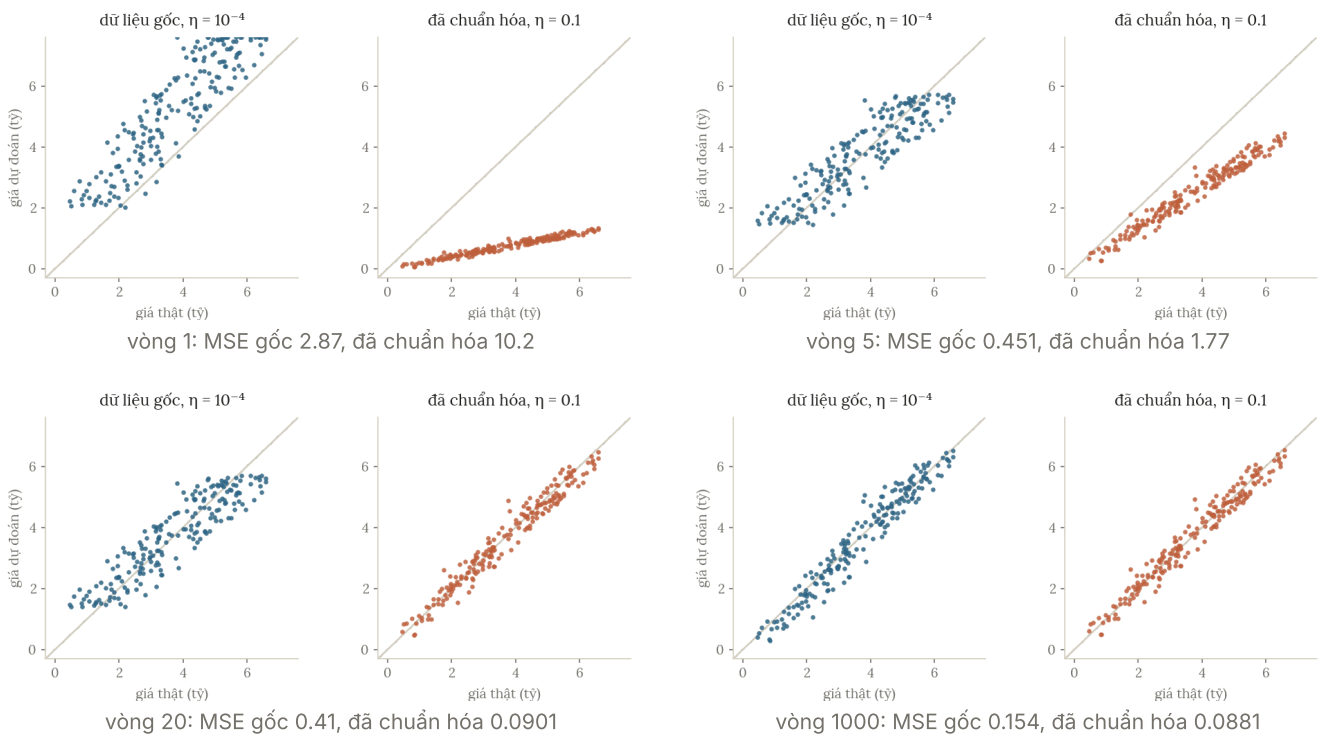
So với dữ liệu gốc: số điều kiện rơi từ 10^5 xuống 1.1, thung lũng hẹp đã thành một cái bát gần tròn, và ngưỡng learning rate tăng từ $1.4 \cdot 10^{-4}$ lên 0.95. Đo cụ thể hơn: để loss xuống sát đáy (trong vòng 0.1%), dữ liệu gốc cần khoảng 180 nghìn vòng dù learning rate đã chọn sát ngưỡng, còn dữ liệu chuẩn hóa với $\eta = 0.1$ chỉ cần 28 vòng. Nhanh hơn khoảng sáu nghìn lần, chỉ nhờ hai dòng code. Còn $\eta = 1$ vượt ngưỡng 0.95 nên vẫn nổ, đúng như lý thuyết.



