Phương pháp chọn coreset sử dụng các phương pháp tối ưu

Ngày 15 tháng 4 năm 2025

1 Tổng quan về Coreset

1.1 Khái niệm Coreset và ứng dụng

1.1.1 Khái niệm Coreset

Coreset là một tập hợp con nhỏ của các điểm dữ liệu gốc, thường đi kèm với trọng số, sao cho nó xấp xỉ hóa một cách hiệu quả các thuộc tính quan trọng của toàn bộ tập dữ liệu ban đầu đối với một hàm chi phí và một tập hợp các truy vấn cụ thể. Nói một cách đơn giản, coreset là một phiên bản thu nhỏ, có trọng số của dữ liệu lớn, cho phép chúng ta đạt được hiệu suất tương đương hoặc gần tương đương như khi sử dụng toàn bộ dữ liệu, nhưng với chi phí tính toán thấp hơn. Coreset có thể là một cấu trúc dữ liệu nhỏ cho phép xấp xỉ tổng khoảng cách có trọng số, khoảng cách tối đa có trọng số, hoặc nói chung là bất kỳ hàm chi phí nào dựa trên các khoảng cách này.

1.1.2 Ứng dụng của Coreset

Mục tiêu chung của việc sử dụng coreset là giảm kích thước dữ liệu cần xử lý, từ đó tiết kiệm thời gian, bộ nhớ và chi phí tính toán. Coreset là một kỹ thuật nén dữ liệu mạnh mẽ với nhiều ứng dụng trong học máy và các lĩnh vực liên quan. Dưới đây là một số ứng dụng chính được đề cập trong các nguồn tài liệu:

- Học máy hiệu quả (Efficient Machine Learning): Coreset được sử dụng để chọn một tập hợp con có trọng số của dữ liệu huấn luyện, giúp tăng tốc độ huấn luyện các mô hình học máy mà vẫn duy trì được hiệu suất tương đương. Điều này đặc biệt hữu ích khi làm việc với các tập dữ liệu lớn.

- Huấn luyện mạng nơ-ron (Neural Network Training): Coreset có thể được áp dụng để chọn các tập dữ liệu con hiệu quả cho việc huấn luyện mạng nơ-ron, bao gồm cả trong các thiết lập học liên tục (continual learning). Một phương pháp cụ thể là GMC (Gradient Matching Coreset) đã được chứng minh là hiệu quả cho việc chọn coreset từ các lô dữ liệu tuần tự không độc lập và có phân phối giống nhau (non-iid).

- Giảm chi phí tính toán cho dò tìm siêu tham số (Compute-efficient Hyperparameter Tuning): Các kỹ thuật lựa chọn tập con dữ liệu dựa trên gradient (liên quan đến coreset) có thể được sử dụng để đánh giá các cấu hình siêu tham số một cách hiệu quả hơn.

- Học bán giám sát (Semi-supervised Learning): Framework RETRIEVE đề xuất sử dụng coreset để chọn một tập con không nhãn hiệu hiệu quả, khi kết hợp với dữ liệu có nhãn, sẽ giúp huấn luyện mô hình tốt hơn.

- Các bài toán tối ưu hóa (Optimization Problems): Coreset được áp dụng rộng rãi cho nhiều bài toán tối ưu hóa trong học máy, bao gồm:
  - Phân cụm (Clustering): k-means, k-median.
  - Hồi quy Logistic (Logistic Regression).
  - Phương pháp Bayesian (Bayesian Methods).
  - Hồi quy tuyến tính (Linear Regression).
  - Tối ưu hóa mạnh mẽ (Robust Optimization).
  - Mô hình hỗn hợp Gaussian (Gaussian Mixture Model).
  - Học tích cực (Active Learning).

- Xử lý dữ liệu lớn và dữ liệu theo thời gian thực (Big Data and Real-time Data): Các coreset có khả năng kết hợp (composable coresets) đặc biệt quan trọng trong việc xử lý dữ liệu lớn hoặc dữ liệu đến liên tục, cho phép xây dựng coreset song song và đệ quy.

