

Vũ Hữu Tiệp

Convex Optimization và Support Vector Machines

Theo blog: <http://machinelearningcoban.com>

Last update:

August 20, 2017

Contents

0 Lời nói đầu	1
----------------------------	----------

Part I Convex Optimization - Tối Ưu Lồi

1 Convex sets và convex functions	4
2 Convex Optimization Problems	24
3 Duality	45

Part II Support Vector Machines

4 Support Vector Machine	60
5 Soft Margin Support Vector Machines	73
6 Kernel Support Vector Machine	92
7 Multi-class Support Vector Machine	103

Part III Phụ lục

51 Ôn tập Đại số tuyến tính	122
---------------------------------------	-----

Index	133
-----------------	-----

0

Lời nói đầu

Chào các bạn,

Tài liệu này là một phần trong [Dự án viết ebook Machine Learning cơ bản](#) của tôi đang được thực hiện, dự kiến sẽ được hoàn thành cuối năm 2017. Khi chuẩn bị tài liệu này, blog đã có 32 bài viết và nhiều ghi chú ngắn về Machine Learning/AI và Optimization. Hướng tiếp cận của tôi là giới thiệu mỗi thuật toán Machine Learning thông qua việc xây dựng một hàm số đặc biệt được gọi là *hàm mất mát*, hoặc *hàm mục tiêu* và các phương pháp tối ưu hàm mục tiêu. Để có thể hiểu sâu về các thuật toán Machine Learning, tôi luôn cho rằng hiểu rõ cách xây dựng hàm mất mát và cách tối ưu các hàm đó đóng một vai trò quan trọng. Vì vậy, tôi cũng dành thời gian cho một số bài viết liên quan đến Tối Ưu.

Tài liệu này hiện gồm ba Phần trong cuốn ebook: Tối Ưu Lồi, Support Vector Machines, và Ôn tập Đại Số Tuyến Tính. Trong đó, hai phần đầu tiên nằm ở phần cuối cùng của cuốn ebook, là phần có nhiều kiến thức toán nhất. Phần thứ ba ở Phụ Lục được thêm vào để bổ sung những kiến thức toán quan trọng.

Trong lĩnh vực Tối Ưu, Tối Ưu Lồi đóng vai trò quan trọng hơn cả vì những tính chất quan trọng của nó. Tôi chọn ba bài viết về Tối Ưu Lồi để chuẩn bị cho tài liệu này vì tôi biết rằng nhiều bạn đọc trong blog muốn đọc những phần có nhiều toán trên giấy hơn là đọc trên máy. Việc in trực tiếp từ màn hình website ra không sự tốt vì dù sao việc chuyển đổi cũng là tự động. Không chỉ trong Machine Learning, các lĩnh vực khoa học kỹ thuật và cả tài chính kinh tế cũng rất cần tới tối ưu. Tôi hy vọng rằng tài liệu này sẽ giúp ích cho nhiều người Việt đang học tập và nghiên cứu ở trong và ngoài nước.

Support Vector Machine là một trong những thuật toán đẹp nhất của Machine Learning. Bài toán tối ưu của nó được chứng minh là một bài toán lồi, và nghiệm tìm được là duy nhất. Các biến thể khác của Machine Learning cũng được đề cập.

Ngôn ngữ trong tài liệu này gần giống với ngôn ngữ trong blog và có liên quan chặt chẽ tới các bài viết khác mà tôi có dẫn links. Tuy nhiên, độc giả chưa đọc blog cũng có thể hiểu được vì đây là kiến thức tổng quan về Tối Ưu Lồi.

Tài liệu này được tổng hợp trong hai ngày, nội dung gần như tương tự như trên blog. Tuy nhiên, vì việc chuyển từ ngôn ngữ trên web sang LaTeX khá phức tạp nên chắc chắn tôi không tránh khỏi sai sót. Nếu thấy phần nào cần phải sửa lại, bạn hãy cho tôi biết qua địa chỉ email vuhuutiep@gmail.com. Tôi sẽ trả lời và chỉnh sửa ngay khi có thể.

Vấn đề bản quyền:

Toàn bộ nội dung trong bài, source code, và hình ảnh minh họa (trừ nội dung có trích dẫn) đều thuộc bản quyền của tôi, Vũ Hữu Tiệp.

