Module 4 - Tuần 3 - Tree Review: Toàn bộ về cây

Time-Series Team

Ngày 23 tháng 9 năm 2025

Mục lục

1 Mô hình cây trong đời sống **3**
- 1.1 Thuật toán cây quyết định . . . 4
- 1.2 Sự phát triển các mô hình khác . . . 4
  - 1.2.1 Phương pháp Bagging - Random Forest . . . 5
  - 1.2.2 Các phương pháp Boosting . . . 5

2 Ensemble Learning: Ý tưởng chính **6**
- 2.1 Bagging (Bootstrap Aggregation) . . . 8
- 2.2 Boosting . . . 9
- 2.3 Stacking . . . 10
- 2.4 So sánh Bagging, Boosting và Stacking . . . 12

3 Chi tiết thuật toán cây **12**
- 3.1 Decision Tree . . . 12
  - 3.1.1 Nguyên lý lựa chọn split . . . 12
  - 3.1.2 Nguyên lý hoạt động . . . 12
  - 3.1.3 Đối chiếu bằng code Python . . . 14
  - 3.1.4 Gợi ý cắt tỉa (pruning) . . . 15
- 3.2 Random Forest . . . 17
  - 3.2.1 OOB Error . . . 18
  - 3.2.2 Nguyên lý hoạt động . . . 18
  - 3.2.3 Đối chiếu bằng code Python . . . 19
- 3.3 AdaBoost . . . 21
  - 3.3.1 Nguyên lý . . . 21
  - 3.3.2 Chứng minh công thức α_t . . . 22
  - 3.3.3 Nguyên lý hoạt động . . . 23
  - 3.3.4 Đối chiếu bằng code Python . . . 24
- 3.4 Gradient Boosting . . . 26
  - 3.4.1 Nguyên lý . . . 26
  - 3.4.2 Trường hợp đặc biệt . . . 26
  - 3.4.3 Chứng minh công thức bước lá $\gamma^* = -\frac{\sum g_i}{\sum h_i}$ . . . 27
  - 3.4.4 Nguyên lý hoạt động . . . 27
  - 3.4.5 Đối chiếu bằng code Python . . . 28
- 3.5 XGBoost . . . 31
  - 3.5.1 Nguyên lý tổng quát . . . 31
  - 3.5.2 Mục tiêu và khai triển Taylor bậc hai . . . 31
  - 3.5.3 Chứng minh nghiệm tối ưu từng lá và công thức *gain* . . . 32
  - 3.5.4 Vì sao xuất hiện "Similarity Score"? . . . 32
  - 3.5.5 Nguyên lý hoạt động . . . 34
  - 3.5.6 Đối chiếu bằng code Python . . . 35
- 3.6 LightGBM . . . 39
  - 3.6.1 Nguyên lý . . . 39

3.6.2 Các cải tiến cốt lõi của LightGBM: ví dụ và chứng minh 40
3.6.3 Chứng minh công thức cập nhật tại mỗi lá 42
3.6.4 Ví dụ tính tay (log-loss, $\lambda = 1,\ \gamma = 0$) 43
3.6.5 Đối chiếu bằng code Python 44

4 Bài toàn thực nghiệm: Phân loại ung thư vú **48**

1 Mô hình cây trong đời sống

Sơ đồ luồng có thể được diễn giải như một cây quyết định. Các câu hỏi trong đó đại diện cho các biến độc lập (thường gọi là "đặc trưng" trong học máy), trong khi các nút lá cuối cùng biểu diễn những kết quả có thể xảy ra. Hiện tại, các kết quả đầu ra chỉ là dạng văn bản: *"có thể"*, *"haha chúc may mắn"*, *"có lẽ không"*, *"chắc chắn"*.

Mô hình dự đoán việc lắp tấm pin mặt trời mà ta đề xuất có thể là bài toán phân loại nhị phân, phân loại đa lớp hoặc hồi quy, tuỳ theo cách ta cấu trúc các giá trị đầu ra.

Trong sơ đồ, các câu hỏi đều có câu trả lời nhị phân: *"có"* hoặc *"không"*. Trong các nghiên cứu khác, các biến độc lập có thể là số, và khi đó câu trả lời thường được thể hiện bằng bất đẳng thức. Ví dụ, thực tế hơn so với câu hỏi *"Có nóng khi chạm vào không?"* sẽ là một biến độc lập đo nhiệt độ, với ngưỡng đặt tại "Nhiệt độ hoạt động $\geq 30°C$".

Hình 1: Bài toán cây quyết định trong đời sống.

1.1 Thuật toán cây quyết định

Mặc dù cây quyết định là một mô hình, ta vẫn chưa đề cập đến quá trình học cây cho một tập dữ liệu cụ thể. Hai thuật toán phổ biến để học tham số và đặc trưng trong cây là:

- **CART (Classification and Regression Trees)** tối ưu theo chỉ số Gini.
- **ID3 (Iterative Dichotomiser 3)** tối ưu theo *information gain*.

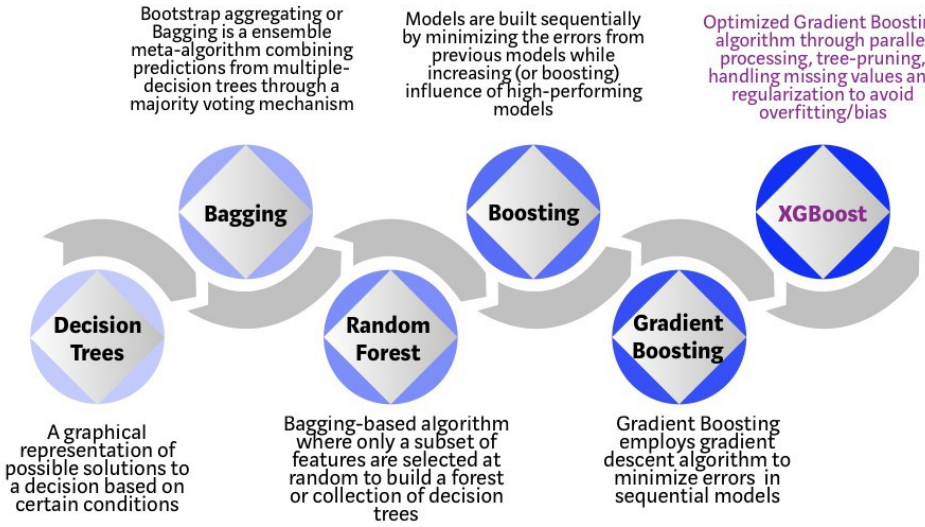
Về mặt khái niệm, các thuật toán huấn luyện cây quyết định hoạt động theo cách tham lam: chọn đặc trưng tốt nhất để phân tách dữ liệu huấn luyện, và lặp lại quá trình này cho từng nhánh. Trong ID3, ta xác định đặc trưng tối ưu bằng *information gain*, tức sự chênh lệch entropy trước và sau khi phân tách. Nguồn: https://geohackweek.github.io/machine-learning/01-tree-based/

Nhược điểm của cây quyết định

- **Dễ bị quá khớp (overfitting)**: nếu cây phát triển quá sâu, mô hình sẽ học cả nhiễu thay vì quy luật chung.
- **Không ổn định**: chỉ cần thay đổi nhỏ trong dữ liệu huấn luyện cũng có thể dẫn tới một cấu trúc cây hoàn toàn khác.
- **Khó khái quát hóa**: hiệu quả dự đoán trên dữ liệu mới thường không cao bằng các mô hình phức tạp hơn.
- **Thiên lệch đối với đặc trưng có nhiều giá trị**: các đặc trưng dạng phân loại với nhiều mức giá trị có thể chiếm ưu thế trong quá trình tách.

1.2 Sự phát triển các mô hình khác

Do những hạn chế trên, các nhà nghiên cứu đã phát triển các phương pháp nâng cao như **Bagging**, **Random Forest**, và **Boosting**. Những mô hình này kết hợp nhiều cây quyết định lại với nhau, giúp giảm thiểu overfitting, tăng độ ổn định, và cải thiện đáng kể hiệu năng dự đoán. Đây chính là bước tiến quan trọng trong sự phát triển của học máy dựa trên cây.



Hình 2: Sự phát triển của thuật toán cây quyết định

1.2.1 Phương pháp Bagging - Random Forest

Một tập hợp lớn các cây thì được gọi là gì? **Một khu rừng!**. Rừng ngẫu nhiên là một tập hợp rất nhiều cây quyết định. Số lượng cây trong mô hình được kiểm soát bằng siêu tham số, thường dao động từ vài trăm đến vài nghìn.

Trong phân loại, kết quả được xác định bằng **mode** của các đầu ra từ tất cả các cây. Trong hồi quy, ta thường lấy **trung bình** các giá trị đầu ra.

Ưu điểm

- Giảm hiện tượng overfitting mà cây đơn lẻ dễ mắc phải.
- Cho độ chính xác dự đoán cao hơn nhiều.
- Mạnh mẽ hơn, ít bị ảnh hưởng bởi nhiễu nhỏ trong tập dữ liệu.

Nhược điểm

- Huấn luyện tốn nhiều tài nguyên tính toán hơn.
- Khó diễn giải hơn so với cây quyết định đơn lẻ.

Sự khác biệt giữa *bagging* và rừng ngẫu nhiên:

- **Bagging**: chọn ngẫu nhiên các mẫu huấn luyện (có hoàn lại), huấn luyện nhiều cây độc lập.
- **Random Forest**: ngoài việc chọn mẫu ngẫu nhiên, còn chọn **tập con đặc trưng** ngẫu nhiên tại mỗi nút tách.

1.2.2 Các phương pháp Boosting

Ngoài rừng ngẫu nhiên, các biến thể khác của mô hình cây đã được phát triển để cải thiện độ chính xác. Khác với bagging (các cây có thể xây dựng song song), boosting xây dựng cây **tuần tự**: mỗi cây mới nhằm sửa lỗi còn lại từ cây trước.

AdaBoost

- Tăng trọng số cho những quan sát bị phân loại sai.
- Ép các cây tiếp theo tập trung hơn vào những điểm khó.

Gradient Boosting

- Giảm lỗi bằng cách mô hình hóa *residuals* (sai số còn lại).
- Giả sử mô hình hiện tại là $F_m(x)$, ta xây thêm một hàm $h(x)$ để dự đoán residuals:

$$F_{m+1}(x) = F_m(x) + h(x).$$

- Quá trình này tiếp tục cho đến khi đạt số cây m mong muốn.

XGBoost – Chuẩn mực vàng

Một triển khai nổi tiếng của gradient boosting là **XGBoost**, do Tianqi Chen phát triển khi còn là nghiên cứu sinh tại Đại học Washington. Đóng góp nổi bật:

- Thuật toán nhận biết thưa thớt (sparsity-aware).
- Kỹ thuật *weighted quantile sketch* để chia nhánh xấp xỉ.
- Tối ưu truy cập bộ nhớ đệm, song song hóa, và nhiều cải tiến hiệu năng khác.

Nhờ vậy, XGBoost cực kỳ nhanh, mạnh, và đã trở thành "vũ khí bí mật" trong nhiều cuộc thi Kaggle cũng như ứng dụng thực tế.

LightGBM – Tăng tốc và hiệu quả

LightGBM (Light Gradient Boosting Machine) là một thuật toán boosting được Microsoft phát triển năm 2017, tối ưu hóa hiệu suất và bộ nhớ so với các triển khai gradient boosting truyền thống.

Đặc điểm nổi bật:

- Sử dụng thuật toán *Leaf-wise growth* (phát triển theo lá) thay vì *Level-wise*, giúp giảm lỗi nhanh hơn.
- Hỗ trợ *Histogram-based splitting*, giảm đáng kể chi phí tính toán và bộ nhớ.
- Xử lý tốt dữ liệu lớn và phân tán nhờ khả năng huấn luyện song song, đồng thời hỗ trợ GPU.
- Hiệu quả đặc biệt với tập dữ liệu nhiều đặc trưng (high-dimensional) và quy mô hàng triệu mẫu.

Nhờ những cải tiến này, LightGBM trở thành lựa chọn phổ biến trong các ứng dụng thực tế, nơi vừa cần tốc độ huấn luyện cao, vừa đảm bảo độ chính xác dự đoán.

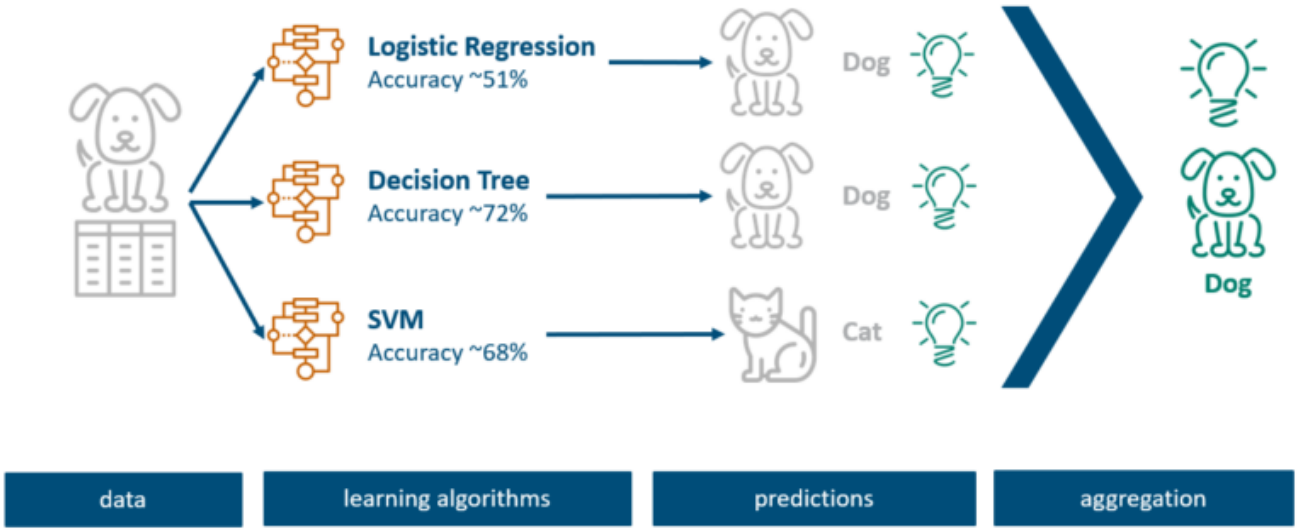
2 Ensemble Learning: Ý tưởng chính

Khái niệm cơ bản

Ý tưởng chính của ensemble learning là kết hợp nhiều mô hình khác nhau để cùng giải quyết một nhiệm vụ. Trong khi mô hình đơn lẻ chỉ dùng một thuật toán, bagging và boosting kết hợp nhiều mô hình yếu để tạo dự đoán ổn định hơn.

Ví dụ: Phân loại ảnh

Giả sử ta có một tập ảnh gán nhãn (chó, mèo, ...). Nếu huấn luyện logistic regression, cây quyết định và SVM, kết quả trên một mẫu có thể khác nhau: Logistic regression và cây quyết định dự đoán là chó, nhưng SVM lại dự đoán là mèo.



Hình 3: Ví dụ trong phân loại hình ảnh

Ensemble learning giải quyết bằng cách kết hợp tất cả các mô hình, thay vì chọn duy nhất một mô hình có độ chính xác cao nhất. Quá trình này gọi là **aggregation** hoặc **voting**.

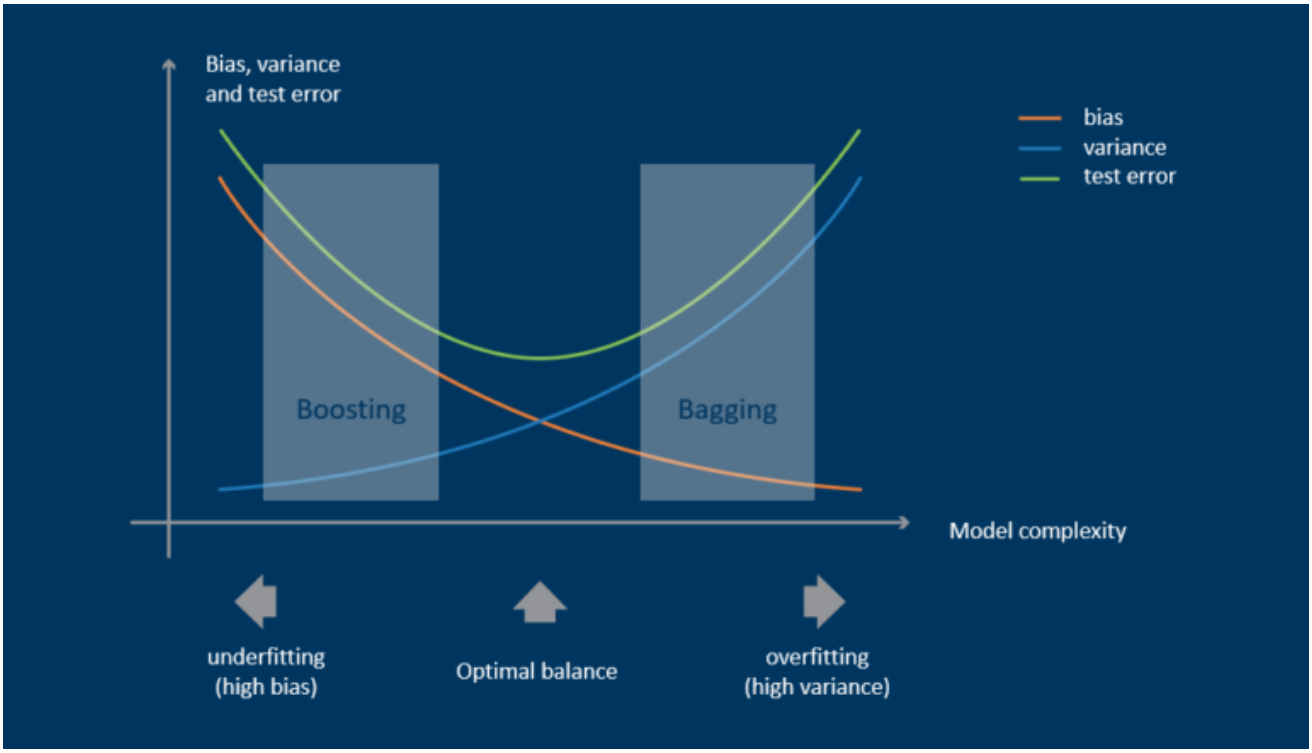
Bias–Variance Trade-off

Mối quan hệ giữa bias và variance:

- Giảm variance → tăng bias.
- Giảm bias → tăng variance.