- Trả lời truy vấn hiệu quả (Efficient Query Answering): Coreset có thể được sử dụng để trả lời các truy vấn SQL một cách nhanh chóng và hiệu quả hơn trên dữ liệu đã nén.

- Lựa chọn mô hình và giảm đặc trưng (Model Selection and Feature Reduction).

- Các ứng dụng cụ thể khác: Ước tính tư thế (pose-estimation) trong hệ thống theo dõi, tạo tóm tắt trực quan (visual precis generation), và các

bài toán liên quan đến hàm đơn điệu (monotonic functions) trong học sâu.

- Tăng tốc các thuật toán hiện có (Speeding up Existing Algorithms) bằng cách làm việc trên một tập dữ liệu nhỏ hơn.

Nhìn chung, coreset là một công cụ linh hoạt và hiệu quả để giải quyết các vấn đề về hiệu suất và khả năng mở rộng trong học máy và phân tích dữ liệu. Các phương pháp xây dựng coreset khác nhau (ví dụ: dựa trên gradient, dựa trên độ nhạy, dựa trên phân cụm) được phát triển để phù hợp với các loại bài toán và mục tiêu tối ưu hóa khác nhau.

1.2 Định nghĩa coreset

Thuật ngữ **coreset** lần đầu tiên được đưa ra lần đầu bởi P. Agarwal, Har-Peled, & Varadarajan năm 2005. Ban đầu, nó được sử dụng để giải quyết các bài toán tìm k quả cầu nhỏ nhất bao phủ một tập hợp các điểm đầu vào. Ngày nay, có nhiều định nghĩa không nhất quán về coreset và các kỹ thuật giảm dữ liệu. Để định nghĩa một coreset một cách tổng quát, trước tiên cần có khái niệm về không gian truy vấn (query space). Một không gian truy vấn là một bộ dữ liệu (P, w, X, f), trong đó:

- P là một tập hợp đầu vào (thường là hữu hạn với kích thước $|P| = n$), được gọi là các điểm.
- X là một tập hợp (thường là vô hạn) các truy vấn (mô hình, hình dạng, bộ phân loại, giả thuyết).
- $w : P \to \mathbb{R}$ là một hàm trọng số (thường là dương), gán tầm quan trọng cho mỗi điểm.
- $f : P \times X \to \mathbb{R}$ là một hàm giả khoảng cách (pseudo-distance), hoặc đơn giản là khoảng cách, từ p đến x; thường là không âm. Tổng quát hơn, nó có thể được xem như một hàm mất mát. Đối với bài toán tối ưu hóa cực đại, nó có thể được sử dụng như một điểm số (trong đó điểm số cao hơn thì tốt hơn).

Coreset (C) được định nghĩa là một cấu trúc dữ liệu nhỏ cho phép chúng ta xấp xỉ:

- Tổng khoảng cách có trọng số: $\sum_{p \in P} w(p) f(p, x)$
- Khoảng cách tối đa có trọng số: $\max_{p \in P} w(p) f(p, x)$, có liên quan đến các bài toán phủ

- Tổng quát: bất kỳ hàm chi phí (cost function) cost : $\mathbb{R}^n \to \mathbb{R}$ nào ánh xạ $n = |P|$ khoảng cách $(f(p, x))_{p \in P}$ thành một chi phí tổng (thường là không âm).

Các định nghĩa chính xác cho "nhỏ", "cấu trúc dữ liệu" và "xấp xỉ" thay đổi tùy theo từng bài toán..

Một loại coreset quan trọng, đặc biệt liên quan đến việc xử lý dữ liệu lớn hoặc dữ liệu thời gian thực, là composable coresets. Tính chất này có nghĩa là hợp của một cặp coresets $C_1 \cup C_2$ là một coreset cho dữ liệu đầu vào cơ sở $P_1 \cup P_2$, và chúng ta có thể tính toán lại coreset C_3 cho $C_1 \cup C_2$ một cách đệ quy. Đặc biệt, strong và weak coresets là composable.