Chênh thang đo tạo thung lũng hẹp; chuẩn hóa đưa về gần tròn.



Loss theo vòng lặp (thang log) với ba learning rate.



Cùng gradient descent trên 200 căn hộ: dữ liệu gốc tiến nhanh vài vòng đầu rồi gần như đứng yên; dữ liệu đã chuẩn hóa về sát đường chéo sau khoảng 20 vòng.

Dữ liệu đã bị biến đổi, vậy mô hình học được có còn là mô hình định giá nhà ban đầu không? Vẫn là nó, chỉ khác cách viết. Mô hình học trên $\mathbf{z}$ nên trọng số cũng theo thang đo của $\mathbf{z}$. Muốn đọc lại bằng m^2 và km như ban đầu, ta thay $z_j = (x_j - \mu_j)/\sigma_j$ vào:

$$\hat{y} = \sum_j w_j^{(z)} \frac{x_j - \mu_j}{\sigma_j} + b^{(z)} = \sum_j \frac{w_j^{(z)}}{\sigma_j} x_j + \left(b^{(z)} - \sum_j \frac{w_j^{(z)} \mu_j}{\sigma_j} \right),$$

nên trọng số và bias theo đơn vị gốc là

$$w_j = \frac{w_j^{(z)}}{\sigma_j}, \quad b = b^{(z)} - \sum_j \frac{w_j^{(z)} \mu_j}{\sigma_j}.$$

Python

```
theta_s, _ = gd(Xs, y, 0.1, 200)
w_gd = theta_s[:2] / sigma
b_gd = theta_s[2] - np.sum(theta_s[:2] * mu / sigma)
print(w_gd.round(4), round(b_gd, 4))
```

KẾT QUẢ

```
[ 0.0499 -0.1453] 0.9809
```

Khớp với nghiệm đóng đến chữ số thứ tư.

HỎI NHANH

Chuẩn hóa feature rồi học lại cho tới khi MSE chạm đáy. So với mô hình học trên dữ liệu gốc, cái gì thay đổi?

- A Dự đoán chính xác hơn, vì thang đo tốt hơn
- B Trọng số giữ nguyên, chỉ bias đổi theo trung bình
- C Dự đoán y hệt, chỉ cách viết trọng số và tốc độ của gradient descent đổi
- D LinearRegression của scikit-learn cũng hội tụ nhanh hơn

Dùng scikit-learn

Trong thực tế, nhất là trong phòng thi, nên dùng thư viện cho nhanh và chắc. Quy trình chuẩn là chia dữ liệu, học trên tập train, chấm trên tập test:

Python

```
from sklearn.linear_model import LinearRegression
from sklearn.model_selection import train_test_split

X_tr, X_te, y_tr, y_te = train_test_split(X, y, test_size=0.2, random_state=0)
model = LinearRegression().fit(X_tr, y_tr)
print(model.coef_.round(4), round(model.intercept_, 4))
print(model.score(X_tr, y_tr), model.score(X_te, y_te))
```

KẾT QUẢ

```
[ 0.0496 -0.1437] 0.9966
0.9604123189943734 0.9689505848749018
```

`model.score` trả về R^2 , đúng đại lượng ta định nghĩa ở phần hàm mất mát. Điểm trên train (0.960) và trên test (0.969) sát nhau: không có dấu hiệu overfitting.

Điểm test cao hơn train một chút không có gì đặc biệt: tập test chỉ có 40 căn, nên chênh lệch cỡ này phụ thuộc cách chia, đổi `random_state` thì con số cũng đổi. Hệ số cũng hơi khác lúc này, vì lần này mô hình chỉ được học trên 160 căn.

`LinearRegression` giải least squares trực tiếp bằng phân tích ma trận, giống `lstsq`, chứ không dùng gradient descent, nên nó không cần chuẩn hóa mới hội tụ.