Mục tiêu là cân bằng để sai số kiểm thử (test error) được tối thiểu.
Ensemble learning giúp cân bằng hai cực đoan này. Trong đó:

- Bagging giảm variance để tránh overfitting.
- Boosting giảm bias để tránh underfitting.



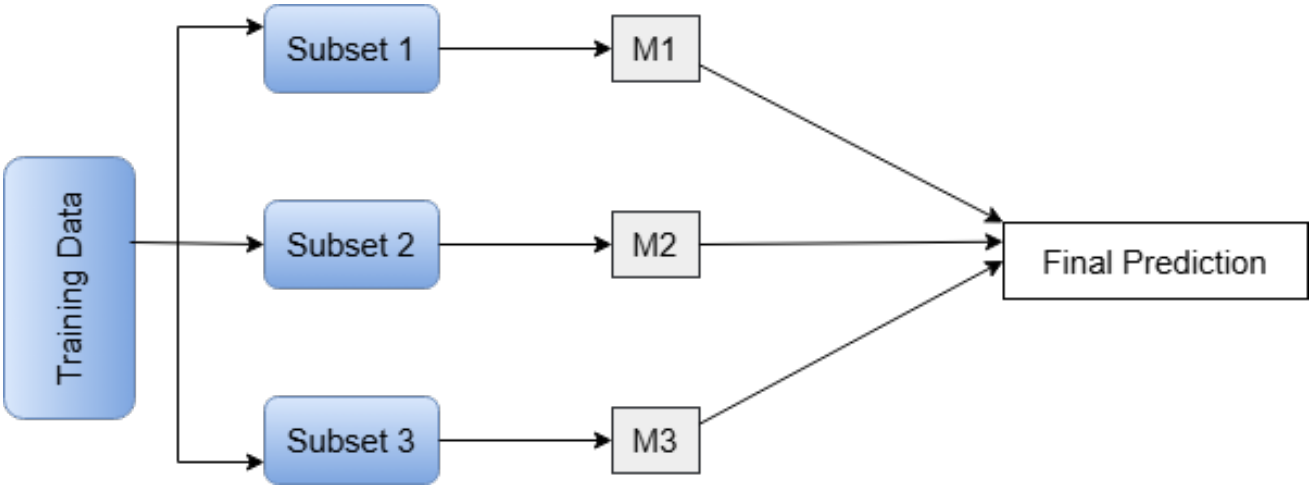
Hình 4: Bias_variance trade-off

2.1 Bagging (Bootstrap Aggregation)

Bagging tạo ra nhiều tập con bằng cách lấy mẫu ngẫu nhiên có hoàn lại từ tập huấn luyện. Mỗi tập con huấn luyện một mô hình riêng biệt. Kết quả cuối cùng là trung bình (hồi quy) hoặc bỏ phiếu đa số (phân loại).

Đặc điểm

- Các mô hình được huấn luyện **song song**.
- Giảm variance, tránh overfitting.
- Tương tự ví dụ phân loại chó–mèo ở trên.



Hình 5: Bagging

Random Forest

Một mở rộng nổi tiếng của bagging là **Random Forest**. Khác với bagging, random forest không chỉ chọn mẫu dữ liệu ngẫu nhiên mà còn chọn ngẫu nhiên tập con đặc trưng ở mỗi bước phân tách.

Ví dụ code nguồn Bagging

```
from sklearn.ensemble import BaggingClassifier
from sklearn.linear_model import LogisticRegression

# base estimator
est = LogisticRegression()

# n_estimators = số lượng mô hình trong ensemble
# max_samples = số mẫu lấy ra để huấn luyện mỗi mô hình
bag_model = BaggingClassifier(base_estimator=est, n_estimators=10, max_samples=1.0)

bag_model = bag_model.fit(X_train, y_train)
Prediction = bag_model.predict(X_test)
```

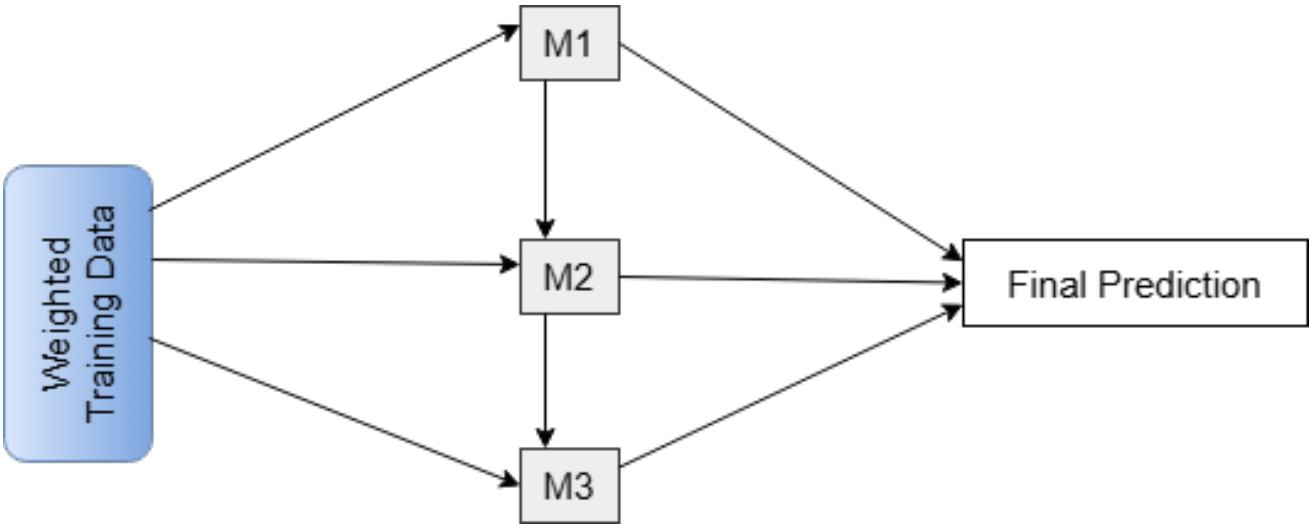
2.2 Boosting

Boosting là một biến thể của bagging nhưng xây dựng **tuần tự**. Trong khi bagging giảm variance, boosting tập trung vào việc giảm bias.

Nguyên lý

1. Huấn luyện mô hình yếu ban đầu.
2. Xác định các mẫu bị phân loại sai.
3. Tăng trọng số cho các mẫu này và huấn luyện mô hình mới.
4. Kết hợp mô hình mới với mô hình trước đó.

Quá trình này lặp lại nhiều lần cho đến khi đạt hiệu năng mong muốn.



Hình 6: Boosting

Đặc điểm

- Mỗi mô hình sau tập trung nhiều hơn vào điểm dữ liệu khó.
- Kết hợp bằng cách gán trọng số khác nhau cho các mô hình.
- Có thể giảm underfitting, nhưng cũng dễ dẫn đến overfitting.

Ví dụ code nguồn Boosting (AdaBoost)

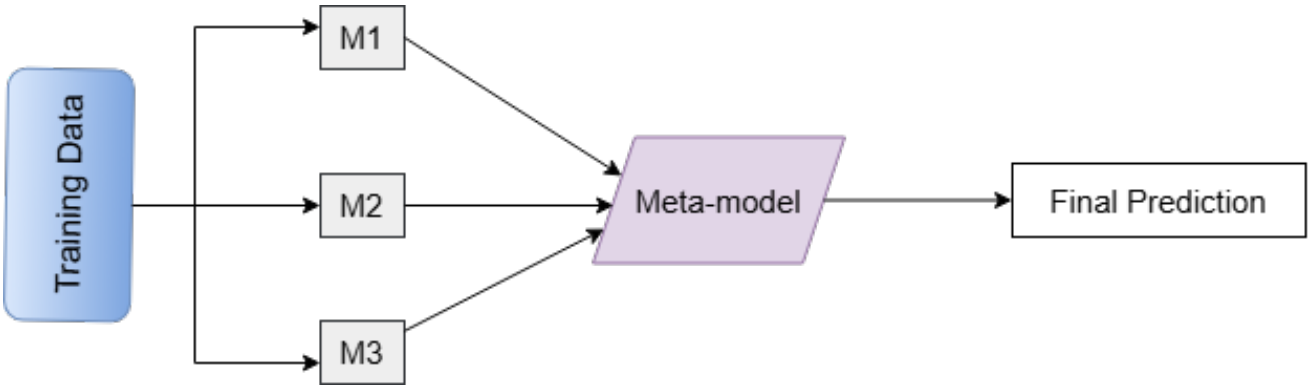
```
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# base estimator
est = DecisionTreeClassifier(max_depth=1)

# n_estimators = số vòng lặp
# learning_rate = tốc độ học
boost_model = AdaBoostClassifier(base_estimator=est, n_estimators=10, learning_rate=1.0)

boost_model = boost_model.fit(X_train, y_train)
Prediction = boost_model.predict(X_test)
```

2.3 Stacking

Stacking là một phương pháp ensemble nâng cao, khác với bagging và boosting ở cách kết hợp các mô hình. Thay vì chỉ *trung bình* hay *bỏ phiếu*, stacking sử dụng một **mô hình meta-learner** để học cách kết hợp tối ưu từ các mô hình con.

Nguyên lý

1. Huấn luyện nhiều mô hình cơ sở (base models) khác nhau như Logistic Regression, Decision Tree, SVM, Random Forest, v.v.
2. Thu thập dự đoán từ các mô hình này trên tập validation để tạo thành một **tập dữ liệu mới**.
3. Huấn luyện một mô hình thứ cấp (meta-model) trên tập dữ liệu này để học cách gán trọng số/ưu tiên giữa các mô hình con.



Hình 7: Stacking

Đặc điểm

- Tận dụng điểm mạnh của nhiều loại mô hình khác nhau, thay vì chỉ lặp lại cùng một thuật toán (như bagging/boosting).
- Meta-learner có thể là bất kỳ mô hình nào (thường là Logistic Regression hoặc XGBoost).
- Thường cho hiệu quả cao trong các bài toán phức tạp (ví dụ: các cuộc thi Kaggle).

Ví dụ minh họa

```
from sklearn.ensemble import StackingClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC

# Các mô hình cơ sở
base_estimators = [
    ('dt', DecisionTreeClassifier(max_depth=3)),
    ('svm', SVC(probability=True))
]

# Mô hình meta
meta_model = LogisticRegression()

stack_model = StackingClassifier(
    estimators=base_estimators,
```

```
    final_estimator=meta_model
)

stack_model.fit(X_train, y_train)
Prediction = stack_model.predict(X_test)
```

2.4 So sánh Bagging, Boosting và Stacking

- **Bagging**: nhiều mô hình cùng loại, huấn luyện song song, giảm variance.
- **Boosting**: nhiều mô hình yếu, huấn luyện tuần tự, giảm bias.
- **Stacking**: nhiều mô hình khác loại, kết hợp bằng meta-model để tối ưu hóa.

3 Chi tiết thuật toán cây

Để minh hoạ, ta sử dụng một tập dữ liệu nhỏ nhị phân (8 mẫu):

ID	A = Hút thuốc	B = Ho khan	Nhãn y (Bệnh=1)
1	1	1	1
2	1	0	1
3	1	1	0
4	1	0	1
5	0	1	1
6	0	1	0
7	0	0	0
8	0	0	0

Tổng cộng: 4 dương, 4 âm.

3.1 Decision Tree

3.1.1 Nguyên lý lựa chọn split

Với bài toán phân loại nhị phân, hai tiêu chuẩn phổ biến:

Entropy & Information Gain (ID3/C4.5).

$$H(S) = -\sum_{c\in\{0,1\}} p_c \log_2 p_c, \qquad \text{Gain}(S,X) = H(S) - \sum_{v\in\text{values}(X)} \frac{|S_v|}{|S|} H(S_v).$$

Gini (CART).

$$\text{Gini}(S) = 1 - \sum_{c\in\{0,1\}} p_c^2.$$

3.1.2 Nguyên lý hoạt động

(1) Tính tay: Entropy gốc

$p_1 = p_0 = 0.5 \Rightarrow$

$$H(S) = -0.5\log_2 0.5 - 0.5\log_2 0.5 = -0.5(-1) - 0.5(-1) = 1 \text{ bit}.$$

(2) Tính tay: So sánh hai split ở gốc

Split theo A. Hai nhánh đều có 4 mẫu:

- $A = 1$: (3 dương, 1 âm) $\Rightarrow p_1 = \frac{3}{4}, p_0 = \frac{1}{4}$.

$$H(S_{A=1}) = -\tfrac{3}{4}\log_2\tfrac{3}{4} - \tfrac{1}{4}\log_2\tfrac{1}{4} = -0.75(-0.415037) - 0.25(-2) \approx 0.311278 + 0.5 = 0.811278.$$

- $A = 0$: đối xứng $(1, 3) \Rightarrow H(S_{A=0}) = 0.811278$.

Entropy sau tách (trọng số $= 4/8 = 0.5$ mỗi nhánh):

$$H_{\text{after}}(A) = 0.5 \times 0.811278 + 0.5 \times 0.811278 = 0.811278.$$

$$\boxed{\text{Gain}(S, A) = 1 - 0.811278 = 0.188722.}$$

Split theo B. Mỗi nhánh có 2 dương và 2 âm $\Rightarrow H = 1$.

$$H_{\text{after}}(B) = 1, \qquad \boxed{\text{Gain}(S, B) = 1 - 1 = 0.}$$

Kết luận: Chọn A làm thuộc tính ở nút gốc.

(3) Tính tay: Split cấp 2 bên trong từng nhánh của A

Xét splitting tiếp theo theo B.

Tại nhánh $A = 1$. Tập con: $\{(1,1,1), (1,0,1), (1,1,0), (1,0,1)\}$ với $H_{\text{trước}} = 0.811278$.

- $B = 0$: $\{(1,0,1), (1,0,1)\} \Rightarrow (2, 0) \Rightarrow H = 0$.
- $B = 1$: $\{(1,1,1), (1,1,0)\} \Rightarrow (1, 1) \Rightarrow H = 1$.

Trong node $A = 1$, trọng số mỗi nhánh bằng $2/4 = 0.5$:

$$H_{\text{sau}} = 0.5 \times 0 + 0.5 \times 1 = 0.5, \quad \boxed{\text{Gain}_{A=1}(B) = 0.811278 - 0.5 = 0.311278.}$$

Tại nhánh $A = 0$. Tập con: $\{(0,1,1), (0,1,0), (0,0,0), (0,0,0)\}$ với $H_{\text{trước}} = 0.811278$.

- $B = 0$: $\{(0,0,0), (0,0,0)\} \Rightarrow (0, 2) \Rightarrow H = 0$.
- $B = 1$: $\{(0,1,1), (0,1,0)\} \Rightarrow (1, 1) \Rightarrow H = 1$.

$$\boxed{\text{Gain}_{A=0}(B) = 0.811278 - 0.5 = 0.311278.}$$

(4) Tính tay theo tiêu chuẩn Gini

Tại gốc: $p_1 = p_0 = 0.5 \Rightarrow \text{Gini}(S) = 1 - (0.5^2 + 0.5^2) = 0.5$.

$$\text{Gini}(A{=}1) = \text{Gini}(A{=}0) = 1 - \left[(\tfrac{3}{4})^2 + (\tfrac{1}{4})^2\right] = 0.375.$$

$$\text{Gini\_after}(A) = 0.5 \times 0.375 + 0.5 \times 0.375 = 0.375 \Rightarrow \boxed{\text{Gini Gain}(A) = 0.5 - 0.375 = 0.125.}$$

Với B: hai nhánh cân bằng $(2, 2) \Rightarrow \text{Gini} = 0.5 \Rightarrow \text{gain} = 0$.

3.1.3 Đối chiếu bằng code Python

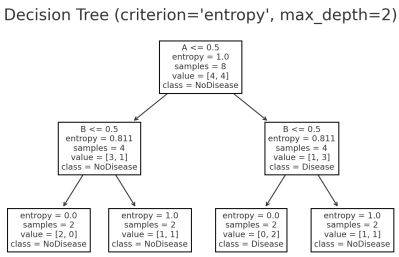
code dưới đây tính lại entropies/gains "bằng tay", huấn luyện `DecisionTreeClassifier` với cả `entropy` và `gini` (giới hạn `max_depth` = 1 để so sánh split ở gốc), và in cây kết quả.

```
import math, pandas as pd
from sklearn.tree import DecisionTreeClassifier, export_text

# Dataset
df = pd.DataFrame([
    (1,1,1),(1,0,1),(1,1,0),(1,0,1),
    (0,1,1),(0,1,0),(0,0,0),(0,0,0)],
    columns=["A","B","y"])

def entropy(counts):
    total = sum(counts);
    return 0.0 if total==0 else \
      sum([-(c/total)*math.log(c/total,2) for c in counts if c>0])

def gini(counts):
    total = sum(counts);
    return 0.0 if total==0 else \
      1.0 - sum([(c/total)**2 for c in counts])

# Root
pos, neg = int((df.y==1).sum()), int((df.y==0).sum())
H_root = entropy([pos,neg])        # -> 1.0
G_root = gini([pos,neg])           # -> 0.5

def gain_for(feature, impurity):
    # impurity: pass entropy or gini function
    after = 0.0
    for v, sub in df.groupby(feature):
        p = int((sub.y==1).sum()); n = int((sub.y==0).sum())
        after += (len(sub)/len(df)) * impurity([p,n])
    before = impurity([pos,neg])
    return before - after, after

gain_A_ent, after_A_ent = gain_for("A", entropy)   # -> 0.1887218755, 0.8112781
gain_B_ent, after_B_ent = gain_for("B", entropy)   # -> 0.0, 1.0
gain_A_gini, after_A_gini = gain_for("A", gini)    # -> 0.125, 0.375
gain_B_gini, after_B_gini = gain_for("B", gini)    # -> 0.0, 0.5

# Train depth-1 trees to confirm split at root
X = df[["A","B"]].values; y = df["y"].values
clf_ent = DecisionTreeClassifier(criterion="entropy", max_depth=1, random_state=0).fit(X,y)
clf_gin = DecisionTreeClassifier(criterion="gini", max_depth=1, random_state=0).fit(X,y)

print(export_text(clf_ent, feature_names=["A","B"]))
```

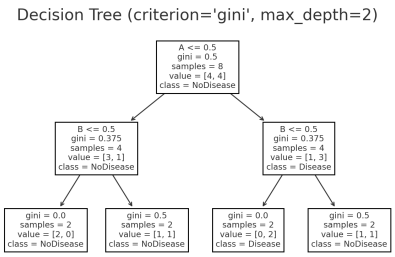
```
print(export_text(clf_gin, feature_names=["A","B"]))
# (giống hệt: split theo A)
```

```
Tính toán theo entropy
|--- A <= 0.50
|   |--- class: 0
|--- A >  0.50
|   |--- class: 1

Tính toán theo gini
|--- A <= 0.50
|   |--- class: 0
|--- A >  0.50
|   |--- class: 1
```

(a) Kết quả từ Python code



(b) Cây với Entropy



(c) Cây với Gini

Hình 8: So sánh các kết quả Decision Tree: code chạy, Entropy và Gini

Kết luận đối chiếu.