Tuy nhiên hầu hết các coresets hiện có có thể được định nghĩa chính thức như một ϵ-sample. Cho một không gian truy vấn (P, w, X, f) và một sai số $\epsilon \in (0, 1)$ (thường là hằng số), một ϵ-sample là một không gian truy vấn (C, u, X, f) với $C \subseteq P$ và $u : C \to \mathbb{R}$ là một hàm trọng số gán cho mỗi điểm trong C, sao cho chi phí trên dữ liệu gốc

$$\text{cost}((w(p)f(p, x))_{p \in P}) = \sum_{p \in P} w(p)f(p, x)$$

được xấp xỉ bởi chi phí trên coreset

$$\text{cost}((u(q)f(q, x))_{q \in C}) = \sum_{q \in C} u(q)f(q, x)$$

cho mọi truy vấn $x \in X$, với sai số cộng không quá ϵ, tức là:

$$|\sum_{p \in P} w(p)f(p, x) - \sum_{q \in C} u(q)f(q, x)| \leq \epsilon.$$

Do đó, C là một weighted subset strong coreset, và cũng là một composable coreset.

Khi không thể tìm thấy coreset cho một vấn đề, định nghĩa của coreset có thể được thay đổi. Tóm lại, một coreset là một bản tóm tắt dữ liệu nhỏ của dữ liệu lớn, sao cho mọi truy vấn có thể có đều cho ra kết quả xấp xỉ tương tự trên cả hai tập dữ liệu. Các coresets có thể là các tập con có trọng số của dữ liệu gốc hoặc các cấu trúc dữ liệu phức tạp hơn, tùy thuộc vào bài toán và yêu cầu về độ chính xác cũng như hiệu quả tính toán. Mục tiêu chính là đạt được sự đánh đổi có thể chứng minh giữa kích thước của coreset và sai số xấp xỉ.

2 Một số phương pháp sử dụng gradient để chọn coreset

Có một số phương pháp sử dụng gradient để chọn coreset, tập trung vào việc chọn một tập hợp con dữ liệu huấn luyện nhỏ hơn nhưng vẫn giữ được thông tin quan trọng cho việc huấn luyện mô hình. Mục tiêu chung là ước tính gradient của toàn bộ tập dữ liệu một cách chính xác bằng cách sử dụng gradient của coreset, từ đó tăng tốc quá trình huấn luyện mà không làm giảm đáng kể hiệu suất của mô hình. Dưới đây là một số phương pháp.

2.1 CRAIG (CoResets for Accelerating Incremental Gradient descent)

Phương pháp này trực tiếp xấp xỉ gradient của toàn bộ tập dữ liệu bằng cách chọn một tập hợp con có trọng số sao cho nó đại diện cho gradient đầy đủ. CRAIG chọn một tập hợp con S bằng cách tối đa hóa một hàm vị trí cơ sở (submodular facility location function), mà theo chứng minh, sẽ giảm thiểu một giới hạn trên cho sai số ước tính của gradient đầy đủ.

- Việc chọn tập hợp con được thực hiện một cách tham lam (greedy algorithm), mang lại hiệu quả tính toán.
- Một lợi ích khác là tập hợp S được tạo ra một cách tăng dần, tạo ra một thứ tự tự nhiên cho dữ liệu trong S. Việc xử lý dữ liệu theo thứ tự này giúp tăng tốc độ hội tụ của phương pháp incremental gradient.
- Đối với các bài toán lồi (convex), CRAIG có thể được sử dụng như một bước tiền xử lý trước khi bắt đầu tối ưu hóa.
- Đối với mạng nơ-ron sâu (non-convex), việc áp dụng trực tiếp bất đẳng thức để giới hạn sự khác biệt giữa các gradient trở nên khó khăn. Do đó, CRAIG có thể cần cập nhật tập hợp con sau một số lần cập nhật tham số trong quá trình huấn luyện. Phương pháp này dựa trên việc giới hạn sự khác biệt gradient thông qua gradient của lớp cuối cùng của mạng.
- CRAIG đảm bảo rằng việc chạy các phương pháp incremental gradient trên coreset sẽ hội tụ đến giải pháp (gần) tối ưu trong cùng số lượng epochs như khi chạy trên toàn bộ dữ liệu, nhưng với tốc độ nhanh hơn do chỉ tính toán gradient trên tập hợp con S.

