Thuật toán XGBoost

Hồ Nguyễn Phú

23 tháng 09, 2025

I. Ôn lại Gradient Boost:

1. Hàm mục tiêu:

Gradient Boosting hướng đến tối ưu hóa hàm mục tiêu bên dưới (nguồn). Trong đó, ta lấy dự đoán là tổng có trọng số các weak–learner ($\sum_{n=1}^{N} c_n w_n$). Sau đó ta đưa dự đoán vào trong hàm loss L.

$$min(L(y, \sum_{n=1}^{N} c_n w_n))$$

Với Regression, ta dùng Least Mean Square Error (LMSE):

$$LMSE = \frac{1}{N} \sum_{n=1}^{N} (\hat{y}_n - y_n)^2$$

Với Classification, Gradient Boosting sử dụng hàm Negative Log–Likelihood:

$$L = -\sum_{n=1}^{N} [\, y_n log(\hat{y}_n) + (1 - y_n) log(1 - \hat{y}_n) \,]$$

2. Thuật toán:

a. Bước 1: Khởi tạo cây đầu tiên $m = 0$, với dự đoán $F_0(x)$

Ta khởi tạo cây đầu tiên với một giá trị hằng số cho tất cả dự đoán. Với Regression, giá trị hằng số được chọn sao cho LMSE là nhỏ nhất. LMSE nhỏ nhất nhỏ nhất khi $\hat{y} = \frac{1}{N} \sum_{n=1}^{N} y_n$, tức giá trị trung bình của các nhãn. Do đó, ta đặt $F_0(x)$ bằng giá trị trung bình (với classification, $F_0(x) = 0$ dẫn đến dự đoán đầu tiên là)

b. Bước 2: Tính toán Residuals r_i giữa nhãn thật và kết quả dự đoán:

$$r_i = y_i - \hat{y}_{i,m-1}$$

c. Bước 3: Lặp lại quá trình sau cho mỗi cây m

- Với Regression, fit cây m trên r để huấn luyện "hàm" dự đoán residual $h_m(x)$ sao cho:
  - $h_m(x)$ được nhân với tốc độ học / shrinkage η.
  - $\eta h_m(x)$ cộng với $F_{m-1}(x)$ sẽ cho ra dự đoán mới tại m (tương tự với **Gradient Descent**).

$$F_m(x) = F_{m-1}(x) + \eta h_m(x)$$

- Với Classification:
  - Mỗi node lá j với K mẫu sẽ trả về công thức sau (dựa trên dự đoán cây trước đó):

$$\gamma_j = \frac{\sum_{k=1}^{K} r_k}{\sum_{k=1}^{K} p_k(1-p_k)}$$

  - Ta dùng γ_j để cập nhật dự đoán class k ở node j như sau:

$$F_m^{(k)} = F_{m-1}^{(k)} + \eta\gamma_j$$

3. Về log–odds:

Trong classification Gradient Boost, ta sẽ không dự đoán trực tiếp xác suất của lớp, mà dự đoán log–odds (logit). Thuật toán Gradient Boosting nói riêng và Boosting nói chung mang tính chất cộng (additive), do đó các giá trị xác suất cần được biểu diễn trong miền $(-\infty; \infty)$ để xử lý trước khi đưa về $[0; 1]$. Để thực hiện điều này, ta chuyển giá trị xác suất thành log–odds thông qua công thức:

$$h(x) = log\ odds = log(\frac{p}{1-p})$$

Lúc này, $h(x)$ nằm trong miền $(-\infty; \infty)$ như mong muốn. Ta có thể thực hiện phép tổng như khi cập nhật mô hình Regression với residual. Sau khi có kết quả $\sum_{n=1}^{N} h(x_n)$, ta sử dụng hàm sigmoid $\sigma(h) = \frac{1}{1+e^{-h}}$ để ánh xạ sang miền xác suất.

II. Thuật toán XGBoost:

XGBoost là một thuật toán Tree Boosting. Thuật toán được đề cập trong bài báo XGBoost: A Scalable Tree Boosting System (Chen & Guestrin, 2016). Điểm nổi bật: nâng cao hiệu suất tính toán (nhanh gấp 10 lần các thuật toán cùng thời) thông qua cải tiến thuật toán.

1. Thuật toán Tree Boosting (Tree Boosting in a nutshell):

a. Regularized (chính quy hóa) learning objective:

Sơ lược về bối cảnh, chúng ta có L1 và L2 regularization – với L1 regularization đưa các giá trị trọng số về 0 (sparse representation) còn L2 regularization đưa các giá trị trọng số về **gần** 0 (nhưng không hoàn toàn về 0). Ngoài ra, ta cũng thường thực hiện chính quy hóa thông qua số lá *T* để làm giảm độ phức tạp của cây. Các giá trị trên được cộng vào hàm loss.