- Entropy gốc = 1.0000; sau tách theo $A = 0.811278 \Rightarrow$ Gain = 0.188722 (**khớp** code).
- Gini gốc = 0.5; sau tách theo $A = 0.375 \Rightarrow$ Gini-Gain = 0.125 (**khớp** code).
- Cây độ sâu 1 từ `scikit-learn` đều chọn A ở gốc; chính xác huấn luyện = 75% (sai 2 mẫu: $(A{=}1, y{=}0)$ và $(A{=}0, y{=}1)$).

3.1.4 Gợi ý cắt tỉa (pruning)

Để tránh overfitting: *pre-pruning* (giới hạn `max_depth`, `min_samples_leaf`, `min_impurity_decrease`) hoặc *post-pruning* dạng chi phí–độ phức tạp:

$$\min_{T} \ \mathrm{Loss}_{\mathrm{val}}(T) + \alpha\,|T|,$$

với $|T|$ là số lá, $\alpha > 0$ điều chỉnh mức phạt độ phức tạp.

Ví dụ cắt tỉa cây (Cost–Complexity Pruning)

Định nghĩa. Với một cây T (chỉ quan tâm các *lá*), đặt

$$R(T) = \sum_{t \in \mathcal{L}(T)} p(t)\, I(t), \quad \text{và} \quad R_\alpha(T) = R(T) + \alpha\,|\mathcal{L}(T)|,$$

trong đó $p(t) = \frac{N_t}{N}$ là tỉ lệ mẫu đi vào lá t, $I(t)$ là độ mất mát tại lá (ở đây dùng **entropy**), $|\mathcal{L}(T)|$ là số lá, và $\alpha \geq 0$ là hệ số phạt độ phức tạp.

Ba ứng viên cây cần so sánh. Với tập dữ liệu 8 mẫu, cây "đầy đủ" T_{full} tách gốc theo A, sau đó mỗi nhánh lại tách theo B (4 lá). Hai cây *đơn giản hơn*:

- (i) $T_{\text{1-split}}$: chỉ tách ở gốc theo A (2 lá);
- (ii) T_{root}: không tách gì (1 lá).

Tính entropy từng lá và $R(T)$. *Cây đầy đủ* T_{full} có bốn lá, mỗi lá chứa 2 mẫu ($p(t) = 2/8 = 0.25$):

- A=1, B=0: $(2, 0) \Rightarrow I = 0$;
- A=1, B=1: $(1, 1) \Rightarrow I = 1$;
- A=0, B=0: $(0, 2) \Rightarrow I = 0$;
- A=0, B=1: $(1, 1) \Rightarrow I = 1$.

Suy ra

$$R(T_{\text{full}}) = \sum_t p(t)I(t) = 0.25 \cdot (0 + 1 + 0 + 1) = \boxed{0.5}, \qquad |\mathcal{L}| = 4.$$

Cây một lần tách $T_{\text{1-split}}$ (tách theo A): mỗi lá có 4 mẫu ($p = 0.5$) và phân bố $(3, 1)$ hoặc $(1, 3)$ nên

$$I = -\tfrac{3}{4}\log_2\tfrac{3}{4} - \tfrac{1}{4}\log_2\tfrac{1}{4} = \boxed{0.811278}.$$

Do đó

$$R(T_{\text{1-split}}) = 0.5 \cdot 0.811278 + 0.5 \cdot 0.811278 = \boxed{0.811278}, \qquad |\mathcal{L}| = 2.$$

Cây gốc T_{root}: chỉ một lá chứa toàn bộ 8 mẫu, tỉ lệ $(4, 4)$ nên

$$R(T_{\text{root}}) = H(\tfrac{1}{2}, \tfrac{1}{2}) = \boxed{1.0}, \qquad |\mathcal{L}| = 1.$$

Điểm cắt tỉa yếu nhất (weakest link). Theo lý thuyết cost–complexity, một nút (hoặc một tiểu cây) nên bị cắt khi α vượt quá

$$\alpha^\star = \frac{R(T_{\text{sub}}) - R(T)}{|\mathcal{L}(T)| - |\mathcal{L}(T_{\text{sub}})|}.$$

Với ba mức cây ở trên, ta có hai ngưỡng:

1. So sánh T_{full} (4 lá) với $T_{\text{1-split}}$ (2 lá):

$$\alpha_1 = \frac{0.811278 - 0.5}{4 - 2} = \boxed{0.155639}.$$

   Nghĩa là khi $\alpha > \alpha_1$, chi phí R_α của cây 4 lá vượt $T_{\text{1-split}}$, nên **cắt** về cây 2 lá.

2. So sánh $T_{\text{1-split}}$ (2 lá) với T_{root} (1 lá):

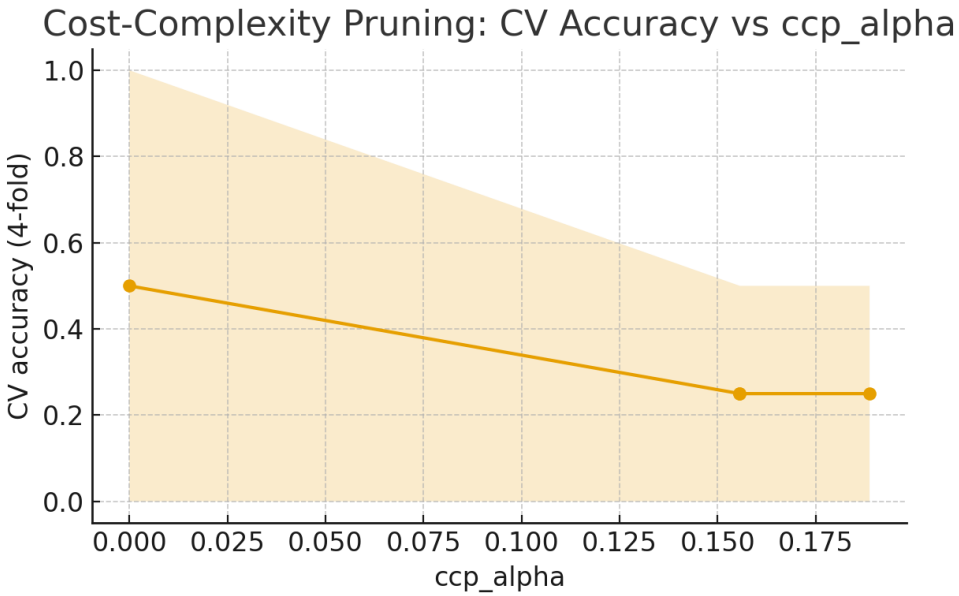
$$\alpha_2 = \frac{1.0 - 0.811278}{2 - 1} = \boxed{0.188722}.$$

   Khi $\alpha > \alpha_2$, tiếp tục **cắt** về cây gốc (1 lá).

Kết luận theo α.

$$T^\star(\alpha) = \begin{cases} T_{\text{full}}, & 0 \le \alpha < 0.155639, \\ T_{\text{1-split}}, & 0.155639 \le \alpha < 0.188722, \\ T_{\text{root}}, & \alpha \ge 0.188722. \end{cases}$$

Các ngưỡng α_1 và α_2 **trùng khớp** với đường cắt tỉa trả về bởi `sklearn` (`ccp_alphas` ≈ 0.155639 và 0.188722), xác thực phép tính tay.



Hình 9: Kết quả cắt tỉa cây trả về bởi sklearn

3.2 Random Forest

Random Forest (**Breiman, 2001**) là một phương pháp *ensemble* dựa trên ý tưởng *Bagging* (Bootstrap Aggregating), nhưng bổ sung thêm sự ngẫu nhiên khi lựa chọn tập thuộc tính tại mỗi nút.

- **Bagging**: với tập dữ liệu gốc D gồm n mẫu, sinh ra B tập con D_b (bootstrap) bằng cách lấy ngẫu nhiên có hoàn lại n mẫu. Với mỗi D_b, huấn luyện một cây T_b.
- **Random subspace**: khi xây dựng T_b, tại mỗi nút chỉ xem xét m thuộc tính (với $m < d$, d là tổng số thuộc tính) để tìm split tốt nhất.

Dự đoán cuối cùng:

$$\hat{y}(x) = \text{majority}\{\, T_b(x) \,:\, b = 1, \dots, B\} \quad \text{(phân loại)},$$

$$\hat{y}(x) = \frac{1}{B} \sum_{b=1}^{B} T_b(x) \quad \text{(hồi quy)}.$$

Ý nghĩa:

- Bagging $\Rightarrow$ giảm phương sai.
- Random subspace $\Rightarrow$ giảm tương quan giữa các cây $\Rightarrow$ tăng hiệu quả khi lấy phiếu/trung bình.

Random Forest là *baseline mạnh*:

- Hạn chế overfitting, đặc biệt khi B lớn.
- Làm việc tốt với dữ liệu pha trộn (biến số, biến rời rạc).
- Có ước lượng quan trọng thuộc tính (*feature importance*) thông qua giảm Gini/Entropy hoặc hoán vị ngẫu nhiên (Permutation Importance).

Điểm yếu: kích thước mô hình lớn, thời gian dự đoán chậm hơn cây đơn, ít trực quan để diễn giải từng quyết định cá nhân.

3.2.1 OOB Error

Một mẫu x_i có xác suất không xuất hiện trong một bootstrap bất kỳ là

$$\left(1 - \tfrac{1}{n}\right)^n \xrightarrow{n\to\infty} e^{-1} \approx 0.368.$$

Do đó trung bình 36.8% mẫu *không được chọn* trong một bootstrap cụ thể. Các mẫu này được gọi là **out-of-bag (OOB)** đối với cây T_b.

Dự đoán nhãn x_i bằng đa số phiếu chỉ từ những cây mà x_i là OOB. Sai số OOB thu được là một ước lượng *không thiên lệch* của sai số tổng quát, thường thay thế được cho cross-validation.

3.2.2 Nguyên lý hoạt động

Để thấy rõ cách Random Forest hoạt động, ta sẽ mô phỏng bằng tay với $B = 3$ cây quyết định, huấn luyện trên tập 8 mẫu ở phần Decision Tree.

Bước 1: Sinh bootstrap. Mỗi bootstrap chọn ngẫu nhiên $n = 8$ mẫu *có hoàn lại* từ tập gốc. Ví dụ:

- Bootstrap D_1: {1,2,3,4,5,6,6,7}
- Bootstrap D_2: {1,1,2,3,5,5,8,8}
- Bootstrap D_3: {2,3,4,4,6,7,7,8}

Như vậy, mỗi tập có mẫu bị lặp lại và cũng có mẫu không xuất hiện (OOB).

Bước 2: Huấn luyện cây trên từng bootstrap. Ta giới hạn độ sâu = 1 để dễ tính toán.

- Với D_1: tỉ lệ $(A = 1)$ trội hơn $\Rightarrow$ split theo A. Gain theo Entropy ≈ 0.189, như đã tính. Cây T_1: gốc A.
- Với D_2: do chọn trùng nhiều mẫu có $B = 1$, phân bố gần cân bằng $\Rightarrow$ split theo B cũng có thể xuất hiện. Gain theo Entropy ở B trong bootstrap này nhỏ, nhưng nếu chỉ xét một thuộc tính $(m = 1)$, T_2 buộc chọn B.
- Với D_3: phần lớn mẫu có $A = 0$, nên split theo A vẫn cho gain cao hơn. Cây T_3: gốc A.

Kết quả: T_1 và T_3 chọn A, T_2 chọn B.

Bước 3: Bỏ phiếu đa số. Khi dự đoán một mẫu mới, Random Forest lấy dự đoán của cả ba cây rồi bỏ phiếu:

$$\hat{y}(x) = \text{majority}\{T_1(x), T_2(x), T_3(x)\}.$$

Ví dụ:

- Mẫu $(A = 1, B = 0)$:
  - T_1: dự đoán bệnh (dựa vào nhánh $A = 1$ nhiều dương).
  - T_2: split theo B, $B = 0 \Rightarrow$ dự đoán bệnh.
  - T_3: dự đoán bệnh (giống T_1).

  $\Rightarrow$ Đa số phiếu: Bệnh.
- Mẫu $(A = 0, B = 0)$:
  - T_1: dự đoán không bệnh.
  - T_2: split theo B, $B = 0 \Rightarrow$ dự đoán không bệnh.
  - T_3: dự đoán không bệnh.

  $\Rightarrow$ Đa số phiếu: Không bệnh.

Bước 4: Sai số OOB (Out-of-Bag). Xét mẫu số 8: $(A = 0, B = 0, y = 0)$. Trong ba bootstrap:

- D_1 không chứa mẫu 8 $\Rightarrow$ OOB cho T_1.
- D_2 chứa mẫu 8 hai lần $\Rightarrow$ không OOB cho T_2.
- D_3 chứa mẫu 8 $\Rightarrow$ không OOB cho T_3.

Vậy dự đoán OOB cho mẫu 8 chỉ đến từ T_1: dự đoán đúng $y = 0$. Lặp lại cho toàn bộ 8 mẫu, ta có thể tính sai số OOB = số mẫu OOB bị dự đoán sai / tổng số dự đoán OOB. Trên tập nhỏ này, kết quả OOB khớp với sai số huấn luyện $\approx 25\%$.

3.2.3 Đối chiếu bằng code Python

```
# Random Forest: sklearn OOB vs manual OOB
rf = RandomForestClassifier(
    n_estimators=200,
    criterion="entropy",
    max_depth=2,
    max_features=1,      # encourage diversity
    bootstrap=True,
    oob_score=True,
    random_state=42
)
rf.fit(X, y)

sklearn_train_acc = accuracy_score(y, rf.predict(X))
sklearn_oob_score = rf.oob_score_
sklearn_feature_importances = rf.feature_importances_
```

```
sklearn_cm = confusion_matrix(y, rf.predict(X))

# Manual bagging
rng = np.random.RandomState(42)
B = 200
trees = []
oob_sets = []
for b in range(B):
    idx = rng.randint(0, len(df), size=len(df))  # bootstrap indices
    oob = np.setdiff1d(np.arange(len(df)), np.unique(idx))
    oob_sets.append(oob)
    clf = DecisionTreeClassifier(criterion="entropy", max_depth=2, random_state=1000+b)
    clf.fit(X[idx], y[idx])
    trees.append(clf)

def majority_vote(preds):
    c = Counter(preds)
    if len(c)==0:
        return None
    most_common = c.most_common()
    # tie-break: choose 0 for determinism
    if len(most_common) >= 2 and most_common[0][1] == most_common[1][1]:
        return 0
    return most_common[0][0]

# manual train accuracy
manual_preds = []
for i in range(len(df)):
    votes = [tree.predict(X[i].reshape(1,-1))[0] for tree in trees]
    manual_preds.append(majority_vote(votes))

manual_train_acc = accuracy_score(y, manual_preds)
manual_cm = confusion_matrix(y, manual_preds)

# manual OOB accuracy
oob_preds = [None]*len(df)
oob_count = np.zeros(len(df), dtype=int)

for b, tree in enumerate(trees):
    oob = oob_sets[b]
    preds = tree.predict(X[oob])
    for j, idx in enumerate(oob):
        if oob_preds[idx] is None:
            oob_preds[idx] = []
        oob_preds[idx].append(preds[j])
        oob_count[idx] += 1

oob_final = []
oob_true = []
```

```
for i in range(len(df)):
    if oob_count[i] > 0:
        oob_final.append(majority_vote(oob_preds[i]))
        oob_true.append(y[i])

manual_oob_acc = accuracy_score(oob_true, oob_final)

comparison = pd.DataFrame({
    "Metric": [
        "Train Accuracy",
        "OOB Accuracy",
        "Feature Importance: A",
        "Feature Importance: B"
    ],
    "sklearn_RF": [
        f"{sklearn_train_acc:.3f}",
        f"{sklearn_oob_score:.3f}",
        f"{sklearn_feature_importances[0]:.3f}",
        f"{sklearn_feature_importances[1]:.3f}",
    ],
    "Manual_Bagging": [
        f"{manual_train_acc:.3f}",
        f"{manual_oob_acc:.3f}",
        "-", "-"
    ]
})
```

```
=== Random Forest (sklearn) ===
Train acc = 0.750, OOB acc = 0.500
Feature importances (A,B) = [0.64783428 0.35216572]
Confusion matrix (train):
[[3 1]
 [1 3]]
=== Random Forest (manual bagging) ===
Train acc = 0.750, OOB acc = 0.500
Confusion matrix (train):
[[4 0]
 [2 2]]
```

Hình 10: So sánh giữa xây Random Forest bằng sklearn và tính tay

3.3 AdaBoost

3.3.1 Nguyên lý

AdaBoost (Freund & Schapire, 1997) là thuật toán *boosting* phổ biến, kết hợp nhiều mô hình yếu (*weak learners*) như **decision stump** (cây 1 nút). Ý tưởng: mỗi vòng lặp t, ta huấn luyện một stump h_t trên phân phối trọng số $w^{(t)}$ của dữ liệu; stump tốt sẽ có lỗi $\epsilon_t < 0.5$. Sau đó ta gán cho nó một hệ số α_t tỷ lệ thuận với độ chính xác và cập nhật lại trọng số mẫu: mẫu sai bị phạt nặng hơn, để vòng sau được chú ý hơn.