2.2 Gradient Matching Coresets (GMC)

Ý tưởng chính của phương pháp này là chọn một coreset sao cho các gradient do coreset tạo ra khớp với các gradient do tập dữ liệu huấn luyện gốc tạo ra càng chặt chẽ càng tốt.

- GMC định nghĩa sai số khớp gradient dựa trên kỳ vọng của bình phương chuẩn của hiệu giữa gradient của hàm mất mát trên coreset và trên toàn bộ tập dữ liệu.
- Phương pháp này nhúng các hàm gradient vào một không gian hữu hạn chiều, sau đó giải một bài toán tối ưu hóa bậc hai có ràng buộc về số lượng. Mặc dù bài toán này là NP-khó, nó có thể được giải xấp xỉ bằng các phương pháp tham lam.
- Một phát hiện quan trọng là việc chọn phân phối $p(\theta)$ (trên các tham số mô hình) là phân phối khởi tạo của mô hình đã đủ thông tin để trích xuất các coreset tốt. Điều này cho phép chọn coreset trước khi huấn luyện.
- Trong bối cảnh học liên tục (continual learning), GMC có thể được sử dụng để duy trì một bộ nhớ đệm (rehearsal memory) bằng cách chọn một coreset khớp với gradient tổng hợp của tất cả dữ liệu đã thấy.

theta

2.3 FastCore

Đây là một framework chọn coreset hiệu quả hơn dựa trên việc xấp xỉ gradient thông qua phân cụm dữ liệu.

- FastCore chia toàn bộ tập huấn luyện thành nhiều cụm, mỗi cụm chứa các mục có khoảng cách đặc trưng tương tự nhau.
- Framework này chứng minh rằng gradient đầy đủ có thể được giới hạn dựa trên khoảng cách đặc trưng tối đa giữa mỗi mục và mỗi cụm, cho phép chọn coreset hiệu quả hơn bằng cách lặp qua các cụm này.
- FastCore sử dụng kỹ thuật lượng tử hóa tích (product quantization) để ước tính hiệu quả khoảng cách đặc trưng tối đa.
- Thuật toán FastCore sử dụng một framework 3 vòng lặp để chọn coreset, nhưng hiệu quả hơn các phương pháp gradient approximation trước đó vì nó chỉ cần lặp qua các cụm thay vì toàn bộ các mục dữ liệu trong một số bước.

- Đối với mạng nơ-ron sâu, FastCore lặp lại việc huấn luyện một vài epochs trên toàn bộ dữ liệu để tính toán gradient của lớp cuối cùng, sau đó sử dụng các gradient này như các vector đặc trưng để chọn coreset bằng thuật toán tương tự cho bài toán lồi.

2.4 AUTOMATA's Gradient-based Subset Selection (GSS)

Trong framework AUTOMATA, một chiến lược chọn tập hợp con dựa trên gradient (GSS) được sử dụng cho việc điều chỉnh siêu tham số hiệu quả về mặt tính toán. GSS được so sánh với các phương pháp chọn tập hợp con khác như Random và CRAIG.

2.5 RETRIEVE

- Framework này thực hiện việc chọn coreset và học mô hình phân loại trên coreset đã chọn một cách đồng thời.
- RETRIEVE huấn luyện mô hình trên coreset đã chọn trong một số epochs nhất định, và sau đó chọn một coreset mới dựa trên gradient mất mát (loss gradients). Quá trình này lặp lại cho đến khi mô hình hội tụ.