Còn code tự viết thì sao? Cho cả ba cách chạy trên cùng dữ liệu rồi so:

Python

```
full = LinearRegression().fit(X, y)
print(np.allclose(full.coef_, w_cf), np.allclose(full.intercept_, b_cf))
print(np.allclose(w_gd, w_cf), np.allclose(b_gd, b_cf))
```

KẾT QUẢ

```
True True
True True
```

Cả bốn kết quả đều True. Lưu ý dùng `np.allclose` chứ không dùng `==`: hai cách tính cùng đúng vẫn có thể lệch nhau ở chữ số thứ mười lăm. Tự cài lại rồi đối chiếu với thư viện giúp hiểu thư viện đang làm gì; khi cần kết quả thì dùng thư viện.

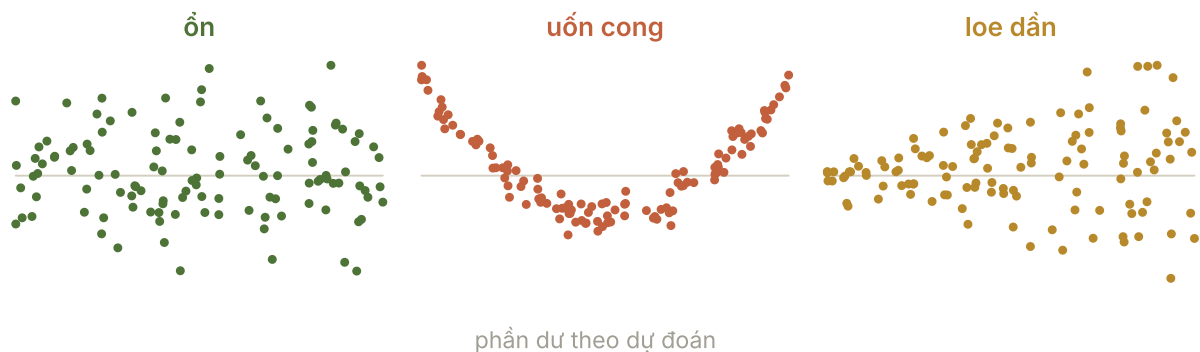
Sau khi huấn luyện, trọng số nằm ở `model.coef_`, bias ở `model.intercept_`: scikit-learn đặt mọi thứ học được vào thuộc tính có dấu gạch dưới ở cuối. Cần đọc chúng đúng cách.

Có ba điểm cần nhớ.

Thứ nhất, hệ số phụ thuộc đơn vị. Hệ số của diện tích (0.05) nhỏ hơn hệ số của khoảng cách (-0.15) về độ lớn, nhưng không có nghĩa là diện tích ít quan trọng hơn, vì một m^2 và một km không so sánh trực tiếp được. Muốn so thì so hệ số sau khi chuẩn hóa.

Thứ hai, hệ số đo mối liên hệ **khi giữ các biến khác cố định**, không phải quan hệ nhân quả. Doanh số kem và số vụ đuối nước cùng tăng vào mùa hè; mô hình sẽ cho hệ số dương, nhưng cấm bán kem không làm giảm số vụ đuối nước.

Thứ ba, luôn vẽ phần dư: trục ngang là dự đoán, trục dọc là phần dư. Mô hình ổn thì các điểm rải đều quanh đường 0, không theo hình thù gì. Phần dư uốn cong là mô hình đang thiếu đặc trưng phi tuyến; phần dư loe dần ra là nên biến đổi y , chẳng hạn lấy log.



Phần dư tốt rải đều quanh 0; uốn cong hay loe dần là có vấn đề.

HỎI NHANH

Linear regression cho hệ số dương giữa số kem bán ra và số vụ đuối nước mỗi tuần. Chỉ từ kết quả này, ta kết luận được gì?