Tôi rất mong muốn kiến thức tôi viết trong blog này đến được với nhiều người. Tuy nhiên, tôi không ủng hộ bất kỳ một hình thức sao chép không trích nguồn nào. Mọi nguồn tin trích đăng bài viết cần nêu rõ tên blog (Machine Learning cơ bản), tên tác giả (Vũ Hữu Tiệp), và kèm link gốc của bài viết. Các bài viết trích dẫn quá 25% toàn văn bất kỳ một post nào trong blog này là không được phép, trừ trường hợp có sự đồng ý của tác giả.

Mọi vấn đề liên quan đến việc sao chép, đăng tải, sử dụng bài viết, cũng như trao đổi, cộng tác, xin vui lòng liên hệ với tôi tại địa chỉ email: vuhuutiep@gmail.com.

Tôi xin chân thành cảm ơn!

Trân trọng,

Vũ Hữu Tiệp

www.machinelearningcoban.com

Hoa Kỳ, ngày 20 tháng 8 năm 2017.

Part I

Convex Optimization - Tối Ưu Lồi

Convex sets và convex functions

1.1 Giới thiệu

Nếu bạn đã đọc các bài trước trong [Blog Machine Learning cơ bản](#), chúng ta đã làm quen với rất nhiều bài toán tối ưu. Học Machine Learning là phải học Toán Tối Ưu, và để hiểu hơn về Toán Tối Ưu, với tôi cách tốt nhất là tìm hiểu các thuật toán Machine Learning. Cho tới lúc này, những bài toán tối ưu các bạn đã nhìn thấy trong blog đều là các bài toán tối ưu không ràng buộc (unconstrained optimization problems), tức tối ưu hàm măt măt mà không có điều kiện ràng buộc (constraints) nào về nghiệm cả.

Không chỉ trong Machine Learning, trên thực tế các bài toán tối ưu thường có rất nhiều ràng buộc khác nhau. Ví dụ:

- Tôi muốn thuê một ngôi nhà cách trung tâm Hà Nội không quá 5km với giá càng thấp càng tốt. Trong bài toán này, giá thuê nhà chính là hàm măt măt (*loss function*, đôi khi người ta cũng dùng *cost function* để chỉ hàm số cần tối ưu), điều kiện khoảng cách không quá 5km chính là ràng buộc (constraint).
- Quay lại [bài toán dự đoán giá nhà theo Linear Regression](#), giá nhà là một hàm tuyến tính của diện tích, số phòng ngủ và khoảng cách tới trung tâm. Rõ ràng, khi làm bài toán này, ta dự đoán rằng giá nhà tăng theo diện tích và số phòng ngủ, giảm theo khoảng cách. Vậy nên một nghiệm được gọi là *có lý một chút* nếu hệ số tương ứng với diện tích và số phòng ngủ là các số dương, hệ số tương ứng với khoảng cách là một số âm. Để tránh các nghiệm ngoại lai không mong muốn, khi giải bài toán tối ưu, ta nên cho thêm các điều kiện ràng buộc này.

Trong Tối Ưu, một bài toán có ràng buộc thường được viết dưới dạng:

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ &h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned}$$

Trong đó, vector $\mathbf{x} = [x_1, x_2, \dots, x_n]^T$ được gọi là *biến tối ưu* (*optimization variable*). Hàm số $f_0 : \mathbb{R}^n \rightarrow \mathbb{R}$ được gọi là *hàm mục tiêu* (*objective function*, các hàm mục tiêu trong Machine Learning thường được gọi là *hàm mất mát*). Các hàm số $f_i, h_j : \mathbb{R}^n \rightarrow \mathbb{R}, i = 1, 2, \dots, m; j = 1, 2, \dots, p$ được gọi là các *hàm ràng buộc* (hoặc đơn giản là *ràng buộc* - constraints). Tập hợp các điểm \mathbf{x} thỏa mãn các *ràng buộc* được gọi là *feasible set*. Mỗi điểm trong *feasible set* được gọi là *feasible point*, các điểm không trong *feasible set* được gọi là *infeasible points*.