2.6 Sequential Coreset

Phương pháp này dựa trên tính chất "cục bộ" của các thuật toán gradient descent. Nó xây dựng một chuỗi các coreset "cục bộ" khi quỹ đạo của giả thuyết (hypothesis) tiến triển trong không gian tham số.

- Coreset cục bộ được xây dựng cho một vùng lân cận của điểm hiện tại trong không gian tham số, và nó đảm bảo rằng hàm mục tiêu (objective function) được bảo toàn xấp xỉ trong vùng đó.
- Phương pháp này cũng bảo toàn xấp xỉ gradient trong vùng cục bộ. Các phương pháp này cho thấy gradient là một tín hiệu quan trọng để xác định các điểm dữ liệu nào là quan trọng nhất để giữ lại trong coreset, nhằm mục đích xấp xỉ chính xác gradient của toàn bộ tập dữ liệu và do đó, đạt được hiệu suất học tập tương đương hoặc gần tương đương với chi phí tính toán thấp hơn.

3 Phương pháp gradient tăng tốc Nesterov (Nesterov's Accelerated Gradient - NAG)

Thông tin về gradient mà NAG sử dụng lại là một yếu tố then chốt trong nhiều kỹ thuật lựa chọn coreset hiệu quả. Các phương pháp này sử dụng gradient (hoặc sự xấp xỉ của chúng) để đánh giá mức độ quan trọng của các điểm dữ liệu và xây dựng một tập hợp con đại diện. Trong các quy trình huấn luyện lặp đi lặp lại hoặc các framework sequential coreset, NAG có thể được sử dụng như thuật toán tối ưu hóa trên coreset đã được chọn bằng các tiêu chí dựa trên gradient.

Phương pháp gradient tăng tốc Nesterov (Nesterov's Accelerated Gradient - NAG) là một thuật toán tối ưu hóa nhằm tăng tốc độ hội tụ của phương pháp gradient descent truyền thống. Ý tưởng chính của NAG là thêm một thành phần "động lượng" vào bước cập nhật tham số, giúp thuật toán "nhìn về phía trước" và giảm thiểu hiện tượng dao động, từ đó hội tụ nhanh hơn.

Công thức cập nhật của phương pháp NAG (một dạng phổ biến): Bắt đầu với tham số ban đầu x_0 và $y_0 = x_0$. Với mỗi bước lặp $k \geq 1$, cập nhật theo các quy tắc sau:

1. Bước "nhìn trước" (look-ahead step): Tính một điểm trung gian y_{k-1} bằng cách kết hợp tham số hiện tại x_{k-1} và sự thay đổi từ bước trước $(x_{k-1} - x_{k-2})$ với một hệ số động lượng (Lưu ý: Hệ số động lượng $\frac{k-1}{k+2}$ có thể thay đổi tùy thuộc vào biến thể của NAG).

2. Bước cập nhật gradient: Tính gradient của hàm mục tiêu f tại điểm nhìn trước y_{k-1}, và cập nhật tham số x_k theo hướng ngược lại của gradient này với một kích thước bước s, trong đó s là kích thước bước (learning rate) và $\nabla f(y_{k-1})$ là gradient của hàm f tại y_{k-1}.

Vai trò của thành phần động lượng:

- Thành phần $\frac{k-1}{k+2}(x_{k-1} - x_{k-2})$ được gọi là thành phần động lượng. Nó cho phép bước cập nhật hiện tại mang theo một phần của hướng di chuyển ở bước trước.
- Điều này giúp thuật toán vượt qua các vùng có gradient nhỏ nhanh hơn.
- Động lượng cũng giúp giảm thiểu sự dao động của thuật toán khi tiến gần đến điểm tối ưu.