- A Giảm bán kem sẽ giảm số vụ đuối nước đúng theo hệ số
- B Ăn kem làm tăng nguy cơ đuối nước khi bơi
- C Nhiệt độ chắc chắn là nguyên nhân duy nhất của cả hai
- D Hai đại lượng cùng tăng, còn tác động của kem thì chưa biết

Tuyến tính theo tham số

Quan hệ trong thực tế thường không thẳng: giá nhà không tăng đều theo diện tích mái. Nhưng chữ “tuyến tính” trong tên model là tuyến tính theo $\mathbf{w}$, không phải theo $\mathbf{x}$. Ta được tự do biến đổi đặc trưng: thêm cột x^2, x^3 là polynomial regression (hồi quy đa thức); thêm $\log x, \sqrt{x}$ hay tích $x_1 x_2$ cũng được. Tất cả vẫn giải bằng đúng code của bài này. Ví dụ dữ liệu hình chữ U thì đường thẳng khớp kém, thêm cột x^2 là đủ.

Thêm đặc trưng không bao giờ làm lỗi train tăng, vì model luôn có thể đặt trọng số của cột mới bằng 0. Nhưng thêm quá nhiều thì model bắt đầu học thuộc nhiễu (overfitting, bài 1.2), và lỗi validation tăng.

Một ứng dụng cho lập trình thi đấu: đoán độ phức tạp của chương trình trước khi nộp. Nếu thời gian chạy $t \approx Cn^k$ thì lấy log hai vế:

$$\log t = k \log n + \log C.$$

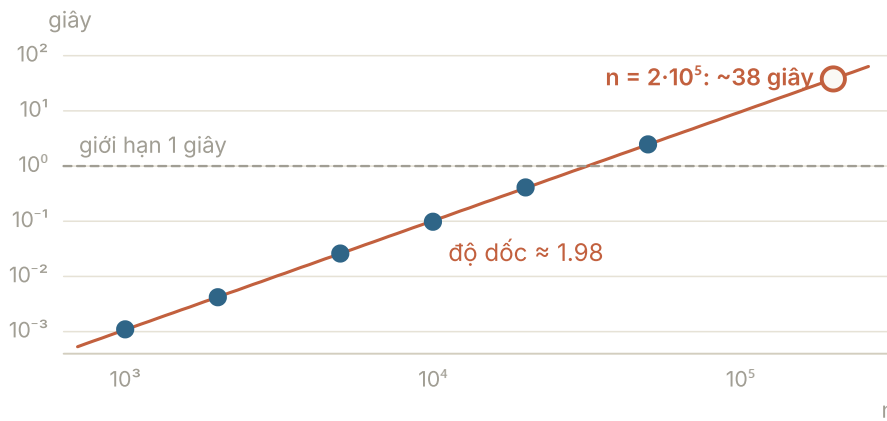
Đó là một đường thẳng theo $\log n$, và hệ số góc chính là số mũ k . Đo thời gian chạy với vài giá trị n , rồi hồi quy. `np.polyfit(x, y, 1)` chính là linear regression với một đặc trưng, trả về hệ số góc và hệ số tự do:

Python

```
ns = np.array([1e3, 2e3, 5e3, 1e4, 2e4, 5e4])
t = np.array([0.0011, 0.0042, 0.026, 0.098, 0.41, 2.47]) # giây, đo trên máy nhà
k, c = np.polyfit(np.log(ns), np.log(t), 1) # least squares bậc 1
print(round(k, 2))
print(np.exp(c) * 2e5 ** k) # dự đoán cho n = 2·105
```

KẾT QUẢ

```
1.98
37.89981222044977
```



Trục log-log: độ dốc gần 2 nghĩa là thời gian tăng theo n^2 .

Hệ số góc xấp xỉ 2: thời gian tăng gần theo n^2 . Ngoại suy cho $n = 2 \cdot 10^5$ được khoảng 38 giây, quá xa giới hạn 1 giây, nên cần thuật toán tốt hơn trước khi nộp. (Vài phép đo không chứng minh được độ phức tạp, nhưng đủ để cảnh báo sớm.)