Chú ý:

- Nếu bài toán là tìm giá trị lớn nhất thay vì nhỏ nhất, ta chỉ cần đổi dấu của $f_0(\mathbf{x})$.
- Nếu ràng buộc là *lớn hơn hoặc bằng*, tức $f_i(\mathbf{x}) \geq b_i$, ta chỉ cần đổi dấu của ràng buộc là sẽ có điều kiện *nhỏ hơn hoặc bằng* $-f_i(\mathbf{x}) \leq -b_i$.
- Các ràng buộc cũng có thể là *lớn hơn hoặc nhỏ hơn*.
- Nếu ràng buộc là *bằng nhau*, tức $h_j(\mathbf{x}) = 0$, ta có thể viết nó dưới dạng hai bất đẳng thức $h_j(\mathbf{x}) \leq 0$ và $-h_j(\mathbf{x}) \leq 0$. Trong một vài tài liệu, người ta bỏ các phương trình ràng buộc $h_j(\mathbf{x}) = 0$ đi.
- Trong bài viết này, \mathbf{x}, \mathbf{y} được dùng chủ yếu để ký hiệu các biến số, không phải là dữ liệu như trong các bài trước. Biến tối ưu chính là biến được ghi dưới dấu $\arg \min$. Khi viết một bài toán Tối Ưu, ta cần chỉ rõ biến nào cần được tối ưu, biến nào là cố định.

Các bài toán tối ưu, nhìn chung không có cách giải tổng quát, thậm chí có những bài chưa có lời giải. Hầu hết các phương pháp tìm nghiệm không chứng minh được nghiệm tìm được có phải là *global optimal* hay không, tức đúng là điểm làm cho hàm số đạt giá trị nhỏ nhất hay lớn nhất hay không. Thay vào đó, nghiệm thường là các *local optimal*, tức các *điểm cực trị*. Trong nhiều trường hợp, các nghiệm *local optimal* cũng mang lại những kết quả tốt.

Để bắt đầu học Tối Ưu, chúng ta cần học một mảng rất quan trọng trong đó, có tên là *Tối Ưu Lồi* (convex optimization), trong đó *hàm mục tiêu* là một *hàm lồi* (convex function), *feasible set* là một *tập lồi* (convex set). Những tính chất đặc biệt về *local optimal* và *global optimal* của một *hàm lồi* khiến Tối Ưu Lồi trở nên cực kỳ quan trọng. Trong bài viết này, tôi sẽ giới thiệu tới các bạn các định nghĩa và tính chất cơ bản của *tập lồi* và *hàm lồi*. *Bài toán tối ưu lồi* (convex optimization problems) sẽ được đề cập trong bài tiếp theo.



Example of convex sets

Hình 1.1: Các ví dụ về convex sets.

1.2 Convex sets

1.2.1 Định nghĩa

Khái niệm về *convex sets* có lẽ không xa lạ với các bạn học sinh Việt Nam khi chúng ta đã nghe về *đa giác lồi*. *Lồi*, hiểu đơn giản là *phình ra ngoài*, hoặc *nhô ra ngoài*. Trong toán học, *bằng phẳng* cũng được coi là *lồi*.

Định nghĩa 1: Một tập hợp được gọi là *tập lồi* (convex set) nếu đoạn thẳng nối hai điểm bất kỳ trong tập hợp hợp đó nằm trọn vẹn trong tập hợp đó.

Một vài ví dụ về convex sets được cho trong Hình 1.1.

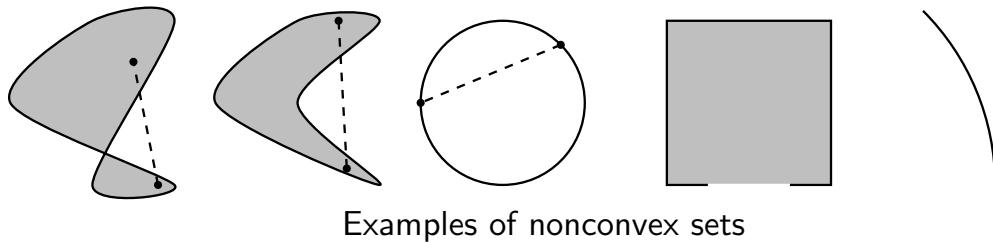
Các hình với đường biên màu đen thể hiện việc bao gồm cả biên, biên màu trắng thể hiện việc biên đó không nằm trong tập hợp đang xét. Đường hoặc đoạn thẳng cũng là một tập lồi theo định nghĩa phía trên.

Một vài ví dụ thực tế:

- Giả sử có một căn phòng có dạng hình *lồi*, nếu ta đặt một bóng đèn đủ sáng ở bất kỳ vị trí nào trong phòng, mọi điểm trong căn phòng đều được chiếu sáng.
- Nếu một đất nước có bản đồ dạng một hình *lồi* thì đường bay nối giữa hai thành phố bất kỳ trong đất nước đó đều nằm trọn vẹn trong không phận của nước đó. (Không như Việt Nam, muốn bay thẳng Hà Nội - Hồ Chí Minh phải bay qua không phận Campuchia).