Tốc độ hội tụ:

- Đối với các hàm lồi với gradient Lipschitz liên tục (với hằng số L), NAG đạt được tốc độ hội tụ $\leq O(\frac{|x_0-x^*|^2}{sk^2})$*, trong đó f^* là giá trị tối ưu của hàm f.
- Tốc độ hội tụ này là tối ưu cho các phương pháp bậc nhất chỉ sử dụng thông tin về gradient.
- So với gradient descent thông thường, chỉ đạt được tốc độ hội tụ $O(1/k)$, NAG hội tụ nhanh hơn đáng kể.

4 Bài toán xác định coreset cho hồi quy tuyến tính

Bài toán hồi quy tuyến tính: Tìm một vector hệ số $\beta \in \mathbb{R}^d$ sao cho hàm dự đoán tuyến tính $f_\beta(x) = x^\top \beta$ xấp xỉ tốt nhất nhãn $y \in \mathbb{R}$ tương ứng với dữ liệu đầu vào $x \in \mathbb{R}^d$.

Input: Ma trận đặc trưng $X \in \mathbb{R}^{n\times d}$ (mỗi dòng là một điểm dữ liệu) và vector nhãn $y \in \mathbb{R}^n$.

Bài toán tối ưu: $\min_{\beta\in\mathbb{R}^d} \mathcal{L}(\beta) = \|X\beta - y\|^2$.
Nếu $X^\top X$ khả nghịch, nghiệm tối ưu là:

$$\beta^* = (X^\top X)^{-1} X^\top y.$$

Bài toán xác định coreset cho hồi quy tuyến tính

Mục tiêu của bài toán: Cho một tập dữ liệu $X \in \mathbb{R}^{n\times d}$ và $y \in \mathbb{R}^n$, ta muốn chọn ra một tập con (coreset) đủ nhỏ sao cho khi huấn luyện mô hình hồi quy tuyến tính trên coreset, nghiệm thu được vẫn gần với nghiệm của toàn bộ dữ liệu β^* hoặc tạo ra mô hình có sai số nhỏ trên toàn tập.

Cách biểu diễn coreset: Thay vì chọn tập con trực tiếp, ta gán trọng số $w_i \in [0, 1]$ cho từng điểm, tạo vector $w \in [0, 1]^n$. Từ đó xây dựng:

- Ma trận trọng số $W = \text{diag}(w_1, w_2, \ldots, w_n)$
- Nghiệm hồi quy tương ứng

$$\beta(w) = (X^\top W X)^{-1} X^\top W y$$

Hàm mất mát theo trọng số w: Ta muốn định nghĩa loss $\mathcal{L}(w)$ sao cho nhỏ nhất khi w chọn đúng coreset.

- **Cách 1**: Sai số dự đoán toàn tập

$$\mathcal{L}(w) = \|X\beta(w) - y\|^2$$

  Ý nghĩa: dùng nghiệm $\beta(w)$ từ coreset để dự đoán toàn tập, nếu loss này nhỏ thì coreset tốt.

- Cách 2: Sai lệch nghiệm so với toàn tập (nếu biết β^*)

$$\mathcal{L}_{\text{diff}}(w) = \|\beta(w) - \beta^*\|^2 \quad \text{với } \beta^* = (X^\top X)^{-1} X^\top y$$

  Ý nghĩa: đo trực tiếp sai khác giữa nghiệm coreset và nghiệm toàn tập. Dùng trong nghiên cứu lý thuyết, nhưng không dùng trong thực tế (vì không có β^*).

Bài toán tối ưu hàm mất mát

$$\min_{w \in [0,1]^n} \mathcal{L}(w) = \|X\beta(w) - y\|^2 \quad \text{subject to: } \sum w_i \leq k$$

$*$ **Nhận xét:**

- Hàm $\mathcal{L}(w)$ không lồi, không giả lồi vì có cấu trúc phi tuyến, chứa nghịch đảo ma trận, nhưng vẫn khả vi, vẫn tính được gradient nên dùng được GD/NAG, và có thể trong thực nghiệm vẫn có thể hội tụ tốt.
- "Nới lỏng" $\mathcal{L}(w)$ mà không thay đổi bài toán gốc quá nhiều sử dụng *hàm phạt* (regularization) và *tái tham số hoá* (reparameterizationr).