HỎI NHANH

Sau khi tạo thêm feature, mô hình nào **không** giải được bằng code linear regression của bài này?

A $\hat{y} = w_1 x + w_2 x^2 + b$

B $\hat{y} = w_1 \log x + b$

C $\hat{y} = w_1 x_1 x_2 + w_2 x_1 + b$

D $\hat{y} = e^{wx} + b$

Khi đi thi

- Gặp bài hỏi quy, chạy LinearRegression trước để có mốc so sánh.
- Loss nổ thì chuẩn hóa đặc trưng trước, chỉnh learning rate sau.
- Tự viết gradient thì kiểm bằng sai phân trên dữ liệu nhỏ trước khi chạy thật.

Tóm tắt

- Linear regression dự đoán $\hat{y} = \mathbf{w}^\top \mathbf{x} + b$ và học bằng cách cực tiểu MSE. Mô hình đoán trung bình có MSE bằng $\text{Var}(y)$; R^2 cho biết ta thắng mốc đó bao nhiêu.
- Nghiệm đóng: thêm cột 1 rồi giải hệ phương trình chuẩn bằng `solve`. Về hình học, đó là phép chiếu vuông góc của $\mathbf{y}$.
- Gradient descent đi từng bước ngược gradient và hội tụ khi $\eta < 2/\lambda_{\max}$. Số điều kiện quyết định nhanh hay chậm, và chuẩn hóa đặc trưng kéo nó về gần 1.
- Trong thực tế, dùng scikit-learn với `train/test`; đọc hệ số như mối liên hệ chứ không như nhân quả; giữ vector ở shape $(n,)$.
- “Tuyến tính” là theo tham số, nên đặc trưng biến đổi tùy ý, kể cả $\log n$ để đoán độ phức tạp.

Kiểm tra cuối bài

Mười câu dưới đây bao quát cả bài. Làm hết rồi xem điểm; câu nào sai thì đọc lại mục tương ứng.

CÂU 1

Hệ phương trình chuẩn của một bộ dữ liệu có số điều kiện cỡ 10^{10} . Vì sao điều này đáng lo khi giải bằng máy?

- A Ma trận chắc chắn suy biến, nên hệ vô nghiệm
- B Định thức cỡ 10^{10} nên phép tính bị tràn số
- C Ma trận có quá nhiều phần tử khác 0 nên chạy chậm
- D Sai số nhỏ ở dữ liệu có thể thành sai số lớn ở nghiệm

ĐỀ THI NOAI 2026 Singapore · Vòng sơ loại · Câu 35

CÂU 2

Trên cùng một tập, MSE của mô hình bằng 1 và variance của y bằng 4, cả hai đều chia cho số mẫu n . y không phải hằng số. R^2 bằng bao nhiêu?

A 3

B -3

C 0.75

D 0.25

CÂU 3

Với $L(\theta) = N^{-1} \sum_{n=0}^{N-1} (y_n - \theta^\top x_n)^2$, gradient theo θ là gì?

A $\frac{2}{N} \sum (\theta^\top x_n - y_n)$

B $\frac{2}{N} \sum (\theta^\top x_n - y_n) x_n$

C $\frac{2}{N} \sum (y_n - \theta^\top x_n) x_n$

D $\frac{1}{N} \sum (\theta^\top x_n - y_n) x_n$

ĐỀ THI USA-NA-AIO 2026 · Vòng 1 · Part 4.2

CÂU 4

Cho $J(w) = (w - 3)^2$, xuất phát từ $w = 0$ với $\eta = 0.25$. Sau một bước gradient descent, w bằng bao nhiêu?

A 1.5

B 0.75

C 6

D -1.5

CÂU 5

Vẫn với $J(w) = (w - 3)^2$. Learning rate nào làm gradient descent phân kỳ?