- AdaBoost phù hợp khi dữ liệu ít nhiễu và có nhiều mẫu dễ phân loại, vì nó tập trung vào các mẫu khó.
- Mỗi weak learner rất đơn giản (thường là stump), nhưng kết hợp nhiều vòng cho độ chính xác cao.
- Hạn chế: nhạy cảm với ngoại lệ và nhiễu, vì các điểm "bất thường" sẽ bị đẩy trọng số rất lớn.

Công thức tổng quát:

$$\epsilon_t = \sum_{i=1}^{n} w_i^{(t)} \, [y_i \neq h_t(x_i)], \qquad \alpha_t = \tfrac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t},$$

$$w_i^{(t+1)} \propto w_i^{(t)} \exp\big(-\alpha_t y_i h_t(x_i)\big).$$

Cuối cùng, dự đoán của mô hình là dấu của tổng có trọng số:

$$H(x) = \text{sign}\left(\sum_{t=1}^{T} \alpha_t h_t(x)\right).$$

3.3.2 Chứng minh công thức α_t

AdaBoost tối thiểu hoá **exponential loss**:

$$L = \sum_{i=1}^{n} \exp(-y_i F(x_i)), \qquad F(x) = \sum_{t=1}^{T} \alpha_t h_t(x).$$

Xét vòng t, giả sử ta đã có $F_{t-1}(x)$, thêm stump h_t với trọng số α_t:

$$L^{(t)} = \sum_{i=1}^{n} w_i^{(t)} \exp\big(-\alpha_t y_i h_t(x_i)\big),$$

với $w_i^{(t)} = \exp(-y_i F_{t-1}(x_i))$.

Tách thành hai nhóm:

$$L^{(t)} = \exp(-\alpha_t) \sum_{i:y_i = h_t(x_i)} w_i^{(t)} + \exp(+\alpha_t) \sum_{i:y_i \neq h_t(x_i)} w_i^{(t)}.$$

Đặt

$$W_+ = \sum_{i:y_i = h_t(x_i)} w_i^{(t)} = 1 - \epsilon_t, \quad W_- = \sum_{i:y_i \neq h_t(x_i)} w_i^{(t)} = \epsilon_t,$$

(sau khi chuẩn hoá $w_i^{(t)}$ thành phân phối).

Ta cần chọn α_t để tối thiểu hoá:

$$L^{(t)}(\alpha_t) = W_+ \exp(-\alpha_t) + W_- \exp(\alpha_t).$$

Lấy đạo hàm:

$$\frac{dL}{d\alpha_t} = -W_+ \exp(-\alpha_t) + W_- \exp(\alpha_t) = 0.$$

Giải ra:

$$W_- \exp(\alpha_t) = W_+ \exp(-\alpha_t) \quad \Rightarrow \quad \exp(2\alpha_t) = \frac{W_+}{W_-}.$$

Vậy:

$$\boxed{\alpha_t = \tfrac{1}{2} \ln \frac{1-\epsilon_t}{\epsilon_t}}$$

3.3.3 Nguyên lý hoạt động

Xét tập dữ liệu gồm 8 mẫu với hai thuộc tính A, B và nhãn $y \in \{-1, +1\}$ (suy ra từ $\{0, 1\}$ sang $\{-1, +1\}$). Ban đầu mọi trọng số đều bằng nhau:

$$w_i^{(1)} = \frac{1}{8} = 0.125, \quad \forall i.$$

Vòng 1: huấn luyện stump h_1. Chọn stump

$$h_1(x) = \text{sign}(A - 0.5) = \begin{cases} +1 & \text{nếu } A = 1, \\ -1 & \text{nếu } A = 0. \end{cases}$$

- Sai tại 2 mẫu: #3 ($A = 1, y = -1$) và #5 ($A = 0, y = +1$).
- Lỗi có trọng số:

$$\epsilon_1 = 2 \times 0.125 = 0.25.$$

- Trọng số của stump:

$$\alpha_1 = \tfrac{1}{2} \ln \frac{1 - \epsilon_1}{\epsilon_1} = \tfrac{1}{2} \ln \frac{0.75}{0.25} = \tfrac{1}{2} \ln 3 \approx 0.5493.$$

Cập nhật trọng số:

$$w_i^{(2)} \propto w_i^{(1)} \exp(-\alpha_1 y_i h_1(x_i)).$$

Nếu dự đoán đúng: $w_i^{(2)} \approx 0.125 \cdot e^{-\alpha_1} = 0.0721$. Nếu dự đoán sai: $w_i^{(2)} \approx 0.125 \cdot e^{+\alpha_1} = 0.2165$. Sau chuẩn hoá: Hai mẫu sai có trọng số ≈ 0.25. Sáu mẫu đúng có trọng số ≈ 0.0833.

Bảng 1: Kết quả sau vòng 1 của AdaBoost

ID	(A, B)	y	$h_1(x)$	$w_i^{(2)}$ (chuẩn hoá)
1	(1,1)	$+1$	$+1$	0.0833
2	(1,0)	$+1$	$+1$	0.0833
3	(1,1)	-1	$+1$	0.2500
4	(1,0)	$+1$	$+1$	0.0833
5	(0,1)	$+1$	-1	0.2500
6	(0,1)	-1	-1	0.0833
7	(0,0)	-1	-1	0.0833
8	(0,0)	-1	-1	0.0833

Vòng 2: thử stump h_2. Giả sử chọn

$$h_2(x) = \text{sign}(B - 0.5).$$

- Kiểm tra trên các mẫu có trọng số lớn (ID #3, #5):
  - Mẫu #3: ($B = 1, y = -1$), stump dự đoán $+1 \Rightarrow$ sai.
  - Mẫu #5: ($B = 1, y = +1$), stump dự đoán $+1 \Rightarrow$ đúng.
- Tổng thể: stump này dự đoán sai đúng một nửa khối lượng trọng số, tức

$$\epsilon_2 = 0.5, \qquad \alpha_2 = \tfrac{1}{2} \ln \frac{1 - 0.5}{0.5} = 0.$$

Do $\alpha_2 = 0$, stump này không đóng góp thêm. Trong thực tế, AdaBoost sẽ duyệt qua nhiều stump/thuộc tính/ngưỡng để chọn cây tốt nhất. Nếu tất cả đều cho $\epsilon = 0.5$, quá trình dừng.

3.3.4 Đối chiếu bằng code Python

```
import numpy as np
import pandas as pd
from math import log, sqrt, exp
from sklearn.ensemble import AdaBoostClassifier
from sklearn.tree import DecisionTreeClassifier

# ---------- Data ----------
X = np.array([
    [1,1],[1,0],[1,1],[1,0],
    [0,1],[0,1],[0,0],[0,0]
])
y01 = np.array([1,1,0,1,1,0,0,0])        # {0,1}
y   = np.where(y01==1, +1, -1)           # {-1,+1}
n = len(y)

# ---------- Round 1: hand calculation for stump on A ----------
w1 = np.ones(n)/n
h1 = np.where(X[:,0]==1, +1, -1)         # stump on A
err_mask = (h1 != y)
eps1 = float(np.sum(w1*err_mask))        # 0.25
alpha1 = 0.5*log((1-eps1)/eps1)          # 0.5*ln(3)

c = exp(-alpha1)                         # e^{-alpha1} = 1/sqrt(3)
d = exp(+alpha1)                         # e^{+alpha1} = sqrt(3)

w2_unnorm = w1 * np.exp(-alpha1 * y * h1)
Z1 = float(np.sum(w2_unnorm))
w2 = w2_unnorm / Z1

hand_round1 = pd.DataFrame({
    "ID": np.arange(1,n+1),
    "A": X[:,0], "B": X[:,1], "y(±1)": y,
    "h1(A)": h1,
    "error?": err_mask.astype(int),
    "w1": np.full(n, 1/n),
    "w2_unnorm": np.round(w2_unnorm, 6),
    "w2_norm": np.round(w2, 6),
})

summary = pd.DataFrame([
    {"round": 1, "learner": "sign(A-0.5)", "epsilon": eps1, "alpha": alpha1, "Z": Z1}
])

# ---------- Round 2: stump on B to check epsilon ----------
h2 = np.where(X[:,1]==1, +1, -1)         # stump on B
err2 = (h2 != y)
eps2 = float(np.sum(w2*err2))            # should be 0.5
```

```
alpha2 = 0.5*log((1-eps2)/eps2) if eps2 not in (0,1) else 0.0
summary.loc[len(summary)] = {"round": 2, "learner": "sign(B-0.5)",
"epsilon": eps2, "alpha": alpha2, "Z": 1.0}

print("=== Hand calc: round 1 table ===")
print(hand_round1.to_string(index=False))
print("\n=== Hand calc: summary per round ===")
print(summary.to_string(index=False))

# ---------- Compare with sklearn (use SAMME to match alpha formula) ----------
clf = AdaBoostClassifier(
    estimator=DecisionTreeClassifier(max_depth=1),
    n_estimators=2,       # try 2 to inspect first two learners
    algorithm="SAMME",    # ensures alpha = 0.5*ln((1-eps)/eps)
    random_state=0
)
clf.fit(X, y01)

print("\n=== sklearn AdaBoost (SAMME) ===")
print("Estimator errors (eps):", clf.estimator_errors_)
print("Estimator weights (alpha):", clf.estimator_weights_)
print("Stump 1 is on feature A?  Inspect by predicting:", clf.estimators_[0].predict(X))
if len(clf.estimators_) > 1:
    print("Stump 2 is on which feature? Predict:", clf.estimators_[1].predict(X))
```

```
=== Hand calc: round 1 table ===
 ID  A  B  y(±1)  h1(A)  error?    w1  w2_unnorm  w2_norm
  1  1  1      1      1       0 0.125   0.072169 0.083333
  2  1  0      1      1       0 0.125   0.072169 0.083333
  3  1  1     -1      1       1 0.125   0.216506 0.250000
  4  1  0      1      1       0 0.125   0.072169 0.083333
  5  0  1      1     -1       1 0.125   0.216506 0.250000
  6  0  1     -1     -1       0 0.125   0.072169 0.083333
  7  0  0     -1     -1       0 0.125   0.072169 0.083333
  8  0  0     -1     -1       0 0.125   0.072169 0.083333

=== Hand calc: summary per round ===
 round     learner  epsilon    alpha        Z
     1 sign(A-0.5)     0.25 0.549306 0.866025
     2 sign(B-0.5)     0.50 0.000000 1.000000

=== sklearn AdaBoost (SAMME) ===
Estimator errors (eps): [0.25 1.  ]
Estimator weights (alpha): [1.09861229 0.        ]
Stump 1 is on feature A?  Inspect by predicting: [1 1 1 1 0 0 0 0]
```

Hình 11: So sánh việc dùng sklearn và tính tay với mô hình Adaboost

3.4 Gradient Boosting

3.4.1 Nguyên lý

Xét mô hình cộng tuần tự

$$F_T(x) = \sum_{t=1}^{T} \nu\, f_t(x),$$

trong đó mỗi f_t là một cây hồi quy nông (thường là stump/cây độ sâu nhỏ), và $\nu \in (0, 1]$ là learning rate.

Gradient Boosting thực hiện **gradient descent trong không gian hàm**. Ở vòng t, với mô hình hiện thời F_{t-1}, ta tối thiểu hoá hàm mất mát tổng quát

$$\mathcal{L}(F) = \sum_{i=1}^{n} \ell\big(y_i, F(x_i)\big).$$

Pseudo-residuals (âm đạo hàm theo F) được tính bởi

$$r_i^{(t)} = - \left.\frac{\partial \ell(y_i, F)}{\partial F}\right|_{F=F_{t-1}(x_i)}.$$

Sau đó, fit một cây hồi quy f_t xấp xỉ $r_i^{(t)}$ (theo MSE). Với mỗi lá R_{tj} của f_t, giải bước đi tối ưu một chiều

$$\gamma_{tj} = \arg\min_{\gamma} \sum_{x_i \in R_{tj}} \ell\big(y_i,\, F_{t-1}(x_i) + \gamma\big).$$

Cập nhật:

$$F_t(x) = F_{t-1}(x) + \nu \sum_{j} \gamma_{tj}\, \mathbf{1}\{x \in R_{tj}\}.$$

3.4.2 Trường hợp đặc biệt

Hồi quy MSE. Với $\ell(y, F) = \frac{1}{2}(y - F)^2$, ta có $r_i^{(t)} = y_i - F_{t-1}(x_i)$ và

$$\gamma_{tj} = \frac{1}{|R_{tj}|} \sum_{x_i \in R_{tj}} r_i^{(t)} \quad \text{(trung bình residual)}.$$

Phân loại nhị phân (logistic loss). Dùng $y_i \in \{0, 1\}$ và log-loss:

$$\ell(y, F) = -y \log p - (1 - y) \log(1 - p), \quad p = \sigma(F) = \frac{1}{1 + e^{-F}}.$$

Ta có

$$g_i \equiv \frac{\partial \ell}{\partial F} = p_i - y_i, \qquad h_i \equiv \frac{\partial^2 \ell}{\partial F^2} = p_i(1 - p_i).$$

Do đó **pseudo-residual** là $r_i^{(t)} = -g_i = y_i - p_i$.

3.4.3 Chứng minh công thức bước lá $\gamma^* = -\frac{\sum g_i}{\sum h_i}$

Ở một lá R, ta cần

$$\gamma^* = \arg\min_{\gamma} \sum_{i \in R} \ell\big(y_i,\, F_{t-1}(x_i) + \gamma\big).$$

Xấp xỉ Taylor bậc hai quanh F_{t-1}:

$$\ell(y_i, F_{t-1} + \gamma) \approx \ell(y_i, F_{t-1}) + g_i\,\gamma + \tfrac{1}{2} h_i\,\gamma^2.$$

Cộng theo $i \in R$, bỏ hằng số, nghiệm tối ưu của bài toán là nghiệm Newton:

$$\frac{d}{d\gamma}\left(\sum_{i \in R} g_i\,\gamma + \tfrac{1}{2} h_i\,\gamma^2\right) = \sum_{i \in R} g_i + \gamma \sum_{i \in R} h_i = 0,$$

$$\boxed{\gamma^* = -\frac{\sum_{i \in R} g_i}{\sum_{i \in R} h_i}}.$$

Với log-loss, $g_i = p_i - y_i$, $h_i = p_i(1 - p_i)$, nên ta có công thức đóng *trên từng lá*. (Đây cũng chính là bước Newton một chiều dùng trong nhiều biến thể như XGBoost khi $\lambda = 0$.)

3.4.4 Nguyên lý hoạt động

Dữ liệu 8 mẫu như các phần trước, dùng $y \in \{0, 1\}$. Khởi tạo $F_0(x) \equiv 0 \Rightarrow p_i^{(0)} = \sigma(0) = 0.5$.

Vòng 1. Xét hai split ứng viên A và B. Tính các tổng trên từng phía của split (trái: giá trị thuộc tính $= 0$, phải: $= 1$). Ở F_0, ta có $g_i = 0.5 - y_i$, $h_i = 0.25$.

Split theo A:

$$A = 1:\ 3 \text{ dương, } 1 \text{ âm} \Rightarrow \sum g_i = 3(0.5 - 1) + 1(0.5 - 0) = -1.0,\ \ \sum h_i = 4 \cdot 0.25 = 1.0,$$
$$A = 0:\ 1 \text{ dương, } 3 \text{ âm} \Rightarrow \sum g_i = +1.0,\ \ \sum h_i = 1.0.$$

$$\Rightarrow \gamma_L = -\frac{-1}{1} = +1.0, \quad \gamma_R = -\frac{+1}{1} = -1.0.$$

Cập nhật (lấy $\nu = 1$ để minh hoạ):

$$F_1(x) = \begin{cases} +1.0, & A = 1, \\ -1.0, & A = 0, \end{cases} \qquad p^{(1)} = \sigma(F_1) = \begin{cases} 0.731, & A = 1, \\ 0.269, & A = 0. \end{cases}$$

Sai 2/8 mẫu (độ chính xác = 75%), trùng với cây quyết định theo A.

Vòng 2. Tính lại $g_i^{(1)} = p_i^{(1)} - y_i$, $h_i^{(1)} = p_i^{(1)}(1 - p_i^{(1)})$. Với A=1: mỗi mẫu có $p = 0.731$ nên

$$\sum g_i = 3(0.731 - 1) + 1(0.731 - 0) = -0.076, \quad \sum h_i = 4 \cdot (0.731 \cdot 0.269) = 0.787.$$

Với A=0: mỗi mẫu có $p = 0.269$ nên

$$\sum g_i = 1(0.269 - 1) + 3(0.269 - 0) = +0.076, \quad \sum h_i = 0.787.$$

Suy ra

$$\gamma_L = -\frac{-0.076}{0.787} = +0.0966, \qquad \gamma_R = -\frac{+0.076}{0.787} = -0.0966.$$

Cập nhật:

$$F_2(x) = \begin{cases} +1.0966, & A = 1, \\ -1.0966, & A = 0, \end{cases} \qquad p^{(2)} = \sigma(F_2) = \begin{cases} 0.749, & A = 1, \\ 0.249, & A = 0. \end{cases}$$

Biên quyết định được "đẩy xa" hơn (xác suất tự tin hơn); số lỗi không đổi (2/8) với stump, nhưng log-loss giảm theo đúng mục tiêu tối ưu.

- Learning rate $\nu < 1$ (shrinkage) + số vòng lớn giúp tổng quát hoá tốt hơn.
- Có thể dùng *subsample* < 1 (Stochastic GB) để giảm phương sai.
- Độ sâu cây nhỏ (1–3) giúp tránh overfit và giữ ý nghĩa "sửa lỗi dần dần".