Reparameterization

Đây là kỹ thuật thay đổi biến tối ưu từ w sang một biến khác z, sao cho dễ tối ưu hơn mà vẫn bảo đảm ràng buộc (ví dụ: $w \in [0, 1]$) và Gradient tốt hơn, mượt hơn. Cụ thể, với $w \in [0, 1]^n$, ta đặt:

$$w_i = \sigma(z_i) = \frac{1}{1 + e^{-z_i}} \quad \text{(sigmoid)}$$

Bây giờ tối ưu trên $z \in \mathbb{R}^n$, không cần ràng buộc nữa.

Regularization

Đây là kỹ thuật thêm một điều kiện phụ hoặc ưu tiên vào hàm mất mát để làm mượt hàm, tránh overfitting hoặc phân bố xấu và hướng thuật toán về vùng dễ hội tụ hơn. Biểu diễn tổng quát:

$$\mathcal{L}_{\text{reg}}(w) = \mathcal{L}(w) + \lambda \cdot \Omega(w)$$

Với $\Omega(w)$ là hàm phạt (ví dụ: norm, entropy,...), và $\lambda > 0$ điều chỉnh mức phạt.

- Entropy regularization (ưu tiên phân bố mượt, tránh sparsity gắt):

$$\Omega(z) = -\sum_{i=1}^{n} w_i \log(w_i + \varepsilon) = -\sum \sigma(z_i) \log(\sigma(z_i) + \varepsilon)$$

- L_2 regularization (tránh w_i quá lớn): $\Omega(z) = \|w\|^2 = \sum \sigma(z_i)^2$

5 Bài toán xác định coreset cho hồi quy logistic

Bài toán hồi quy logistic: Tìm một vector hệ số $\beta \in \mathbb{R}^d$ sao cho mô hình logistic

$$f_\beta(x) = \frac{1}{1 + e^{-x^\top \beta}}$$

gần đúng xác suất $y \in \{0, 1\}$ tương ứng với điểm dữ liệu $x \in \mathbb{R}^d$.

Input: Ma trận dữ liệu $X \in \mathbb{R}^{n \times d}$ (mỗi dòng là một điểm dữ liệu) và vector nhãn $y \in \{0, 1\}^n$.

Hàm mất mát (negative log-likelihood):

$$\mathcal{L}(\beta) = -\sum_{i=1}^{n} [y_i \log f_\beta(x_i) + (1 - y_i) \log(1 - f_\beta(x_i))]$$

Bài toán xác định coreset cho hồi quy logistic

Mục tiêu của bài toán: Cho tập dữ liệu (X, y), ta muốn chọn ra một tập con (coreset) đủ nhỏ sao cho khi huấn luyện mô hình logistic trên coreset, nghiệm thu được vẫn gần với nghiệm trên toàn bộ dữ liệu β^*, hoặc tạo ra mô hình có sai số nhỏ trên toàn tập.

Cách biểu diễn coreset: Thay vì chọn tập con trực tiếp, ta gán trọng số $w_i \in [0,1]$ cho từng điểm, tạo vector $w \in [0,1]^n$. Từ đó xây dựng:

- Hàm mất mát theo trọng số:

$$\mathcal{L}(w,\beta) = -\sum_{i=1}^{n} w_i \left[y_i \log f_\beta(x_i) + (1-y_i)\log(1-f_\beta(x_i))\right]$$

- Nghiệm logistic tương ứng:

$$\beta(w) = \arg\min_{\beta} \mathcal{L}(w,\beta)$$

Bài toán tối ưu xác định coreset:

$$\min_{w\in[0,1]^n} \mathcal{L}(w,\beta(w)) \quad \text{subject to } \sum w_i \le k$$

Ý nghĩa: Trọng số w xác định coreset sao cho nghiệm tối ưu từ coreset tạo ra mô hình có sai số dự đoán nhỏ nhất trên toàn tập. k là tham số để giới hạn số phần tử của coreset, có thể chọn $k = \alpha \cdot n$ (với $\alpha \in (0,1)$ là tỉ lệ số phần tử còn lại mong muốn).