A 0.1

B 0.5

C 0.9

D 1.2

CÂU 6

Công thức nghiệm $\theta^* = (\tilde{X}^\top \tilde{X})^{-1} \tilde{X}^\top \mathbf{y}$ dùng được khi nào?

- A Luôn luôn, với mọi ma trận dữ liệu
- B Khi $\tilde{X}$ vuông, tức số mẫu bằng số cột
- C Khi các cột của $\tilde{X}$ độc lập tuyến tính
- D Khi $\tilde{X}$ đối xứng qua đường chéo

ĐỀ THI NOAI 2026 Singapore · Vòng sơ loại · Câu 34

CÂU 7

Một feature có độ lệch chuẩn $\sigma = 10$. Học trên dữ liệu đã chuẩn hóa, nó nhận trọng số $w^{(z)} = 0.5$. Theo đơn vị gốc, trọng số của nó là bao nhiêu?

- A 0.05
- B 5
- C 0.5
- D 25

CÂU 8

Ta đọc `y = df[["price"]].to_numpy()`, được shape `(100, 1)`, còn `y_hat` có shape `(100,)`. Dòng nào tính đúng MSE?

A `np.mean((y_hat - y.ravel()) ** 2)`

B `np.mean((y_hat - y) ** 2)`

C `np.mean((y_hat - y.T) ** 2)`

D `np.mean(y_hat - y.ravel()) ** 2`

CÂU 9

Gradient do hàm grad tự viết lệch tương đối 0.5 so với gradient tính bằng sai phân. Ta nên làm gì?

A Coi như đúng, vì sai phân chỉ là xấp xỉ

B Tìm lỗi trong grad trước khi chạy tiếp

C Giảm h xuống 10^{-12} rồi kiểm lại

D Tăng learning rate để bù phần gradient thiếu

CÂU 10

Mô hình định giá nhà có trọng số của diện tích là 0.05 tỷ đồng mỗi m^2 . Hai căn hộ giống nhau ở mọi feature khác, chỉ chênh nhau 10 m^2 . Giá dự đoán chênh nhau bao nhiêu?

- A 0.05 tỷ
- B 0.5 tỷ
- C 5 tỷ
- D Chưa biết, vì còn phụ thuộc khoảng cách

Đọc thêm

- James, Witten, Hastie, Tibshirani — [An Introduction to Statistical Learning](#), chương 3. Linear regression từ góc nhìn thống kê: độ tin cậy của hệ số, tương tác giữa các đặc trưng, các vấn đề thường gặp. Sách miễn phí, có bản dùng Python.
- Géron — *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow*, chương 4 “Training Models”. Nghiệm đóng, các biến thể của gradient descent, polynomial regression và learning curve, viết rất thực hành.
- Zhang và cộng sự — [Dive into Deep Learning](#), chương 3. Linear regression được trình bày như một mạng neural một lớp, cài từ đầu rồi bằng PyTorch; cầu nối sang Chương 3.
- Andrew Ng — [CS229 Lecture Notes](#), chương 1. Least squares, phương trình chuẩn, và lý do xác suất đứng sau việc dùng MSE.
- Boyd, Vandenberghe — [Introduction to Applied Linear Algebra](#), chương 12–13. Least squares bằng đại số tuyến tính với nhiều ví dụ ứng dụng; dành cho ai muốn hiểu sâu phần nghiệm đóng. Sách miễn phí.
- 3Blue1Brown — [Essence of Linear Algebra](#). Loạt video trực quan về vector, ma trận, phép chiếu và trị riêng.
- Gabriel Goh — [Why Momentum Really Works](#), Distill (2017). Phân tích gradient descent trên hàm bậc hai qua trị riêng và số điều kiện, có hình tương tác; đọc sau mục chọn learning rate.
- Stanford CS231n — [Gradient checks](#). Các lưu ý khi kiểm gradient bằng sai phân trong thực tế.
- Tài liệu chính thức: [NumPy — Broadcasting](#), [scikit-learn — Linear Models](#).