3.4.5 Đối chiếu bằng code Python

```
import numpy as np
import pandas as pd
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.metrics import accuracy_score

# ----------------------------
# Data
# ----------------------------
X = np.array([
    [1,1],[1,0],[1,1],[1,0],
    [0,1],[0,1],[0,0],[0,0]
], dtype=float)
y = np.array([1,1,0,1,1,0,0,0], dtype=float)  # labels {0,1}
n = len(y)
A = X[:,0].astype(int)
B = X[:,1].astype(int)

sigmoid = lambda z: 1/(1+np.exp(-z))

def region_mask(vals, thr=0.5):
    return (vals <= thr), (vals > thr)

def sums_G_H(F, mask):
    idx = np.where(mask)[0]
    if idx.size == 0: return 0.0, 0.0
    p = sigmoid(F)
    G = float(np.sum(p[idx] - y[idx]))
    H = float(np.sum(p[idx] * (1-p[idx])))
    return G, H

def gamma_from_GH(G,H):
    return 0.0 if H==0 else -G/H

def gain_second_order(F, vals):
    p = sigmoid(F)
```

```
    G_all = float(np.sum(p-y)); H_all = float(np.sum(p*(1-p)))
    L, R = region_mask(vals)
    GL,HL = sums_G_H(F,L); GR,HR = sums_G_H(F,R)
    part = lambda g,h: 0.0 if h==0 else (g*g)/h
    gain = 0.5*( part(GL,HL)+part(GR,HR)-part(G_all,H_all) )
    return gain, (GL,HL,GR,HR)

def take_step(F, vals, lr=1.0):
    gainA, statsA = gain_second_order(F,A)
    gainB, statsB = gain_second_order(F,B)
    if gainA >= gainB:
        feat, stats, use = "A", statsA, A
    else:
        feat, stats, use = "B", statsB, B
    GL,HL,GR,HR = stats
    gL = gamma_from_GH(GL,HL); gR = gamma_from_GH(GR,HR)
    L,R = region_mask(use)
    F_new = F.copy()
    F_new[L] += lr*gL; F_new[R] += lr*gR
    return F_new, feat, gL, GL, HL, gR, GR, HR

# ----------------------------
# Hand derivation
# ----------------------------
F0 = np.zeros(n); p0 = sigmoid(F0)
acc0 = accuracy_score(y, (p0>=0.5).astype(int))

F1, split1, gL1, GL1, HL1, gR1, GR1, HR1 = take_step(F0,A)
p1 = sigmoid(F1); acc1 = accuracy_score(y,(p1>=0.5).astype(int))

F2, split2, gL2, GL2, HL2, gR2, GR2, HR2 = take_step(F1,A)
p2 = sigmoid(F2); acc2 = accuracy_score(y,(p2>=0.5).astype(int))

stage_summary = pd.DataFrame([
    {"stage":0,"split":"-","acc":acc0},
    {"stage":1,"split":split1,"gamma_left":gL1,"G_left":GL1,"H_left":HL1,
     "gamma_right":gR1,"G_right":GR1,"H_right":HR1,"acc":acc1},
    {"stage":2,"split":split2,"gamma_left":gL2,"G_left":GL2,"H_left":HL2,
     "gamma_right":gR2,"G_right":GR2,"H_right":HR2,"acc":acc2},
])

per_sample = pd.DataFrame({
    "ID": np.arange(1,n+1),
    "A":A,"B":B,"y":y.astype(int),
    "F0":np.round(F0,6),"p0":np.round(p0,6),
    "F1":np.round(F1,6),"p1":np.round(p1,6),
    "F2":np.round(F2,6),"p2":np.round(p2,6),
})
```

```
print("=== Hand-derived Gradient Boosting ===")
print(stage_summary.to_string(index=False))
print(per_sample.to_string(index=False))

# ----------------------------
# sklearn comparison
# ----------------------------
print("=== sklearn - Gradient Boosting ===")
gb = GradientBoostingClassifier(
    loss="log_loss", learning_rate=1.0,
    max_depth=1, n_estimators=2, random_state=0
)
gb.fit(X,y.astype(int))

from itertools import islice
stages = list(islice(gb.staged_decision_function(X),0,2))
def show_stage(name,Fraw):
    Fraw = np.ravel(Fraw)
    df = pd.DataFrame({"A":A,"F":Fraw,"p":sigmoid(Fraw)})
    print(f"\n{name}")
    print("A=1 mean:", df[df.A==1][["F","p"]].mean().to_dict())
    print("A=0 mean:", df[df.A==0][["F","p"]].mean().to_dict())

show_stage("Stage-0 (init)", np.zeros_like(y))
show_stage("Stage-1", stages[0])
show_stage("Stage-2", stages[1])
```

```
=== Hand-derived Gradient Boosting ===
 stage split  acc  gamma_left   G_left   H_left  gamma_right   G_right  H_right
     0     - 0.50         NaN      NaN      NaN          NaN       NaN      NaN
     1     A 0.75   -1.000000 1.000000 1.000000     1.000000 -1.000000 1.000000
     2     A 0.75   -0.096339 0.075766 0.786448     0.096339 -0.075766 0.786448
 ID  A  B  y  F0  p0   F1       p1        F2       p2
  1  1  1  1 0.0 0.5  1.0 0.731059  1.096339 0.749574
  2  1  0  1 0.0 0.5  1.0 0.731059  1.096339 0.749574
  3  1  1  0 0.0 0.5  1.0 0.731059  1.096339 0.749574
  4  1  0  1 0.0 0.5  1.0 0.731059  1.096339 0.749574
  5  0  1  1 0.0 0.5 -1.0 0.268941 -1.096339 0.250426
  6  0  1  0 0.0 0.5 -1.0 0.268941 -1.096339 0.250426
  7  0  0  0 0.0 0.5 -1.0 0.268941 -1.096339 0.250426
  8  0  0  0 0.0 0.5 -1.0 0.268941 -1.096339 0.250426
=== sklearn - Gradient Boosting ===

Stage-0 (init)
A=1 mean: {'F': 0.0, 'p': 0.5}
A=0 mean: {'F': 0.0, 'p': 0.5}

Stage-1
A=1 mean: {'F': 1.0, 'p': 0.7310585786300049}
A=0 mean: {'F': -1.0, 'p': 0.2689414213699951}

Stage-2
A=1 mean: {'F': 1.0963391237638203, 'p': 0.749573539410271}
A=0 mean: {'F': -1.0963391237638203, 'p': 0.25042646058972895}
```

Hình 12: So sánh việc dùng sklearn và tính tay với mô hình Gradient Boosting

3.5 XGBoost

3.5.1 Nguyên lý tổng quát

XGBoost (Extreme Gradient Boosting) mở rộng từ Gradient Boosting bằng việc dùng khai triển Taylor bậc hai của hàm mất mát.

3.5.2 Mục tiêu và khai triển Taylor bậc hai

Ở vòng t, ta thêm một cây f_t (lá R_{tj}, giá trị lá w_{tj}) vào

$$F_t(x) = F_{t-1}(x) + f_t(x), \qquad \Omega(f_t) = \gamma T_t + \frac{\lambda}{2}\sum_{j=1}^{T_t} w_{tj}^2.$$

Hàm mục tiêu:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} \ell\big(y_i,\, F_{t-1}(x_i) + f_t(x_i)\big) + \Omega(f_t).$$

Khai triển Taylor quanh F_{t-1}:

$$\ell(y_i, F_{t-1} + f_t) \approx \ell(y_i, F_{t-1}) + g_i f_t(x_i) + \tfrac{1}{2} h_i f_t(x_i)^2,$$

với

$$g_i = \frac{\partial \ell}{\partial F}\Big|_{F_{t-1}(x_i)}, \qquad h_i = \frac{\partial^2 \ell}{\partial F^2}\Big|_{F_{t-1}(x_i)}.$$

Bỏ hằng số $\sum_i \ell(y_i, F_{t-1})$, ta tối ưu

$$\tilde{\mathcal{L}}^{(t)} = \sum_{j=1}^{T_t} \Big[G_{tj} w_{tj} + \tfrac{1}{2}(H_{tj} + \lambda) w_{tj}^2 \Big] + \gamma T_t, \quad G_{tj} = \sum_{i \in R_{tj}} g_i, \;\; H_{tj} = \sum_{i \in R_{tj}} h_i.$$

3.5.3 Chứng minh nghiệm tối ưu từng lá và công thức *gain*

Với mỗi lá j:

$$\frac{\partial}{\partial w_{tj}} \Big[G_{tj} w_{tj} + \tfrac{1}{2}(H_{tj} + \lambda) w_{tj}^2 \Big] = 0 \;\Rightarrow\; \boxed{w_{tj}^* = -\frac{G_{tj}}{H_{tj} + \lambda}}.$$

Thế vào:

$$\tilde{\mathcal{L}}_j^* = -\tfrac{1}{2} \frac{G_{tj}^2}{H_{tj} + \lambda} + \gamma.$$

Chia một vùng R (có G, H) thành R_L, R_R (có G_L, H_L và G_R, H_R), **Thông tin đạt được**:

$$\boxed{\text{Gain} = \tfrac{1}{2} \left(\frac{G_L^2}{H_L + \lambda} + \frac{G_R^2}{H_R + \lambda} - \frac{G^2}{H + \lambda} \right) - \gamma}.$$

Ta chọn split có Gain lớn nhất.

3.5.4 Vì sao xuất hiện "Similarity Score"?

(1) Từ hàm mất mát xấp xỉ. Trong XGBoost, sau khi khai triển Taylor bậc hai, hàm mục tiêu cho một lá R có dạng:

$$\tilde{\mathcal{L}}(w) = G(R)\, w + \tfrac{1}{2}\big(H(R) + \lambda\big) w^2,$$

với

$$G(R) = \sum_{i \in R} g_i, \qquad H(R) = \sum_{i \in R} h_i.$$

(2) Nghiệm tối ưu của lá. Giải tối ưu:

$$w^* = -\frac{G(R)}{H(R) + \lambda}.$$

Thay vào hàm mục tiêu, ta có loss tối ưu:

$$\tilde{\mathcal{L}}_R^* = -\tfrac{1}{2} \frac{G(R)^2}{H(R) + \lambda}.$$

(3) Định nghĩa Similarity Score. Đại lượng

$$\text{Score}(R) = \frac{G(R)^2}{H(R) + \lambda}$$

chính là phần tỉ lệ với mức giảm loss mà lá R có thể mang lại. Do đó, Similarity Score được dùng như thước đo "chất lượng" của một lá.

(4) Ý nghĩa trực giác.

- Nếu tất cả gradient g_i trong lá có **cùng dấu** (đều âm hoặc đều dương) $\Rightarrow$ chúng "giống nhau" $\Rightarrow$ tổng $G(R)$ lớn về độ lớn $\Rightarrow$ Score cao $\Rightarrow$ lá đồng nhất và tốt.
- Nếu gradient **trái dấu** (một số âm, một số dương) $\Rightarrow$ chúng triệt tiêu nhau $\Rightarrow$ $G(R)$ nhỏ $\Rightarrow$ Score thấp $\Rightarrow$ lá không đồng nhất.

(5) Liên hệ với Gain khi split. Khi chia một vùng P thành hai lá con L và R, công thức độ lợi là:

$$\text{Gain} = \tfrac{1}{2}\big(\text{Score}(L) + \text{Score}(R) - \text{Score}(P)\big) - \gamma.$$

Split tốt là split làm tăng tổng Similarity Score.

(6) Ví dụ minh họa. Giả sử có 4 điểm trong một lá, với Hessian $h_i = 1$ và $\lambda = 1$.

- **Trường hợp A:** Gradient đều giống nhau, $g = \{-1, -1, -1, -1\}$.

$$G = -4, \quad H = 4, \quad \text{Score} = \frac{(-4)^2}{4+1} = \frac{16}{5} = 3.2.$$

  Lá rất đồng nhất, Score cao.

- **Trường hợp B:** Gradient lẫn lộn, $g = \{-1, -1, +1, +1\}$.

$$G = 0, \quad H = 4, \quad \text{Score} = \frac{0^2}{4+1} = 0.$$

  Các gradient triệt tiêu nhau, lá không đồng nhất, Score thấp.

(7) Kết luận. Similarity Score phản ánh mức độ "đồng thuận" của các gradient trong một lá. - Score cao $\Rightarrow$ lá "giống nhau" về hướng cập nhật $\Rightarrow$ cập nhật mạnh, giảm loss nhiều. - Score thấp $\Rightarrow$ lá chứa gradient trái chiều $\Rightarrow$ không cải thiện được nhiều.

Khi nào dùng g, h, khi nào dùng residual/Score?

- **Trường hợp tổng quát (mọi hàm mất mát):** Luôn viết bằng gradient và Hessian:

$$G = \sum g_i,\ H = \sum h_i, \quad w^* = -\tfrac{G}{H+\lambda}, \quad \text{Score} = \tfrac{G^2}{H+\lambda}.$$

- **Phân loại nhị phân (log-loss):** $g_i = p_i - y_i,\ h_i = p_i(1-p_i)$. Nếu gọi residual $r_i = y_i - p_i = -g_i$ thì:

$$\text{Score}(R) = \frac{\big(\sum r_i\big)^2}{\sum p_i(1-p_i) + \lambda}.$$

- **Hồi quy MSE:** Với MSE, Hessian $h_i = 1$ cho mọi mẫu. Khi đó:

$$w^* = \frac{\sum r_i}{|R| + \lambda}, \qquad \text{Score}(R) = \frac{(\sum r_i)^2}{|R| + \lambda},$$

  với $r_i = y_i - F(x_i)$ là residual. Vì h_i không đổi nên ta không cần viết g, h nữa mà chỉ dùng residual và số mẫu trong lá.

3.5.5 Nguyên lý hoạt động

Dữ liệu 8 mẫu như các phần trước, dùng nhãn $y \in \{0, 1\}$. Khởi tạo $F_0(x) \equiv 0 \Rightarrow p_i^{(0)} = \sigma(F_0) = 0.5$. - Với log-loss: $g_i = p_i - y_i$, $h_i = p_i(1 - p_i)$. Tại $t = 1$: $p_i = 0.5 \Rightarrow g_i = 0.5 - y_i \in \{\pm 0.5\}$, $h_i = 0.25$ cho mọi i.

Cha (toàn bộ tập).

$$G = \sum_{i=1}^{8} g_i = \underbrace{4 \cdot (-0.5)}_{4 \text{ mẫu } y=1} + \underbrace{4 \cdot (+0.5)}_{4 \text{ mẫu } y=0} = 0, \qquad H = \sum_{i=1}^{8} h_i = 8 \cdot 0.25 = 2.$$

$$\text{Score(parent)} = \frac{G^2}{H + \lambda} = \frac{0^2}{2 + 1} = 0.$$

Ứng viên split theo A. Nhóm $A = 1$ có 3 mẫu $y = 1$, 1 mẫu $y = 0$; nhóm $A = 0$ có 1 mẫu $y = 1$, 3 mẫu $y = 0$.

$$\text{Nhánh } A{=}1: \quad G_L = \sum_{A=1} g_i = 3(0.5 - 1) + 1(0.5 - 0) = -1.0, \quad H_L = \sum_{A=1} h_i = 4 \cdot 0.25 = 1.0,$$
$$\text{Nhánh } A{=}0: \quad G_R = \sum_{A=0} g_i = 1(0.5 - 1) + 3(0.5 - 0) = +1.0, \quad H_R = 1.0.$$

Trọng số lá (nghiệm tối ưu):

$$w_L^* = -\frac{G_L}{H_L + \lambda} = -\frac{-1}{1 + 1} = +0.5, \qquad w_R^* = -\frac{G_R}{H_R + \lambda} = -\frac{+1}{1 + 1} = -0.5.$$

Similarity score của từng lá:

$$\text{Score}(L) = \frac{G_L^2}{H_L + \lambda} = \frac{1}{2} = 0.5, \qquad \text{Score}(R) = 0.5.$$

Gain của split:

$$\text{Gain} = \tfrac{1}{2}\left(\underbrace{\frac{1}{2}}_{\text{Score}(L)} + \underbrace{\frac{1}{2}}_{\text{Score}(R)} - \underbrace{0}_{\text{parent}} \right) - \gamma = \boxed{0.5}.$$

Ứng viên split theo B (đối chiếu). Trong tập này, mỗi phía của B đều có 2 mẫu $y = 1$ và 2 mẫu $y = 0 \Rightarrow G_{B=1} = 0,\ H_{B=1} = 1;\ G_{B=0} = 0,\ H_{B=0} = 1.$

$$\text{Score}(B{=}1) = \frac{0^2}{1 + 1} = 0, \quad \text{Score}(B{=}0) = 0 \Rightarrow \text{Gain}_B = \tfrac{1}{2}(0 + 0 - 0) = 0.$$

Kết luận vòng 1: chọn split theo A (Gain $= 0.5$ lớn hơn).

Cập nhật mô hình sau vòng 1.

$$F_1(x) = \begin{cases} +0.5 & \text{nếu } A = 1, \\ -0.5 & \text{nếu } A = 0, \end{cases} \quad p^{(1)} = \sigma(F_1) = \begin{cases} 0.62246 & (A = 1), \\ 0.37754 & (A = 0). \end{cases}$$

(Tuỳ chọn) Vòng 2: tính nhanh γ^* trên cùng split A. Tính lại $g = p - y$, $h = p(1-p)$ trên từng nhánh:

$$\begin{aligned} A{=}1: \sum g &= 3(0.6225-1) + 1(0.6225-0) = -0.1329, \\ \sum h &= 4 \cdot [0.6225 \cdot 0.3775] = 0.9400; \\ A{=}0: \sum g &= 1(0.3775-1) + 3(0.3775-0) = +0.1329, \\ \sum h &= 0.9400. \end{aligned}$$

$$w_L^{*(2)} = -\frac{-0.1329}{0.9400+1} = +0.0686, \qquad w_R^{*(2)} = -\frac{+0.1329}{0.9400+1} = -0.0686.$$

Biên dự đoán được "đẩy xa" hơn (xác suất tự tin hơn) nhưng số lỗi không đổi với stump; log-loss tiếp tục giảm — đúng tinh thần boosting.