Một regularized objective như sau:

$$\mathcal{L}(\phi) = \sum_i l(\hat{y}_i, y_i) + \sum_k \Omega(f_k)$$
$$\text{where } \Omega(f) = \gamma T + \frac{1}{2}\lambda \|w\|^2$$

Trong đó:

- l – hàm loss lồi khả vi (differentiable convex loss function)
- $\Omega(f_k)$ – phạt độ phức tạp của cây thứ k
- γ – càng lớn thì cây sẽ "khó" split (giảm số lá)
- λ – càng lớn thì trọng số lá sẽ nhỏ hơn

b. Gradient tree boosting:

Gần tương tự như Gradient Boosting, ta tính toán loss $L^{(k)}$ của cây thứ k thông qua hàm:

$$\mathcal{L}^{(t)} = \sum_{i=1}^{n} l(y_i, \hat{y_i}^{(t-1)} + f_t(\mathbf{x}_i)) + \Omega(f_t)$$

Để tăng tốc độ tính toán, ta thực hiện xấp xỉ Taylor với hàm trên. Thông qua xử lý, ta có:

$$\tilde{\mathcal{L}}^{(t)}(q) = -\frac{1}{2}\sum_{j=1}^{T} \frac{(\sum_{i\in I_j} g_i)^2}{\sum_{i\in I_j} h_i + \lambda} + \gamma T. \quad (1)$$

Trong đó:

- g: đạo hàm bậc 1 (gradient) của hàm mất mát
- h: đạo hàm bậc 2 (hessian) của hàm mất mát
- λ: hệ số gắn với regularizer theo số lá
- q: cấu trúc cây q

Tuy nhiên, ta không thể xét đến tất cả cấu trúc cây khả thi của q, do đó ta dùng thuật toán split greedy. Trong đó, tại mỗi node lá chuẩn bị split, ta có công thức tính hàm loss sau khi split như sau (L là nút con trái; R là nút con phải):

$$\mathcal{L}_{split} = \frac{1}{2}\left[\frac{(\sum_{i\in I_L} g_i)^2}{\sum_{i\in I_L} h_i + \lambda} + \frac{(\sum_{i\in I_R} g_i)^2}{\sum_{i\in I_R} h_i + \lambda} - \frac{(\sum_{i\in I} g_i)^2}{\sum_{i\in I} h_i + \lambda}\right] - \gamma \quad (2)$$

Theo tài liệu đọc thêm của AIO, nếu $L_{split} > 0$, lợi ích thu được từ phân nhánh sẽ cao hơn chi phí γ, ta sẽ thực hiện split; ngược lại, $L_{split} \leq 0$, thì cây không split. L_{split} ở đây tương đương với Gain trong tài liệu.

c. Shrinkage và Column subsampling:

Bên cạnh các phương pháp trên, tác giả còn giới thiệu hai biện pháp chống overfitting là: **shrinkage** và **column (feature) subsampling**.

Phương pháp thứ nhất bao gồm việc sử dụng η như learning rate trong stochastic optimization. Còn phương pháp thứ hai ta đã được làm quen trong Random Forest – thực hiện subsampling (lấy mẫu con) trên các cột, tức chọn ra những feature ngẫu nhiên làm dữ liệu huấn luyện cho mô hình.

2. Thuật toán tìm kiếm split thích hợp (Split Finding Algorithm):

Nhìn chung, mục này bàn về thách thức và phương pháp split dựa trên bách phân vị, cũng như sơ lược về thuật toán Sparsity-aware Split Finding để xử lý dữ liệu ma trận thưa (sparse matrix).

a. Overview:

"Exact greedy algorithm" tìm cách tối ưu hóa hàm (2) bên trên bằng cách duyệt tất cả split khả thi. Cách này kém hiệu quả, và sẽ gặp vấn đề khi data không fit vào trong bộ nhớ. Do đó, ta cần một thuật toán xấp xỉ.

Thuật toán như sau: (1) đề xuất các điểm ứng viên (*candidate points*) dựa trên percentile của phân phối dữ liệu; (2) ánh xạ các giá trị liên tục thành các buckets dựa trên các ứng viên; (3) tổng hợp (aggregate) dữ liệu và tìm ra giải pháp tốt nhất giữa những dữ liệu đã được nhóm lại.