Hình 1.2 minh họa một vài ví dụ về *nonconvex sets*, tức tập hợp mà không phải là lồi:

Ba hình đầu tiên không phải là lồi vì các đường nét đứt chứa nhiều điểm không nằm trong các tập đó. Hình thứ tư, hình vuông không có biên ở đáy, không phải là *tập lồi* vì đoạn



Hình 1.2: Các ví dụ về nonconvex sets.

thẳng nối hai điểm ở đây có thể chứa phần ở giữa không thuộc tập đang xét (Nếu không có biên thì hình vuông vẫn là một *tập lồi*, nhưng biên nửa vời như ví dụ này thì hãy chú ý). Một đường cong bất kỳ cũng không phải là *tập lồi* vì dễ thấy đường thẳng nối hai điểm bất kỳ không thuộc đường cong đó.

Để mô tả một *tập lồi* dưới dạng toán học, ta sử dụng:

Định nghĩa 2: Một tập hợp \mathcal{C} được gọi là *convex* nếu với hai điểm bất kỳ $\mathbf{x}_1, \mathbf{x}_2 \in \mathcal{C}$, điểm $\mathbf{x}_\theta = \theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2$ cũng nằm trong \mathcal{C} với bất kỳ $0 \leq \theta \leq 1$.

Có thể thấy rằng, tập hợp các điểm có dạng $(\theta\mathbf{x}_1 + (1 - \theta)\mathbf{x}_2)$ chính là *đoạn thẳng* nối hai điểm \mathbf{x}_1 và \mathbf{x}_2 .

Với các định nghĩa này thì *toàn bộ không gian* là một *tập lồi* vì đoạn nào cũng nằm trong không gian đó. Tập rỗng cũng có thể coi là một trường hợp đặc biệt của *tập lồi*.

Dưới đây là một vài ví dụ hay gấp về *tập lồi*.

1.2.2 Ví dụ

Hyperplanes và halfspaces

Một **hyperplane** (siêu mặt phẳng) trong không gian n chiều là tập hợp các điểm thỏa mãn phương trình:

$$a_1x_1 + a_2x_2 + \cdots + a_nx_n = \mathbf{a}^T \mathbf{x} = b \quad (1.1)$$

với $b, a_i, i = 1, 2, \dots, n$ là các số thực.

Hyperplanes là các *tập lồi*. Điều này có thể dễ dàng suy ra từ Định nghĩa 1. Với Định nghĩa 2, chúng ta cũng dễ dàng nhận thấy. Nếu:

$$\mathbf{a}^T \mathbf{x}_1 = \mathbf{a}^T \mathbf{x}_2 = b$$

thì với $0 \leq \theta \leq 1$ bất kỳ:

$$\mathbf{a}^T \mathbf{x}_\theta = \mathbf{a}^T (\theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2) = \theta b + (1 - \theta)b = b$$

Một **halfspace** (nửa không gian) trong không gian n chiều là tập hợp các điểm thỏa mãn phương trình:

$$a_1 x_1 + a_2 x_2 + \cdots + a_n x_n = \mathbf{a}^T \mathbf{x} \leq b$$

với $b, a_i, i = 1, 2, \dots, n$ là các số thực.

Các halfspace cũng là các tập lồi, bạn đọc có thể dễ dàng nhận thấy theo Định nghĩa 1 hoặc chứng minh theo Định nghĩa 2.

Norm balls

Euclidean balls (hình tròn trong mặt phẳng, hình cầu trong không gian ba chiều) là tập hợp các điểm có dạng:

$$B(\mathbf{x}_c, r) = \{\mathbf{x} \mid \|\mathbf{x} - \mathbf{x}_c\|_2 \leq r\} = \{\mathbf{x}_c + r\mathbf{u} \mid \|\mathbf{u}\|_2 \leq 1\}$$

Theo Định nghĩa 1, chúng ta có thể thấy Euclidean balls là các tập lồi, nếu phải chứng minh, ta dùng Định nghĩa 2 và [các tính chất của norms](#). Với $\mathbf{x}_1, \mathbf{x}_2$ bất kỳ thuộc $B(\mathbf{x}_c, r)$ và $0 \leq \theta \leq 1$ bất kỳ:

$$\begin{aligned} \|\mathbf{x}_\theta - \mathbf{x}_c\|_2 &= \|\theta(\mathbf{x}_1 - \mathbf{x}_c) + (1 - \theta)(\mathbf{x}_2 - \mathbf{x}_c)\|_2 \\ &\leq \theta \|\mathbf{x}_1 - \mathbf{x}_c\|_2 + (1 - \theta) \|\mathbf{x}_2 - \mathbf{x}_c\|_2 \\ &\leq \theta r + (1 - \theta)r = r \end{aligned}$$

Vậy nên $\mathbf{x}_\theta \in B(\mathbf{x}_c, r)$.