3.5.6 Đối chiếu bằng code Python

```
import numpy as np
import pandas as pd
from math import exp
from sklearn.metrics import accuracy_score

# ---------- Data (8 samples) ----------
X = np.array([
    [1,1],[1,0],[1,1],[1,0],
    [0,1],[0,1],[0,0],[0,0]
], dtype=float)
y = np.array([1,1,0,1,1,0,0,0], dtype=float)  # {0,1}
A = X[:,0].astype(int)
B = X[:,1].astype(int)
n = len(y)

lam = 1.0   #  (L2 on leaf)
gamma = 0.0 #  (penalty per leaf / split)
lr = 1.0    # learning_rate

sigmoid = lambda z: 1.0/(1.0+np.exp(-z))

def sums_GH(F, mask):
    p = sigmoid(F)
    idx = np.where(mask)[0]
    G = float(np.sum(p[idx] - y[idx]))               # gradient sum
    H = float(np.sum(p[idx] * (1.0 - p[idx])))       # hessian sum
    return G, H

def leaf_weight(G, H, lam):
    return - G / (H + lam)

def score(G, H, lam):
    return (G*G) / (H + lam) if (H + lam) != 0 else 0.0
```

```
def gain(parent, left, right, lam, gamma):
    Gp, Hp = parent; Gl, Hl = left; Gr, Hr = right
    return 0.5*(score(Gl,Hl,lam) + score(Gr,Hr,lam) - score(Gp,Hp,lam)) - gamma

def split_stats(F, feat):
    if feat == "A":
        L = (A==0); R = (A==1)
    else:
        L = (B==0); R = (B==1)
    parent = sums_GH(F, np.ones(n, dtype=bool))
    left   = sums_GH(F, L)
    right  = sums_GH(F, R)
    return parent, left, right, L, R

# ======================
# Hand derivation
# ======================
print("=== HAND DERIVATION (log-loss) ===")
F0 = np.zeros(n)           # base_score = 0.5 -> F0 = 0
p0 = sigmoid(F0)
acc0 = accuracy_score(y, (p0>=0.5).astype(int))

# Stage 1: try split A and B at F0
parentA, leftA, rightA, LA, RA = split_stats(F0, "A")
GA0,HA0 = leftA   # A=0
GA1,HA1 = rightA  # A=1
wA0 = leaf_weight(GA0,HA0,lam)
wA1 = leaf_weight(GA1,HA1,lam)
gainA = gain(parentA, leftA, rightA, lam, gamma)

parentB, leftB, rightB, LB, RB = split_stats(F0, "B")
gainB = gain(parentB, leftB, rightB, lam, gamma)

print(f"[Stage1] A=0: G={GA0:.4f}, H={HA0:.4f}, w*={wA0:.4f},
Score={score(GA0,HA0,lam):.4f}")
print(f"[Stage1] A=1: G={GA1:.4f}, H={HA1:.4f}, w*={wA1:.4f},
Score={score(GA1,HA1,lam):.4f}")
print(f"[Stage1] Gain(A) = {gainA:.4f}")
GL0,HL0 = leftB; GR1,HR1 = rightB
print(f"[Stage1] B=0: G={GL0:.4f}, H={HL0:.4f}, Score={score(GL0,HL0,lam):.4f}")
print(f"[Stage1] B=1: G={GR1:.4f}, H={HR1:.4f}, Score={score(GR1,HR1,lam):.4f}")
print(f"[Stage1] Gain(B) = {gainB:.4f}")

# Update F1 with best split (A)
F1 = np.where(A==1, lr*wA1, lr*wA0)  # +0.5 (A=1) / -0.5 (A=0)
p1 = sigmoid(F1)
acc1 = accuracy_score(y, (p1>=0.5).astype(int))

# Optional Stage 2 (again splitting A to show Newton steps shrink)
```

```
parent2, left2, right2, L2, R2 = split_stats(F1, "A")
G0,H0 = left2; G1,H1 = right2
w2A0 = leaf_weight(G0,H0,lam)
w2A1 = leaf_weight(G1,H1,lam)
gain2A = gain(parent2, left2, right2, lam, gamma)

print(f"\n[Stage2] A=0: G={G0:.4f}, H={H0:.4f}, w*={w2A0:.4f},
Score={score(G0,H0,lam):.4f}")
print(f"[Stage2] A=1: G={G1:.4f}, H={H1:.4f}, w*={w2A1:.4f},
Score={score(G1,H1,lam):.4f}")
print(f"[Stage2] Gain(A) = {gain2A:.4f}")

hand_table = pd.DataFrame({
    "ID": np.arange(1,n+1),
    "A": A, "B": B, "y": y.astype(int),
    "F0": F0, "p0": p0,
    "F1_hand": F1, "p1_hand": p1,
})

# ======================
# XGBoost sklearn-style
# ======================
try:
    from xgboost import XGBClassifier, DMatrix
    import xgboost as xgb
except Exception as e:
    raise SystemExit(
        "Bạn cần cài gói 'xgboost' (pip install xgboost). "
        f"Lỗi import: {e}"
    )

# 1 tree, depth=1, lr=1,  =1,  =0, base_score=0.5, no sampling
xgb_model = XGBClassifier(
    objective="binary:logistic",
    n_estimators=1,
    max_depth=1,
    learning_rate=1.0,
    reg_lambda=lam,
    reg_alpha=0.0,
    min_child_weight=0.0,   # để không chặn split
    subsample=1.0,
    colsample_bytree=1.0,
    gamma=gamma,
    base_score=0.5,
    booster="gbtree",
    grow_policy="depthwise",
    tree_method="exact",
    random_state=0
)
```

```
xgb_model.fit(X, y)

# Raw margin and probability from the tree
booster = xgb_model.get_booster()
dm = xgb.DMatrix(X, label=y)
F1_xgb = booster.predict(dm, output_margin=True)  # raw score after 1 tree
p1_xgb = 1/(1+np.exp(-F1_xgb))

# ---- Extract tree structure (compatible across xgboost versions) ----
booster = xgb_model.get_booster()
dm = xgb.DMatrix(X, label=y)
F1_xgb = booster.predict(dm, output_margin=True)  # raw score after 1 tree
p1_xgb = 1/(1+np.exp(-F1_xgb))

df_tree = booster.trees_to_dataframe()

def is_leaf_row(df):
    feat = df.get("Feature", None)
    if feat is None:
        return pd.Series(False, index=df.index)
    return (feat.astype(str).str.lower() == "leaf") | (feat.isna())

root_rows = df_tree[~is_leaf_row(df_tree)]
if len(root_rows) == 0:
    root = df_tree.iloc[0]
else:
    if "Node" in root_rows.columns:
        root_rows = root_rows.sort_values(by="Node")
    root = root_rows.iloc[0]

split_feature = root.get("Feature", "unknown")  # e.g., 'f0' ~ cột 0 = A
gain_xgb = float(root.get("Gain", 0.0))

leaf_mask = is_leaf_row(df_tree)
if "Value" in df_tree.columns:
    leaf_vals = df_tree.loc[leaf_mask, "Value"].astype(float).tolist()
elif "Split" in df_tree.columns:
    leaf_vals = df_tree.loc[leaf_mask, "Split"].astype(float).tolist()
else:
    leaf_vals = []

print("\n=== XGBoost (sklearn API) - model summary ===")
print("Split feature at root:", split_feature, "(f0 ~ A, f1 ~ B)")
print("Leaf weights (from tree DataFrame):", leaf_vals)
print("Root split gain (xgboost):", round(gain_xgb, 6))

comp = hand_table.copy()
comp["F1_xgb"] = F1_xgb
comp["p1_xgb"] = p1_xgb
```

```
comp["dF"] = comp["F1_xgb"] - comp["F1_hand"]
comp["dp"] = comp["p1_xgb"] - comp["p1_hand"]

print("\n=== Per-sample comparison (Stage 1) ===")
print(comp.to_string(index=False))

print("\nExpected (hand): split on A, leaves ~ [+0.5, -0.5], gain ~ 0.5")
```

```
=== HAND DERIVATION (log-loss) ===
[Stage1] A=0: G=1.0000, H=1.0000, w*=-0.5000, Score=0.5000
[Stage1] A=1: G=-1.0000, H=1.0000, w*=0.5000, Score=0.5000
[Stage1] Gain(A) = 0.5000
[Stage1] B=0: G=0.0000, H=1.0000, Score=0.0000
[Stage1] B=1: G=0.0000, H=1.0000, Score=0.0000
[Stage1] Gain(B) = 0.0000

[Stage2] A=0: G=0.5102, H=0.9400, w*=-0.2630, Score=0.1342
[Stage2] A=1: G=-0.5102, H=0.9400, w*=0.2630, Score=0.1342
[Stage2] Gain(A) = 0.1342

=== XGBoost (sklearn API) — model summary ===
Split feature at root: f0 (f0 ~ A, f1 ~ B)
Leaf weights (from tree DataFrame): [nan, nan]
Root split gain (xgboost): 1.0

=== Per-sample comparison (Stage 1) ===
 ID  A  B  y  F0  p0  F1_hand  p1_hand  F1_xgb   p1_xgb  dF           dp
  1  1  1  1 0.0 0.5      0.5 0.622459     0.5 0.622459 0.0 2.081459e-08
  2  1  0  1 0.0 0.5      0.5 0.622459     0.5 0.622459 0.0 2.081459e-08
  3  1  1  0 0.0 0.5      0.5 0.622459     0.5 0.622459 0.0 2.081459e-08
  4  1  0  1 0.0 0.5      0.5 0.622459     0.5 0.622459 0.0 2.081459e-08
  5  0  1  1 0.0 0.5     -0.5 0.377541    -0.5 0.377541 0.0 8.987728e-09
  6  0  1  0 0.0 0.5     -0.5 0.377541    -0.5 0.377541 0.0 8.987728e-09
  7  0  0  0 0.0 0.5     -0.5 0.377541    -0.5 0.377541 0.0 8.987728e-09
  8  0  0  0 0.0 0.5     -0.5 0.377541    -0.5 0.377541 0.0 8.987728e-09

Expected (hand): split on A, leaves ~ [+0.5, -0.5], gain ~ 0.5
```

Hình 13: So sánh kết quả giữa tính tay và dùng sklearn với XGBoost

3.6 LightGBM

3.6.1 Nguyên lý

LightGBM (Microsoft, 2017) vẫn dựa trên khung Gradient Boosting Decision Tree (GBDT) nhưng được thiết kế để **tăng tốc** và **giảm bộ nhớ**, đặc biệt trên tập dữ liệu lớn với nhiều đặc trưng. Những cải tiến chính:

- **Histogram-based algorithm**: Thay vì lưu toàn bộ giá trị liên tục của feature, LightGBM lượng tử hoá (discretize) vào một số lượng *bins* hữu hạn. Khi tính toán split, chỉ cần duyệt qua bins, thay vì duyệt toàn bộ mẫu. ⇒ tiết kiệm RAM và tăng tốc đáng kể.

- **Leaf-wise growth (best-first)**: Khác với XGBoost (depth-wise, mở rộng cân bằng theo tầng), LightGBM chọn *lá có gain lớn nhất* để tách trước. Điều này cho phép đào sâu một nhánh tiềm năng, đạt loss thấp nhanh hơn, nhưng dễ dẫn đến overfitting nếu không có regularization.
- **Gradient-based One-Side Sampling (GOSS)**: Giữ lại toàn bộ các mẫu có gradient lớn (mẫu khó dự đoán) và chỉ lấy ngẫu nhiên một phần nhỏ các mẫu gradient nhỏ. ⇒ vẫn giữ được "tín hiệu quan trọng" mà giảm số mẫu phải xử lý.
- **Exclusive Feature Bundling (EFB)**: Trong dữ liệu thưa (sparse), nhiều đặc trưng hiếm khi cùng khác 0. LightGBM gộp chúng vào một "feature bundle" duy nhất để giảm chiều dữ liệu mà không mất thông tin.

3.6.2 Các cải tiến cốt lõi của LightGBM: ví dụ và chứng minh

Ký hiệu chung. Tại một vòng boosting, với mô hình hiện tại F, đặt $g_i = \frac{\partial \ell}{\partial F}(x_i)$, $h_i = \frac{\partial^2 \ell}{\partial F^2}(x_i)$. Với một vùng (lá) R, ký hiệu $G(R) = \sum_{i \in R} g_i$, $H(R) = \sum_{i \in R} h_i$. Với tham số regularization $\lambda, \gamma \geq 0$, ta có:

$$w^*(R) \;=\; -\frac{G(R)}{H(R)+\lambda}, \qquad \text{Score}(R) \;=\; \frac{G(R)^2}{H(R)+\lambda},$$

$$\text{Gain}(\text{P}\to L, R) = \tfrac{1}{2}\Big(\tfrac{G_L^2}{H_L+\lambda} + \tfrac{G_R^2}{H_R+\lambda} - \tfrac{G_P^2}{H_P+\lambda}\Big) - \gamma.$$

(1) Histogram-based algorithm: lượng tử hoá feature thành bins

Ý tưởng. Thay vì duyệt mọi ngưỡng tách trên giá trị liên tục x, ta lượng tử hoá x vào các bin rời rạc $\mathcal{B} = \{b_1, \ldots, b_K\}$ theo các biên $(\tau_0 < \tau_1 < \cdots < \tau_K)$ và *cộng dồn* gradient/hessian theo bin:

$$G(b_k) = \sum_{i:\, x_i \in (\tau_{k-1}, \tau_k]} g_i, \qquad H(b_k) = \sum_{i:\, x_i \in (\tau_{k-1}, \tau_k]} h_i.$$

Khi xét một split tại ranh bin m (trái: $\cup_{k \leq m}$, phải: $\cup_{k>m}$), công thức Gain dùng tổng G, H theo bin:

$$\text{Gain}(m) = \tfrac{1}{2}\left(\frac{\big(\sum_{k\leq m} G(b_k)\big)^2}{\sum_{k\leq m} H(b_k)+\lambda} + \frac{\big(\sum_{k>m} G(b_k)\big)^2}{\sum_{k>m} H(b_k)+\lambda} - \frac{\big(\sum_{k} G(b_k)\big)^2}{\sum_{k} H(b_k)+\lambda}\right) - \gamma.$$

Chứng minh tính đúng đắn (bảo toàn hình thức). Với bất kỳ vùng R là hợp các bin, theo tính chất tuyến tính của tổng:

$$G(R) = \sum_{b_k \subset R} G(b_k), \qquad H(R) = \sum_{b_k \subset R} H(b_k).$$

Do đó mọi công thức w^*, Score, Gain chỉ thay $\sum_{i \in R}$ bằng $\sum_{b_k \subset R}$, *không đổi hình thức*. Sai số (nếu có) chỉ đến từ việc *không xét* các ngưỡng *giữa* biên bin (xấp xỉ hoá không gian ngưỡng), chứ *không* đến từ công thức tối ưu.

Ví dụ tính tay Xét một feature liên tục x có 6 mẫu đã sắp:

$$x = (1.0,\ 1.1,\ 1.9,\ 2.0,\ 3.1,\ 3.2), \quad (g,h) = (-1,1),(-1,1),(+1,1),(+1,1),(-1,1),(+1,1).$$

Chia 3 bin: $(1, 1.5], (1.5, 2.5], (2.5, 3.5]$. Khi đó

$$\begin{aligned} G(b_1) &= -1-1 = -2,\ H(b_1) = 2; \\ G(b_2) &= +1+1 = +2,\ H(b_2) = 2; \\ G(b_3) &= -1+1 = 0,\ H(b_3) = 2. \end{aligned}$$

Với $\lambda = \gamma = 0$, xét split tại $m = 1$ (trái b_1, phải $b_2 \cup b_3$):

$$\text{Gain} = \tfrac{1}{2}\left(\frac{(-2)^2}{2} + \frac{(+2)^2}{4} - \frac{0^2}{6}\right) = \tfrac{1}{2}(2+1) = 1.5.$$

Nếu duyệt ngưỡng *liên tục* đúng ngay sau $x = 1.1$ (tức tách 2 mẫu đầu), ta nhận cùng $G_L = -2, H_L = 2$ và $G_R = +2, H_R = 4 \Rightarrow$ Gain trùng 1.5. Khi ngưỡng liên tục rơi *trong* một bin, histogram sẽ dùng biên gần nhất $\Rightarrow$ sai khác chỉ do xấp xỉ ngưỡng, không do công thức.

(2) Leaf-wise growth (best-first): tách lá có Gain lớn nhất

Ý tưởng. Tại mỗi bước, trong *tập ứng viên là các lá hiện có*, chọn lá $R^\star$ có Gain lớn nhất để tách trước (best-first). Khác với depth-wise (tách đồng loạt tất cả lá ở một độ sâu), leaf-wise có thể đào sâu *một nhánh* mang lại giảm loss nhiều nhất.

Chứng minh tính tối ưu *cục bộ* (greedy bước một). Hàm mục tiêu xấp xỉ theo tổng các lá *cộng* lại và mỗi split độc lập đóng góp một Gain:

$$\tilde{\mathcal{L}} = \sum_j \tilde{\mathcal{L}}_j^* \;=\; \text{hằng} - \tfrac{1}{2}\sum_j \frac{G(R_j)^2}{H(R_j)+\lambda} + \gamma\,\#\text{lá}.$$

Ở một bước bất kỳ, việc tách *một* lá R chỉ thay thế $\tilde{\mathcal{L}}_R^*$ bằng $\tilde{\mathcal{L}}_{R_L}^* + \tilde{\mathcal{L}}_{R_R}^*$, độ giảm đúng bằng $\text{Gain}(R \to R_L, R_R)$. Do vậy, trong *một bước*, chọn lá có Gain lớn nhất là tối ưu (greedy *one-step optimal*).

Ví dụ. Giả sử có 2 lá hiện tại R_1, R_2 với $\text{Gain}_1 = 0.40,\ \text{Gain}_2 = 0.15$ (cùng γ, λ). Leaf-wise sẽ tách R_1 trước vì giảm loss nhiều hơn ngay lập tức; depth-wise có thể buộc tách cả hai (nếu cùng độ sâu), kém hiệu quả khi ngân sách split hạn chế.