*** Nhận xét:**

- Hàm $\mathcal{L}(w,\beta(w))$ không lồi, vì chứa tối ưu lồng nhau và $f_\beta(x)$ phi tuyến.
- Tuy nhiên, vẫn có thể áp dụng các kỹ thuật như:
  - **Reparameterization:** Thay đổi biến $w_i = \sigma(z_i) = \frac{1}{1+e^{-z_i}}$ để loại bỏ ràng buộc $w \in [0,1]^n$, đưa về không gian $\mathbb{R}^n$ trơn hơn.
  - **Regularization:** Thêm hàm phạt vào hàm mất mát để làm mượt hàm, ví dụ:
    - Entropy regularization: $\Omega(z) = -\sum_{i=1}^{n} \sigma(z_i)\log(\sigma(z_i)+\varepsilon)$
    - L2 regularization: $\Omega(z) = \|\sigma(z)\|^2 = \sum_i \sigma(z_i)^2$

6 Bài toán xác định coreset cho mô hình SVM

Bài toán phân loại SVM

Cho tập dữ liệu $(x_i, y_i) \in \mathbb{R}^d \times \{-1,+1\}$ với $i = 1,\ldots,n$, SVM tìm vector trọng số $\theta \in \mathbb{R}^d$ và bias $b \in \mathbb{R}$ sao cho:

$$f(x) = \text{sign}(\theta^\top x + b)$$

Bài toán tối ưu tương ứng là:

$$\mathcal{L}(\theta, b) = \sum_{i=1}^{n} \max\left(0, 1 - y_i(\theta^\top x_i + b)\right) + \lambda\|\theta\|^2$$

Với $\lambda > 0$ là hệ số điều chỉnh mức phạt (regularization).

Bài toán xác định coreset cho SVM

Mục tiêu: Chọn vector trọng số $w \in [0,1]^n$ sao cho khi huấn luyện mô hình SVM trên tập trọng số w, nghiệm thu được vẫn gần với nghiệm huấn luyện trên toàn bộ dữ liệu.

- Ma trận trọng số: $W = \text{diag}(w_1, \ldots, w_n)$
- Hàm mất mát có trọng số:

$$\mathcal{L}(w; \theta, b) = \sum_{i=1}^{n} w_i \cdot \max(0, 1 - y_i(\theta^\top x_i + b)) + \lambda\|\theta\|^2$$

Bài toán tối ưu chọn coreset:

$$\min_{w \in [0,1]^n} \min_{\theta, b} \mathcal{L}(w; \theta, b) \quad \text{subject to} \quad \sum_{i=1}^{n} w_i \leq k$$

Ghi chú: Đây là bài toán tối ưu hai tầng (bilevel optimization), trong đó w điều chỉnh chọn coreset, còn (θ, b) là tham số mô hình.

Regularization

- **L2 regularization:** Tránh θ quá lớn:

$$\mathcal{L}_{\text{reg}} = \mathcal{L}(w; \theta, b) + \lambda\|\theta\|^2$$

- **Entropy regularization:** Tránh phân phối w quá sparse:

$$\Omega(w) = -\sum_{i=1}^{n} w_i \log(w_i + \varepsilon)$$

Reparameterization

Để tối ưu gradient tốt hơn, ta đặt $w_i = \sigma(z_i) = \dfrac{1}{1 + e^{-z_i}}$, khi đó:

$$w = \sigma(z), \quad z \in \mathbb{R}^n$$