Euclidean ball sử dụng norm 2 làm khoảng cách. Nếu sử dụng norm bất kỳ là khoảng cách, ta vẫn được một *tập lồi*.

Khi sử dụng norm p :

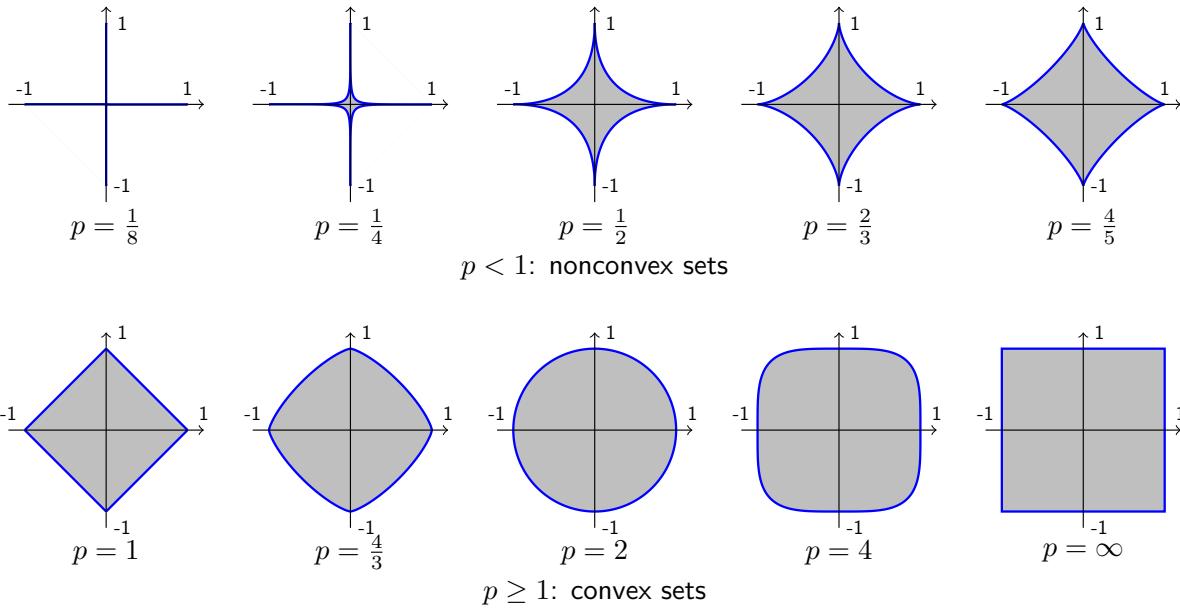
$$\|\mathbf{x}\|_p = (\|x_1\|^p + \|x_2\|^p + \dots + \|x_n\|^p)^{\frac{1}{p}}$$

với p là một số thực bất kỳ không nhỏ hơn 1 ta cũng thu được các *tập lồi*.

Hình 1.13 minh họa tập hợp các điểm có tọa độ (x, y) trong không gian hai chiều thỏa mãn:

$$(|x|^p + |y|^p)^{1/p} \leq 1 \quad (1.2)$$

$$\mathcal{C}_p = \{(x, y) \mid (|x|^p + |y|^p)^{1/p} \leq 1\}$$



Hình 1.3: Hình dạng của các tập hợp bị chặn bởi pseudo-norms (hàng trên) và norm (hàng dưới).

với hàng trên là các tập với $0 < p < 1$ (không phải norm) và hàng dưới tương ứng với $p \geq 1$.

Chúng ta có thể thấy rằng khi p nhỏ gần bằng 0, tập hợp các điểm thỏa mãn bất đẳng thức (1.2) gần như nằm trên các trục tọa độ và bị chặn trong đoạn $[0, 1]$. Quan sát này sẽ giúp ích cho các bạn khi làm việc với (giả) norm 0 sau này. Khi $p \rightarrow \infty$, các tập hợp hội tụ về hình vuông.

Đây cũng là một trong các lý do vì sao cần có điều kiện $p \geq 1$ khi định nghĩa norm.