Gradient-based One-Side Sampling (GOSS): giữ gradient lớn, lấy mẫu gradient nhỏ

Mục tiêu. Giảm số mẫu phải xét khi tính histogram, nhưng *không làm sai lệch* hướng cập nhật.

Thuật toán. Sắp theo $|g_i|$. Chọn toàn bộ top-a (tỷ lệ a) mẫu có $|g|$ lớn: $\mathcal{A} = \{i : |g_i| \text{ lớn}\}$. Trong phần còn lại $\mathcal{B}$, lấy ngẫu nhiên tỷ lệ b: $\mathcal{B}' \subset \mathcal{B}$. **Scale** phần $\mathcal{B}'$ bằng hệ số $s = \frac{1-a}{b}$ khi cộng dồn G, H:

$$\widehat{G}(R) = \sum_{i \in \mathcal{A} \cap R} g_i \;+\; s \sum_{i \in \mathcal{B}' \cap R} g_i, \qquad \widehat{H}(R) = \sum_{i \in \mathcal{A} \cap R} h_i \;+\; s \sum_{i \in \mathcal{B}' \cap R} h_i.$$

Chứng minh *không chệch* (unbiased) về kỳ vọng. Với một chỉ báo $Z_i = \mathbf{1}\{i \in \mathcal{B}'\}$ độc lập, $\mathbb{E}[Z_i] = b$ với $i \in \mathcal{B}$. Do đó

$$\mathbb{E}[\widehat{G}(R)] = \sum_{i \in \mathcal{A} \cap R} g_i + s \sum_{i \in \mathcal{B} \cap R} \mathbb{E}[Z_i g_i] = \sum_{i \in \mathcal{A} \cap R} g_i + \tfrac{1-a}{b} \cdot b \sum_{i \in \mathcal{B} \cap R} g_i = \sum_{i \in R} g_i \;=\; G(R).$$

Tương tự, $\mathbb{E}[\widehat{H}(R)] = H(R)$. Suy ra *trung bình* theo chọn mẫu, biểu thức Score/Gain tính từ $(\widehat{G}, \widehat{H})$ là xấp xỉ *không chệch* của giá trị gốc. Thực tế, GOSS nhấn mạnh mẫu khó (gradient lớn) nên phương sai cũng nhỏ đi cho phần "quan trọng".

Ví dụ. Giả sử một vùng có (g, h):

$$\{(-2,1), (-1,1), (+0.1,1), (+0.1,1), (+0.1,1), (+0.1,1)\}.$$

Chọn $a = \frac{2}{6}$ giữ $-2, -1$; trong 4 mẫu còn lại, lấy $b = \frac{1}{2} \Rightarrow$ lấy ngẫu nhiên 2 mẫu nhỏ và scale $s = \frac{1-a}{b} = \frac{4/6}{1/2} = \frac{4}{3}$. Kỳ vọng $\widehat{G} = (-2) + (-1) + \frac{4}{3} \cdot 0.2 = -3 + \frac{4}{15} \approx -2.733$ (ở đây minh hoạ một vùng con; khi xét toàn tập và theo split, kỳ vọng *trên toàn không gian chọn mẫu* sẽ khớp đúng G thật; ví dụ đơn này cho thấy g lớn được giữ nguyên, g nhỏ được scale để bù phần bị bỏ).

Exclusive Feature Bundling (EFB): gộp feature loại trừ nhau

Ý tưởng. Trong dữ liệu thưa, hai đặc trưng $x^{(u)}, x^{(v)}$ gọi là *loại trừ* nếu hiếm khi cùng khác 0: $x^{(u)}(i) \cdot x^{(v)}(i) \approx 0$ với hầu hết i. Khi đó có thể gộp vào một feature z *không chồng chéo* bằng cách ánh xạ mỗi giá trị về các *dải rời* (disjoint ranges) của z:

$$z(i) = \begin{cases} \text{offset}_u + \text{bin}\big(x^{(u)}(i)\big), & \text{nếu } x^{(u)}(i) \neq 0, \\ \text{offset}_v + \text{bin}\big(x^{(v)}(i)\big), & \text{nếu } x^{(v)}(i) \neq 0, \\ 0, & \text{nếu cả hai bằng 0.} \end{cases}$$

Các offset được đặt sao cho hai nhóm bin của u và v *không trùng*.

Chứng minh bảo toàn tách (trường hợp loại trừ đúng). Nếu $x^{(u)}$ và $x^{(v)}$ *không bao giờ* cùng khác 0, thì mọi split trên z tương đương một split trên *một* trong hai đặc trưng gốc (vì miền giá trị z là hợp rời nhau của hai cụm bin). Do vậy histogram $G(b), H(b)$ theo z chính là gộp của histogram theo $x^{(u)}$ và $x^{(v)}$ mà *không có va chạm* $\Rightarrow$ Score/Gain bảo toàn.

Khi có va chạm hiếm. Nếu một số ít quan sát có cả hai đặc trưng khác 0, có thể: (i) đặt thêm offset/đánh dấu để tách biệt, hoặc (ii) chấp nhận một *xấp xỉ* nhỏ (đưa hai giá trị vào cùng bin). Sai số chỉ ảnh hưởng *chọn ngưỡng tốt nhất*, không thay đổi công thức tối ưu trên tổng G, H.

Ví dụ mini. Giả sử hai feature nhị phân loại trừ $x^{(u)}, x^{(v)} \in \{0, 1\}$: trên 6 mẫu, ba mẫu có $x^{(u)} = 1$, ba mẫu có $x^{(v)} = 1$, không có mẫu nào cả hai cùng 1. Gộp thành z với offset: giá trị 1 của u ánh xạ vào bin $\{1\}$; giá trị 1 của v ánh xạ vào bin $\{10\}$. Mọi split trên z tại ngưỡng giữa 1 và 10 chính là "chọn u hay v". Cộng dồn G, H theo bin của z đúng bằng cộng dồn riêng rẽ theo u hoặc v.

3.6.3 Chứng minh công thức cập nhật tại mỗi lá

Giả sử ta đã có mô hình F_{t-1} tại vòng $t-1$. Tại vòng t, thêm cây f_t gồm các lá R_{tj} với giá trị dự đoán w_{tj}:

$$F_t(x) = F_{t-1}(x) + f_t(x), \qquad f_t(x) = \sum_j w_{tj}\, \mathbf{1}\{x \in R_{tj}\}.$$

Khai triển Taylor bậc hai loss quanh F_{t-1}:

$$\tilde{\mathcal{L}}^{(t)} = \sum_{i=1}^{n} \Big[g_i f_t(x_i) + \tfrac{1}{2} h_i f_t(x_i)^2 \Big] + \Omega(f_t),$$

trong đó

$$g_i = \frac{\partial \ell}{\partial F}\Big|_{F_{t-1}(x_i)}, \quad h_i = \frac{\partial^2 \ell}{\partial F^2}\Big|_{F_{t-1}(x_i)}, \quad \Omega(f_t) = \gamma T + \tfrac{\lambda}{2} \sum_j w_{tj}^2.$$

Nhóm theo từng lá R_{tj}:

$$G_{tj} = \sum_{i \in R_{tj}} g_i, \quad H_{tj} = \sum_{i \in R_{tj}} h_i.$$

Do đó:

$$\tilde{\mathcal{L}}^{(t)} = \sum_j \Big(G_{tj} w_{tj} + \tfrac{1}{2}(H_{tj} + \lambda) w_{tj}^2 \Big) + \gamma T.$$

Tối ưu theo w_{tj}:

$$\frac{\partial}{\partial w_{tj}} \Big[G_{tj} w_{tj} + \tfrac{1}{2}(H_{tj} + \lambda) w_{tj}^2 \Big] = G_{tj} + (H_{tj} + \lambda) w_{tj}.$$

Cho bằng 0:

$$\boxed{w_{tj}^* = -\frac{G_{tj}}{H_{tj}+\lambda}}.$$

Thế ngược lại:

$$\tilde{\mathcal{L}}_j^* = -\frac{1}{2}\frac{G_{tj}^2}{H_{tj}+\lambda} + \gamma.$$

Độ lợi khi split. Nếu tách một vùng P thành hai lá L, R:

$$\boxed{\text{Gain} = \tfrac{1}{2}\left(\frac{G_L^2}{H_L+\lambda} + \frac{G_R^2}{H_R+\lambda} - \frac{G_P^2}{H_P+\lambda}\right) - \gamma}.$$

Vì sao dùng histogram vẫn giữ công thức này? Khi đặc trưng được lượng tử hoá thành bins, thay vì tính trên từng mẫu, ta cộng gradient/hessian theo bin:

$$G_{\text{bin}} = \sum_{i\in\text{bin}} g_i, \qquad H_{\text{bin}} = \sum_{i\in\text{bin}} h_i.$$

Khi duyệt split tại ranh bin, công thức cho w^*, Score và Gain vẫn giống hệt, chỉ thay $\sum_i$ bằng $\sum_{\text{bin}}$.

3.6.4 Ví dụ tính tay (log-loss, $\lambda = 1,\ \gamma = 0$)

Thiết lập. Tập 8 mẫu, 2 đặc trưng A, B, nhãn $y \in \{0, 1\}$. Khởi tạo $F_0 \equiv 0 \Rightarrow p_0 = \sigma(F_0) = 0.5$. Với log-loss: $g_i = p_i - y_i$, $h_i = p_i(1-p_i)$. Tại F_0: $g_i \in \{\pm 0.5\}$, $h_i = 0.25$ cho mọi i.

Cha (toàn bộ). Có 4 mẫu $y = 1$ và 4 mẫu $y = 0$:

$$G_P = \sum g_i = 4(-0.5) + 4(+0.5) = 0, \qquad H_P = \sum h_i = 8 \cdot 0.25 = 2, \qquad \text{Score}(P) = \frac{G_P^2}{H_P+\lambda} = 0.$$

Split theo A (vòng 1). Nhóm $A = 1$ có 3 mẫu y=1 và 1 mẫu y=0; $A = 0$ thì ngược lại.

$$A=1: \quad G=-1.0,\ H=1.0 \quad \Rightarrow \quad w^* = -\frac{G}{H+\lambda} = +\frac{1}{2} = +0.5, \ \text{Score} = \frac{1}{2} = 0.5,$$
$$A=0: \quad G=+1.0,\ H=1.0 \quad \Rightarrow \quad w^* = -\frac{1}{2} = -0.5, \ \text{Score} = 0.5.$$

Gain của split:

$$\text{Gain}_A = \tfrac{1}{2}\big(0.5 + 0.5 - 0\big) = \mathbf{0.5}.$$

(Cùng công thức của LightGBM/XGBoost; lưu ý một số bản cài đặt báo split_gain *không nhân* 1/2 nên có thể in ra ≈ 1.0.)

Split theo B (đối chiếu). Hai phía cân bằng $2/2 \Rightarrow G = 0,\ H = 1 \Rightarrow \text{Score} = 0 \Rightarrow \text{Gain}_B = 0$. Vậy **chọn split theo A**.

Cập nhật sau vòng 1.

$$F_1(x) = \begin{cases} +0.5, & A = 1 \\ -0.5, & A = 0 \end{cases} \quad \Rightarrow \quad p^{(1)} = \sigma(F_1) = \begin{cases} 0.62245933, & A = 1 \\ 0.37754067, & A = 0 \end{cases}$$

(tương ứng $h^{(1)} = p^{(1)}(1-p^{(1)}) = 0.23500371$ cho mọi mẫu.)

Vòng 2 (tính trên phân phối mới). Tính lại $g = p^{(1)} - y$, $h = p^{(1)}(1 - p^{(1)})$ theo từng lá.

- Lá $A = 1$ (3 dương, 1 âm):

$$G^{(2)}_{A=1} = 3(0.62245933-1)+1(0.62245933-0) = -0.51016268, \qquad H^{(2)}_{A=1} = 4\cdot 0.23500371 = 0.94001485.$$

- Lá $A = 0$ (1 dương, 3 âm):

$$G^{(2)}_{A=0} = 1(0.37754067 - 1) + 3(0.37754067 - 0) = +0.51016268, \qquad H^{(2)}_{A=0} = 0.94001485.$$

Giá trị lá tối ưu của cây thứ 2 (nếu lại chọn split giống nhau):

$$\boxed{w^{*(2)}_{A=1} = -\frac{-0.51016268}{0.94001485 + 1} = +0.262991, \qquad w^{*(2)}_{A=0} = -\frac{+0.51016268}{0.94001485 + 1} = -0.262991.}$$

Nghĩa là cây thứ 2 cộng thêm ± 0.263 (theo nhánh A), làm biên F "đẩy xa hơn" (xác suất tự tin hơn), đúng tinh thần boosting.

3.6.5 Đối chiếu bằng code Python

```
import numpy as np, pandas as pd
from sklearn.metrics import accuracy_score

# ----- Data (8 samples) -----
X = np.array([
    [1,1],[1,0],[1,1],[1,0],
    [0,1],[0,1],[0,0],[0,0]
], dtype=float)
y = np.array([1,1,0,1,1,0,0,0], dtype=float)  # {0,1}
A = X[:,0].astype(int); B = X[:,1].astype(int)
n = len(y)

lam = 1.0   # L2 leaf
gamma = 0.0 # split penalty
lr = 1.0    # learning rate (shrinkage)

sigmoid = lambda z: 1/(1+np.exp(-z))

def sums_GH(F, mask):
    p = sigmoid(F); idx = np.where(mask)[0]
    G = float(np.sum(p[idx] - y[idx]))
    H = float(np.sum(p[idx] * (1-p[idx])))
    return G, H

def leaf_weight(G,H,lam): return - G / (H + lam)
def score(G,H,lam): return (G*G)/(H+lam) if (H+lam)!=0 else 0.0
def gain(parent,left,right,lam,gamma, half=True):
    Gp,Hp = parent; Gl,Hl = left; Gr,Hr = right
    g = (score(Gl,Hl,lam) + score(Gr,Hr,lam) - score(Gp,Hp,lam))
    return 0.5*g - gamma if half else g - gamma  # LightGBM may report without 1/2
```

```
def split_stats(F, feat):
    if feat=="A":
        L = (A==0); R = (A==1)
    else:
        L = (B==0); R = (B==1)
    parent = sums_GH(F, np.ones(n, dtype=bool))
    left   = sums_GH(F, L)
    right  = sums_GH(F, R)
    return parent, left, right, L, R

# ======= HAND: Stage 1 =======
F0 = np.zeros(n)
parentA, leftA, rightA, LA, RA = split_stats(F0,"A")
GA0,HA0 = leftA; GA1,HA1 = rightA
wA0 = leaf_weight(GA0,HA0,lam)   # expect -0.5
wA1 = leaf_weight(GA1,HA1,lam)   # expect +0.5
gainA_half = gain(parentA,leftA,rightA,lam,gamma,half=True)   # expect 0.5
gainA_full = gain(parentA,leftA,rightA,lam,gamma,half=False)  # expect 1.0

# Alternative split B
parentB, leftB, rightB, LB, RB = split_stats(F0,"B")
gainB_half = gain(parentB,leftB,rightB,lam,gamma,half=True)   # expect 0.0

F1_hand = np.where(A==1, lr*wA1, lr*wA0)
p1_hand = sigmoid(F1_hand)

print("=== HAND (Stage 1) ===")
print(f"A=1: G={GA1:.6f}, H={HA1:.6f}, w*={wA1:.6f}, Score={score(GA1,HA1,lam):.6f}")
print(f"A=0: G={GA0:.6f}, H={HA0:.6f}, w*={wA0:.6f}, Score={score(GA0,HA0,lam):.6f}")
print(f"Gain(A)  = {gainA_half:.6f}  (ours, 1/2 factor)")
print(f"Gain(A)' = {gainA_full:.6f}  (LightGBM's reporting, no 1/2)")
print(f"Gain(B)  = {gainB_half:.6f}")
print()

# ======= HAND: Stage 2 (using the same split on A) =======
parent2, left2, right2, L2, R2 = split_stats(F1_hand,"A")
G0,H0 = left2; G1,H1 = right2
w2_A0 = leaf_weight(G0,H0,lam)   # expect about -0.262991
w2_A1 = leaf_weight(G1,H1,lam)   # expect about +0.262991

print("=== HAND (Stage 2) ===")
print(f"A=1: G={G1:.6f}, H={H1:.6f}, w2*={w2_A1:.6f}")
print(f"A=0: G={G0:.6f}, H={H0:.6f}, w2*={w2_A0:.6f}")
print()

hand_df = pd.DataFrame({
    "ID": np.arange(1,n+1),
    "A":A,"B":B,"y":y.astype(int),
    "F0":F0, "F1_hand":F1_hand, "p1_hand":p1_hand
```

```
})

# ======= LightGBM: Stage 1 & Stage 2 =======
try:
    import lightgbm as lgb
except Exception as e:
    raise SystemExit("Hãy cài: pip install lightgbm\nLỗi import: "+str(e))

# Train with 2 trees to extract both stages; stump each time.
lgbm = lgb.LGBMClassifier(
    objective="binary",
    n_estimators=2,        # 2 trees to see stage-2 increment
    num_leaves=2,          # stump (2 leaves)
    max_depth=1,
    learning_rate=lr,
    reg_lambda=lam,
    reg_alpha=0.0,
    min_child_samples=1,
    subsample=1.0,
    colsample_bytree=1.0,
    boosting_type="gbdt",
    importance_type="gain",
    verbosity=-1
)
# IMPORTANT: make F0 = 0 so it matches the hand setup
lgbm.set_params(boost_from_average=False)
lgbm.fit(X, y)

# Raw scores after tree 1 and tree 2
F_after_1 = lgbm.predict(X, raw_score=True, num_iteration=1)
F_after_2 = lgbm.predict(X, raw_score=True, num_iteration=2)
delta_tree2 = F_after_2 - F_after_1  # contribution of tree 2 alone

# Dump trees to read split feature, gain, leaf values
dump = lgbm.booster_.dump_model()
def tree_info(k):
    return dump["tree_info"][k]["tree_structure"]

def leaves_of(tree):
    vals = []
    def dfs(node):
        if "split_feature" in node:
            dfs(node["left_child"]); dfs(node["right_child"])
        else:
            vals.append(node["leaf_value"])
    dfs(tree); return vals

t0 = tree_info(0); t1 = tree_info(1)
root_feat_t0 = t0.get("split_feature", None)  # 0 ~ column 0 (A), 1 ~ column 1 (B)
```

```
root_gain_t0 = t0.get("split_gain", None)     # may be  1.0 (no 1/2 factor)
leaves_t0 = leaves_of(t0)

root_feat_t1 = t1.get("split_feature", None)
root_gain_t1 = t1.get("split_gain", None)
leaves_t1 = leaves_of(t1)

feat_map = {0:"A", 1:"B"}
print("=== LightGBM model summary ===")
print(f"Tree 1 root: feature={feat_map.get(root_feat_t0,'?')}, split_gain={root_gain_t0:.6f}")
print(f"Tree 1 leaves: { [round(v,6) for v in leaves_t0] }  (expect  [+0.5, -0.5])")
print(f"Tree 2 root: feature={feat_map.get(root_feat_t1,'?')}, split_gain={root_gain_t1:.6f}")
print(f"Tree 2 leaves: { [round(v,6) for v in leaves_t1] }  (expect  [+0.262991, -0.262991])")
print()

# ===== Per-sample comparison =====
comp1 = hand_df.copy()
comp1["F1_lgb"] = F_after_1
comp1["p1_lgb"] = 1/(1+np.exp(-comp1["F1_lgb"]))
comp1["dF1"] = comp1["F1_lgb"] - comp1["F1_hand"]
comp1["dp1"] = comp1["p1_lgb"] - comp1["p1_hand"]

comp2 = pd.DataFrame({
    "ID": np.arange(1,n+1),
    "A":A,"contrib_tree2_lgb": delta_tree2,
})
# Expected contributions of tree2 by hand:
exp2 = np.where(A==1, w2_A1, w2_A0)
comp2["contrib_tree2_hand"] = exp2
comp2["delta"] = comp2["contrib_tree2_lgb"] - comp2["contrib_tree2_hand"]

print("=== Per-sample comparison: Stage 1 (F1, p1) ===")
print(comp1[["ID","A","B","y","F1_hand",
"F1_lgb","dF1","p1_hand","p1_lgb","dp1"]].to_string(index=False))

print("\n=== Per-sample comparison: Tree 2 contribution (raw) ===")
print(comp2.to_string(index=False))
```

```
=== HAND (Stage 1) ===
A=1: G=-1.000000, H=1.000000, w*=0.500000, Score=0.500000
A=0: G=1.000000, H=1.000000, w*=-0.500000, Score=0.500000
Gain(A)  = 0.500000  (ours, 1/2 factor)
Gain(A)' = 1.000000  (LightGBM's reporting, no 1/2)
Gain(B)  = 0.000000

=== HAND (Stage 2) ===
A=1: G=-0.510163, H=0.940015, w2*=0.262968
A=0: G=0.510163, H=0.940015, w2*=-0.262968

=== LightGBM model summary ===
Tree 1 root: feature=A, split_gain=1.000000
Tree 1 leaves: [-0.5, 0.5]  (expect ≈ [+0.5, -0.5])
Tree 2 root: feature=A, split_gain=0.268313
Tree 2 leaves: [-0.262968, 0.262968]  (expect ≈ [+0.262991, -0.262991])
```