Thuật toán có 2 biến thể: biến thể toàn cục (global variant) và biến thể cục bộ (local variant). Với **biến thể toàn cục**, ta sử dụng chung các *candidate points* ban đầu (có được từ quá trình tạo cây ban đầu) cho cả những split sau đó. Còn với **biến thể cục bộ**, thuật toán đề xuất *candidate points* mới sau mỗi split. Bảng so sánh hai biến thể:

Global proposal	**Local proposal**
Cần ít bước đề xuất hơn	Cần nhiều bước đề xuất
Candidate points không được refined	*Candidate points* được refined
Cần nhiều *candidate points* (đạt đổ chính xác tương đương Local proposal nếu đủ *candidate points*)	Cần ít *candidate points*

b. Weighted quantile sketch:

Một bước quan trọng của thuật toán xấp xỉ là bước đề xuất candidate split points. Trong thực tế, việc tìm kiếm *candidate points* là không đơn giản. Nhiều thuật toán đương thời lại chỉ thực hiện sắp xếp các phần tử con trong tập dữ liệu nhỏ, và không có một nền tảng Toán học vững chắc. Do đó, XGBoost đề xuất Weighted Quantile Sketch (Phác thảo Bách phân vị có trọng số).

Ý tưởng chính của Weighted Quantile Sketch là đề xuất một cấu trúc dữ liệu . Phương pháp được đề cập / chứng minh sâu hơn trong phần Appendix A của bài báo.

Sơ lược: Thuật toán ứng dụng cấu trúc dữ liệu "*quantile summary*" cho bài toán tìm bách phân vị. Tác giả cải tiến kiểu dữ liệu này để hỗ trợ phép merge and prune trên dữ liệu có trọng số (weighted data) – nhằm generalize cho trường hợp của Tree Boosting.

c. Sparsity-aware Split Finding

Để giải quyết vấn đề dữ liệu thưa ("missing data, "zero entries", "feature engineering" artifacts), tác giả đề xuất sử dụng default direction (hướng mặc định). Default direction bao gồm việc đưa tất cả các mẫu dữ liệu thiếu qua một bên phải hoặc trái của split, thay vì xét từng phần tử. Điều này giúp giảm độ phức tạp tính toán và cho phép cây học được pattern của dữ liệu thưa. Ta sẽ bàn chi tiết hơn về các bước của Sparsity-aware Split Finding (xem thêm: Algorithm 3 của paper).

Ta đặt I_k làm tập hợp các dữ liệu của node hiện tại, trong đó feature k của mỗi mẫu x_i không bị thiếu.

Đầu tiên, tính L tương tự bên trên. Nhưng thay vì chỉ xét L một lần cho cả hai split trái (left) và phải (right), ta chia thành 2 trường hợp: khi dữ liệu thiếu nằm hết bên trái hoặc bên phải. Lúc này ta chọn trường hợp đem lại Gain (kết hợp giữa các mẫu I_k với các mẫu dữ liệu thiếu, nếu đang xét) cao nhất, và chọn kết quả bên trái / phải tương ứng làm default direction.

Kết quả, XGBoost với Sparsity-aware Split có thể xử lý dữ liệu thiếu một cách thống nhất ("*unified*"), đồng thời đem lại độ phức tạp tuyến tính (với số mẫu không thiếu). Ngoài ra, thuật toán cũng tăng tốc gấp 50 lần (!) so với bản gốc (naive version).

3. System design:

Các biện pháp được thực hiện để cải thiện hiệu suất tính toán:

- Column Block for Parallel Learning
- Cache-aware Access
- Blocks for Out-of-core Computation.

III. Code:

1. Huấn luyện mô hình:

Tương tự như các mô hình trước (Decision Tree, Random Forest, Ada Boosting), chúng ta có thể tận dụng pipeline gắn với thư viện Sklearn và các mô hình có sẵn. Tuy nhiên, để sử dụng XGBoost thì ta cần import thư viện xgboost:

```
import xgboost as xgb
```

Sau đó gọi class XGBClassifier (cho classification) và xử lý như các mô hình Sklearn:

```
xg_boost_model = xgb.XGBClassifier()
```

Bên cạnh đó, ta cũng có thể tận dụng API train() của thư viện *xgboost* với các tính năng như: parameters, save, load mô hình, etc. Phần này được thực hiện ở mục 3. của Notebook trong trang Github này.

Một số thao tác phân tích dữ liệu nâng cao hơn như Grid Search, K–Fold và Error bar được đề cập trong bài viết của Vititech (đã thêm vào repo Github ở mục 4.).

2. Inference:

Khi inference với XGBClassifier, ta thực hiện tương tự với các mô hình Sklearn khác. Tuy nhiên, với xgboost.train(), có thể ta cần kiểm tra lại shape của output để đảm bảo phù hợp với pipeline có sẵn.

Tài liệu tham khảo:

Bài báo giải thích về XGBoost

Bài báo gốc XGBoost: A scalable Tree Boosting System

Blog trên Toward Data Science

Các video về Gradient Boosting và XGBoost của StatQuest

Document thư viện XGBoost, API Python

Tài liệu đọc thêm của AI Việt Nam 2025 về XGBoost (Step-by-Step: XGBoost)

XGBoost – Bài 14: Tuning Subsample