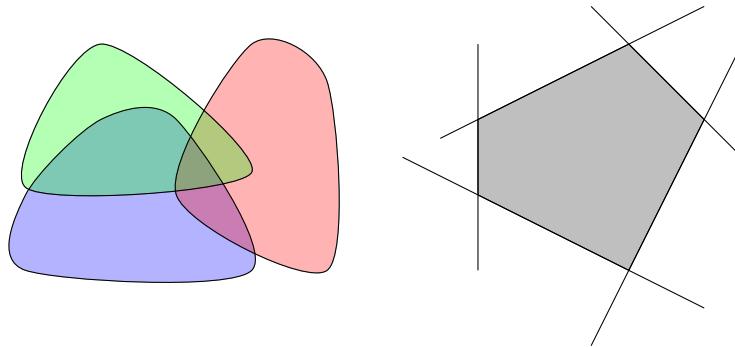
Ellipsoids Các ellipsoids (ellipse trong không gian nhiều chiều) cũng là các *tập lồi*. Thực chất, ellipsoïdes có mối quan hệ mật thiết tới [Khoảng cách Mahalanobis](#). Khoảng cách này vốn dĩ là một norm nên ta có thể chứng minh theo Định nghĩa 2 được tính chất lồi của các ellipsoids.

Mahalanobis norm của một vector $\mathbf{x} \in \mathbb{R}^n$ được định nghĩa là:

$$\|\mathbf{x}\|_{\mathbf{A}} = \sqrt{\mathbf{x}^T \mathbf{A}^{-1} \mathbf{x}}$$

Với \mathbf{A}^{-1} là một ma trận thỏa mãn:

$$\mathbf{x}^T \mathbf{A}^{-1} \mathbf{x} \geq 0, \quad \forall \mathbf{x} \in \mathbb{R}^n \tag{1.3}$$



Hình 1.4: Trái: Giao của các tập lồi là một tập lồi. Phải: giao của các hyperplanes và halfspace là một tập lồi và được gọi là polyhedron (số nhiều là polyhedra).

Khi một ma trận \mathbf{A}^{-1} thỏa mãn điều kiện (1.3), ta nói ma trận đó *xác định dương* (*positive definite*). Một ma trận là *xác định dương* nếu các trị riêng (eigenvalues) của nó là dương.

Nhân tiện, một ma trận \mathbf{B} được gọi là *nửa xác định dương* (*positive semidefinite*) nếu các *trị riêng* của nó là không âm. Khi đó $\mathbf{x}^T \mathbf{B} \mathbf{x} \geq 0, \forall \mathbf{x}$. Nếu dấu bằng xảy ra khi và chỉ khi $\mathbf{x} = 0$ thì ta nói ma trận đó *xác định dương*. Trong biểu thức (1.3), vì ma trận \mathbf{A} có nghịch đảo nên mọi *trị riêng* của nó phải khác không. Vì vậy, \mathbf{A} là một ma trận *xác định dương*.

Một ma trận \mathbf{A} là *xác định dương* hoặc *nửa xác định dương* sẽ được ký hiệu lần lượt như sau:

$$\mathbf{A} \succ 0, \quad \mathbf{A} \succeq 0.$$

Cũng lại nhân tiện, khoảng cách Mahalanobis có liên quan đến *khoảng cách từ một điểm tới một phân phối xác suất* (from a point to a distribution).

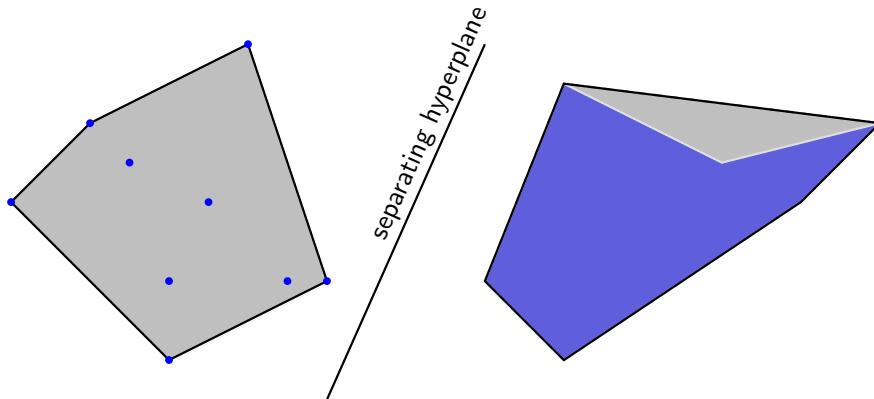
1.2.3 Giao của các tập lồi là một tập lồi.