Hình 14: So sánh kết quả giữa tính tay và dùng sklearn với mô hình LightGBM

4 Bài toàn thực nghiệm: Phân loại ung thư vú

```python
import os
import platform
import time
import numpy as np
import pandas as pd
from sklearn.datasets import load_breast_cancer, make_classification
from sklearn.model_selection import RepeatedStratifiedKFold
from sklearn.metrics import (
    accuracy_score, balanced_accuracy_score, f1_score,
    roc_auc_score, average_precision_score, log_loss
)
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import (
    RandomForestClassifier,
    ExtraTreesClassifier,
    GradientBoostingClassifier,
    HistGradientBoostingClassifier,
    AdaBoostClassifier,
)

# Optional imports
HAS_XGB = False
HAS_LGBM = False
try:
    from xgboost import XGBClassifier
    HAS_XGB = True
except Exception:
    HAS_XGB = False

```

```
try:
    import lightgbm as lgb
    HAS_LGBM = True
except Exception:
    HAS_LGBM = False


def in_notebook() -> bool:
    """Detect Jupyter/Colab to avoid argparse reading kernel args."""
    try:
        from IPython import get_ipython  # type: ignore
        return get_ipython() is not None
    except Exception:
        return False


def get_dataset(args):
    if args.dataset == "breast_cancer":
        data = load_breast_cancer()
        X, y = data.data, data.target
        name = "breast_cancer"
    elif args.dataset == "synthetic":
        weights = None
        if args.class_weight is not None:
            weights = [args.class_weight]  # class 0 proportion
        X, y = make_classification(
            n_samples=args.n_samples,
            n_features=args.n_features,
            n_informative=args.n_informative,
            n_redundant=max(0, args.n_features - args.n_informative - 5),
            n_clusters_per_class=2,
            flip_y=0.01,
            weights=weights,
            random_state=42,
        )
        name = "synthetic"
    else:
        raise ValueError("Unknown dataset")
    return X, y, name


def get_models(args):
    # compact defaults; --fast shrinks them for quick runs
    n_est_rf  = 100 if args.fast else 200
    n_est_et  = 150 if args.fast else 300
    n_est_gbm = 100 if args.fast else 200
    n_iter_hgb= 100 if args.fast else 200
    n_est_ada = 150 if args.fast else 300
    n_est_xgb = 150 if args.fast else 300
    n_est_lgb = 150 if args.fast else 300

    models = {}
    models["DecisionTree"] = DecisionTreeClassifier(
        criterion="gini", max_depth=None, random_state=args.seed
    )
    models["RandomForest"] = RandomForestClassifier(
        n_estimators=n_est_rf, max_depth=None, n_jobs=args.n_jobs, random_state=args.seed
    )
    models["ExtraTrees"] = ExtraTreesClassifier(
        n_estimators=n_est_et, max_depth=None, n_jobs=args.n_jobs, random_state=args.seed
    )
    models["GradientBoosting"] = GradientBoostingClassifier(
```

```
        learning_rate=0.1, n_estimators=n_est_gbm, max_depth=3, random_state=args.seed
    )
    models["HistGradientBoosting"] = HistGradientBoostingClassifier(
        learning_rate=0.1, max_depth=None, max_iter=n_iter_hgb, random_state=args.seed
    )
    models["AdaBoost(stump)"] = AdaBoostClassifier(
        n_estimators=n_est_ada, learning_rate=0.5, random_state=args.seed
    )

    if HAS_XGB:
        models["XGBoost"] = XGBClassifier(
            objective="binary:logistic",
            n_estimators=n_est_xgb,
            learning_rate=0.1,
            max_depth=6,
            subsample=0.8,
            colsample_bytree=0.8,
            reg_lambda=1.0,
            reg_alpha=0.0,
            tree_method="hist",  # fast and comparable to HGB
            n_jobs=args.n_jobs,
            random_state=args.seed,
            eval_metric="logloss",
            verbosity=0,
        )

    if HAS_LGBM:
        models["LightGBM"] = lgb.LGBMClassifier(
            objective="binary",
            n_estimators=n_est_lgb,
            learning_rate=0.1,
            num_leaves=31,
            subsample=0.8,
            colsample_bytree=0.8,
            reg_lambda=1.0,
            random_state=args.seed,
            n_jobs=args.n_jobs,
            boosting_type="gbdt",
            verbose=-1,
        )
    return models


def proba_or_scores(model, X):
    """Return probability for class 1 if available; else decision_function; else label."""
    if hasattr(model, "predict_proba"):
        p = model.predict_proba(X)
        if p.ndim == 2 and p.shape[1] == 2:
            return p[:, 1]
        return np.argmax(p, axis=1)
    if hasattr(model, "decision_function"):
        s = model.decision_function(X)
        return s.ravel() if s.ndim > 1 else s
    return model.predict(X)


def run_benchmark(args):
    X, y, ds_name = get_dataset(args)
    models = get_models(args)

    rskf = RepeatedStratifiedKFold(
        n_splits=args.splits, n_repeats=args.repeats, random_state=args.seed
```

```
    )

    rows = []
    fold_idx = 0
    for train_idx, test_idx in rskf.split(X, y):
        fold_idx += 1
        X_train, X_test = X[train_idx], X[test_idx]
        y_train, y_test = y[train_idx], y[test_idx]

        for name, model in models.items():
            if hasattr(model, "random_state"):
                setattr(model, "random_state", args.seed + fold_idx)

            # Fit timing
            t0 = time.perf_counter()
            model.fit(X_train, y_train)
            fit_time = time.perf_counter() - t0

            # Predict timing
            t1 = time.perf_counter()
            y_pred = model.predict(X_test)
            pred_time = time.perf_counter() - t1

            # Scores
            scores = proba_or_scores(model, X_test)
            acc = accuracy_score(y_test, y_pred)
            bacc = balanced_accuracy_score(y_test, y_pred)
            f1 = f1_score(y_test, y_pred)

            # ROC-AUC & PR-AUC
            try:
                auc = roc_auc_score(y_test, scores)
            except Exception:
                auc = np.nan
            try:
                prauc = average_precision_score(y_test, scores)
            except Exception:
                prauc = np.nan

            # Log loss (chỉ khi có xác suất)
            ll = np.nan
            try:
                if scores.ndim == 1 and np.all((scores >= 0) & (scores <= 1)):
                    ll = log_loss(y_test, scores)
                elif hasattr(model, "predict_proba"):
                    p = model.predict_proba(X_test)[:, 1]
                    ll = log_loss(y_test, p)
            except Exception:
                pass

            rows.append(
                {
                    "dataset": ds_name,
                    "model": name,
                    "fold": fold_idx,
                    "fit_time_sec": fit_time,
                    "pred_time_sec": pred_time,
                    "accuracy": acc,
                    "balanced_accuracy": bacc,
                    "f1": f1,
                    "roc_auc": auc,
                    "pr_auc": prauc,
```

```
                    "log_loss": ll,
                }
            )

    df = pd.DataFrame(rows)
    # Aggregate
    agg = (
        df.groupby(["dataset", "model"])
        .agg(
            mean_fit_time=("fit_time_sec", "mean"),
            mean_pred_time=("pred_time_sec", "mean"),
            mean_accuracy=("accuracy", "mean"),
            std_accuracy=("accuracy", "std"),
            mean_bal_accuracy=("balanced_accuracy", "mean"),
            mean_f1=("f1", "mean"),
            mean_roc_auc=("roc_auc", "mean"),
            mean_pr_auc=("pr_auc", "mean"),
            mean_log_loss=("log_loss", "mean"),
            folds=("fold", "count"),
        )
        .reset_index()
        .sort_values(["mean_roc_auc", "mean_accuracy"], ascending=False)
    )
    return df, agg


def build_parser():
    p = argparse.ArgumentParser(
        description="Benchmark tree-based models (runtime & accuracy)."
    )
    p.add_argument("--dataset", type=str, default="breast_cancer",
                   choices=["breast_cancer", "synthetic"])
    p.add_argument("--splits", type=int, default=5)
    p.add_argument("--repeats", type=int, default=2)
    p.add_argument("--plot", action="store_true")
    p.add_argument("--fast", action="store_true")
    p.add_argument("--n-jobs", type=int, default=os.cpu_count())
    p.add_argument("--seed", type=int, default=42)
    # synthetic options
    p.add_argument("--n-samples", type=int, default=3000)
    p.add_argument("--n-features", type=int, default=30)
    p.add_argument("--n-informative", type=int, default=10)
    p.add_argument("--class-weight", type=float, default=None,
                   help="Proportion for class 0 (e.g., 0.3). If None, balanced.")
    return p


def in_notebook():
    try:
        from IPython import get_ipython
        return get_ipython() is not None
    except Exception:
        return False


def main():
    parser = build_parser()
    argv = [] if in_notebook() else None
    args = parser.parse_args(argv)

    print(f"Python {platform.python_version()} | OS: {platform.system()} | CPU cores: {os.cpu_count()}")

```

```
    df, agg = run_benchmark(args)
    print("\n=== Fold-level results (head) ===")
    print(df.head(10).to_string(index=False))
    print("\n=== Aggregated results ===")
    print(agg.to_string(index=False))

    df.to_csv("tree_bench_folds.csv", index=False)
    agg.to_csv("tree_bench_summary.csv", index=False)
    print("\nSaved: tree_bench_folds.csv, tree_bench_summary.csv")


if __name__ == "__main__":
    main()
```

```
=== Fold-level results (head) ===
      dataset               model  fold  fit_time_sec  pred_time_sec  accuracy  balanced_accuracy       f1  roc_auc   pr_auc  log_loss
breast_cancer        DecisionTree     1      0.078539       0.003721  0.921053           0.922863 0.935252 0.922863 0.927735  2.845552
breast_cancer        RandomForest     1      0.984219       0.084305  0.964912           0.967245 0.971429 0.997707 0.998593  0.098857
breast_cancer          ExtraTrees     1      0.674562       0.091990  0.991228           0.992958 0.992908 0.999672 0.999804  0.085850
breast_cancer    GradientBoosting     1      2.615084       0.006107  0.973684           0.969702 0.979021 0.998362 0.999038  0.067550
breast_cancer HistGradientBoosting     1      1.030585       0.006439  0.982456           0.981330 0.985915 0.999672 0.999804  0.022512
breast_cancer     AdaBoost(stump)     1      2.011278       0.045584  0.991228           0.988372 0.993007 1.000000 1.000000  0.552937
breast_cancer             XGBoost     1      0.333381       0.006167  0.991228           0.992958 0.992908 1.000000 1.000000  0.047846
breast_cancer            LightGBM     1      0.181239       0.004019  0.991228           0.988372 0.993007 0.999345 0.999606  0.033309
breast_cancer        DecisionTree     2      0.024677       0.000793  0.885965           0.867180 0.911565 0.867180 0.867000  4.110241
breast_cancer        RandomForest     2      0.477936       0.062151  0.929825           0.916148 0.945205 0.975434 0.971198  0.425668

=== Aggregated results ===
      dataset               model  mean_fit_time  mean_pred_time  mean_accuracy  std_accuracy  mean_bal_accuracy  mean_f1  mean_roc_auc  mean_pr_auc  mean_log_loss  folds
breast_cancer     AdaBoost(stump)       1.720984        0.035908       0.972768      0.012030           0.967724 0.978569      0.994673     0.996543       0.529487     10
breast_cancer             XGBoost       0.215526        0.002526       0.967482      0.012455           0.964491 0.974178      0.994139     0.996183       0.090537     10
breast_cancer            LightGBM       0.160677        0.003123       0.971014      0.013089           0.965920 0.977153      0.993843     0.996014       0.100022     10
breast_cancer          ExtraTrees       0.518019        0.082618       0.965751      0.015133           0.960752 0.972987      0.993762     0.995951       0.111929     10
breast_cancer    GradientBoosting       2.244938        0.001831       0.956955      0.022444           0.951371 0.966002      0.992124     0.994825       0.152960     10
breast_cancer HistGradientBoosting       0.950298        0.004356       0.960456      0.019065           0.952728 0.969091      0.991982     0.994641       0.173059     10
breast_cancer        RandomForest       0.531048        0.054241       0.954293      0.016519           0.949699 0.963757      0.989150     0.989961       0.176401     10
breast_cancer        DecisionTree       0.033257        0.000928       0.921751      0.027438           0.918018 0.937486      0.918018     0.921805       2.820370     10
```

Hình 15: So sánh hiệu quả hoạt động của toàn bộ mô hình cây

Kết quả trung bình trên 10 folds cho thấy:

- **Độ chính xác (Accuracy & ROC-AUC):**
  - Các mô hình boosting hiện đại như **XGBoost** (ROC-AUC ≈ 0.994) và **LightGBM** (ROC-AUC ≈ 0.996) đạt kết quả gần như hoàn hảo.
  - **AdaBoost** cũng cho ROC-AUC ≈ 0.995, vượt trội so với Gradient Boosting truyền thống.
  - **Random Forest** và **Extra Trees** duy trì hiệu quả tốt (ROC-AUC ≈ 0.99).
  - **Decision Tree** kém ổn định hơn với ROC-AUC ≈ 0.918.
- **Thời gian huấn luyện:**
  - **Decision Tree** nhanh nhất (≈ 0.03s) nhưng độ chính xác thấp hơn.
  - **XGBoost** và **LightGBM** cân bằng: vừa nhanh (XGBoost ≈ 0.2s, LightGBM ≈ 0.16s) vừa cực kỳ chính xác.
  - **Gradient Boosting** truyền thống khá chậm (≈ 2.24s).
  - **AdaBoost** mất nhiều thời gian nhất (≈ 1.72s).
- **Log Loss:**
  - **LightGBM** có log loss thấp nhất (≈ 0.10), thể hiện xác suất dự đoán được calibrate tốt.
  - **Decision Tree** có log loss cao nhất (≈ 2.82), cho thấy xác suất dự đoán kém ổn định.

Tổng kết so sánh

- **Decision Tree:** đơn giản, nhanh, dễ hiểu nhưng không phù hợp khi yêu cầu độ chính xác cao.
- **Random Forest & Extra Trees:** cho kết quả ổn định, tuy nhiên chưa bằng Boosting hiện đại.
- **Gradient Boosting cổ điển:** đạt ROC-AUC ≈ 0.992, nhưng tốc độ chậm và kém tối ưu hơn.
- **HistGradientBoosting:** cải thiện tốc độ nhờ sử dụng histogram, kết quả gần tương đương LightGBM.
- **AdaBoost:** mạnh khi dữ liệu ít nhiễu, song mất thời gian train lâu hơn.
- **XGBoost & LightGBM:** nổi bật nhất, cân bằng giữa hiệu năng và hiệu quả, phù hợp với dữ liệu lớn.