Việc này có thể nhận dễ nhận thấy với Hình 1.4 (trái). Giao của hai trong ba hoặc cả ba tập lồi đều là các tập lồi.

Việc chứng minh việc này theo Định nghĩa 2 cũng không khó. Nếu $\mathbf{x}_1, \mathbf{x}_2$ thuộc vào giao của các tập lồi, tức thuộc tất cả các tập lồi đã cho, thì $(\theta \mathbf{x}_1 + (1 - \theta) \mathbf{x}_2)$ cũng thuộc vào tất cả các tập lồi, tức thuộc vào giao của chúng!

Từ đó suy ra giao của các *halfspaces* và các *hyperplanes* cũng là một tập lồi. Trong không gian hai chiều, tập lồi này chính là *đa giác lồi*, trong không gian ba chiều, nó có tên là *đa diện lồi*.

Trong không gian nhiều chiều, giao của các *halfspaces* và *hyperplanes* được gọi là **polyhedra**.



Hình 1.5: Trái: Giao của các tập lồi là một tập lồi. Phải: giao của các hyperplanes và halfspace là một tập lồi và được gọi là polyhedron (số nhiều là polyhedra).

Giả sử có m halfspace và p hyperplanes. Mỗi một halfspace, theo như đã trình bày phía trên, có thể viết dưới dạng $\mathbf{a}_i^T \mathbf{x} \leq b_i$, $\forall i = 1, 2, \dots, m$. Mỗi một hyperplane có thể viết dưới dạng: $\mathbf{c}_i^T \mathbf{x} = d_i$, $\forall i = 1, 2, \dots, p$.

Vậy nếu đặt $\mathbf{A} = [\mathbf{a}_1, \mathbf{a}_2, \dots, \mathbf{a}_m]$, $\mathbf{b} = [b_1, b_2, \dots, b_m]^T$, $\mathbf{C} = [\mathbf{c}_1, \mathbf{c}_2, \dots, \mathbf{c}_p]$ và $\mathbf{d} = [d_1, d_2, \dots, d_p]^T$, ta có thể viết polyhedra dưới dạng tập hợp các điểm \mathbf{x} thỏa mãn:

$$\mathbf{A}^T \mathbf{x} \preceq \mathbf{b}, \quad \mathbf{C}^T \mathbf{x} = \mathbf{d}$$

trong đó \preceq là *element-wise*, tức mỗi phần tử trong vế trái nhỏ hơn hoặc bằng phần tử tương ứng trong vế phải.

1.2.4 Convex combination và Convex hulls

Một điểm được gọi là **convex combination** (*tổ hợp lồi*) của các điểm $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_k$ nếu nó có thể viết dưới dạng:

$$\mathbf{x} = \theta_1 \mathbf{x}_1 + \theta_2 \mathbf{x}_2 + \dots + \theta_k \mathbf{x}_k, \quad \text{with } \theta_1 + \theta_2 + \dots + \theta_k = 1$$

Convex hull của một **tập hợp bất kỳ** là tập hợp tất cả các điểm là *convex combination* của tập hợp đó. *Convex hull* là một *convex set*. *Convex hull* của một *convex set* là chính nó. Một cách dễ nhớ, *convex hull* của một tập hợp là một *convex set* **nhỏ nhất** chứa tập hợp đó. Khái niệm **nhỏ nhất** rất khó định nghĩa, nhưng nó cũng là một cách nhớ trực quan.

Hai tập hợp được gọi là *linearly separable* nếu các *convex hulls* của chúng không có điểm chung.

Trong Hình 1.5, convex hull của các điểm màu xanh là vùng màu xám bao với các đa giác lồi. Ở Hình 1.5 phải, vùng màu xám nằm dưới vùng màu xanh.

Định lý siêu phẳng phân chia (Separating hyperplane theorem): Định lý này nói rằng nếu hai *tập lồi không rỗng* \mathcal{C}, \mathcal{D} là *disjoint* (không giao nhau), thì tồn tại vector \mathbf{a} và số b sao cho:

$$\mathbf{a}^T \mathbf{x} \leq b, \forall \mathbf{x} \in \mathcal{C}, \quad \mathbf{a}^T \mathbf{x} \geq b, \forall \mathbf{x} \in \mathcal{D}$$

Tập hợp tất cả các điểm \mathbf{x} thỏa mãn $\mathbf{a}^T \mathbf{x} = b$ chính là một hyperplane. Hyperplane này được gọi là *separating hyperplane*.

Ngoài ra còn nhiều tính chất thú vị của các tập lồi và các phép toán bảo toàn chính chất *lồi* của một tập hợp, các bạn được khuyến khích đọc thêm Chương 2 của cuốn Convex Optimization trong phần tài liệu tham khảo.

1.3 Convex functions

Hẳn các bạn đã nghe tới khái niệm này khi ôn thi đại học môn toán. Khái niệm hàm lồi có quan hệ tới đạo hàm bậc hai và **Bất đẳng thức Jensen** (*nếu bạn chưa nghe tới phần này, không sao, bây giờ bạn sẽ biết*).

1.3.1 Định nghĩa

Để trực quan, trước hết ta xem xét các hàm 1 biến, đồ thị của nó là một đường trong một mặt phẳng. Một hàm số được gọi là *lồi* nếu **tập xác định của nó là một tập lồi** và nếu ta nối hai điểm bất kỳ trên đồ thị hàm số đó, ta được một đoạn thẳng nằm về phía trên hoặc nằm trên đồ thị (xem Hình 1.6).

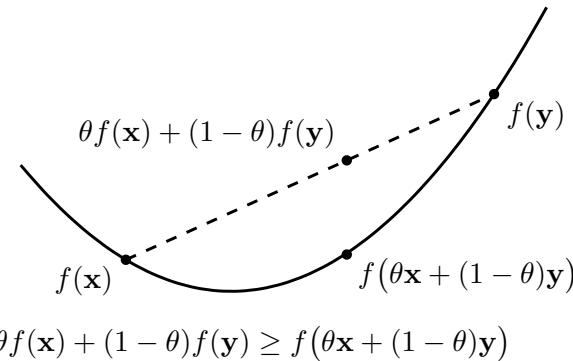
Tập xác định (domain) của một hàm số $f(\cdot)$ thường được ký hiệu là $\text{dom } f$.

Định nghĩa theo toán học:

Định nghĩa convex function: Một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ được gọi là một *hàm lồi* (convex function) nếu $\text{dom } f$ là một *tập lồi*, và:

$$f(\theta \mathbf{x} + (1 - \theta)\mathbf{y}) \leq \theta f(\mathbf{x}) + (1 - \theta)f(\mathbf{y})$$

với mọi $\mathbf{x}, \mathbf{y} \in \text{dom } f, 0 \leq \theta \leq 1$.



Hình 1.6: Định nghĩa hàm lồi. Diễn đạt bằng lời, một hàm số là lồi nếu đoạn thẳng nối 2 điểm bất kỳ trên đồ thị của nó không nằm dưới đồ thị đó.

Điều kiện $\text{dom } f$ là một *tập lồi* là rất quan trọng, vì nếu không có nó, ta không định nghĩa được $f(\theta\mathbf{x} + (1 - \theta)\mathbf{y})$.

Một hàm số f được gọi là **concave** (nếu bạn muốn dịch là *lõm* cũng được, tôi không thích cách dịch này) nếu $-f$ là **convex**. Một hàm số có thể không thuộc hai loại trên. Các hàm tuyến tính vừa *convex*, vừa *concave*.

Định nghĩa strictly convex function: (tiếng Việt có một số tài liệu gọi là *hàm lồi mạnh* hoặc *hàm lồi chặt*) Một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ được gọi là *strictly convex* nếu $\text{dom } f$ là một *tập lồi*, và:

$$f(\theta\mathbf{x} + (1 - \theta)\mathbf{y}) < \theta f(\mathbf{x}) + (1 - \theta)f(\mathbf{y})$$

với mọi $\mathbf{x}, \mathbf{y} \in \text{dom } f, \mathbf{x} \neq \mathbf{y}, 0 < \theta < 1$.

Tương tự với định nghĩa **strictly concave**.

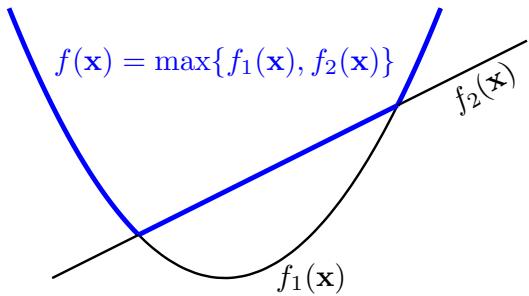
Đây là một điểm quan trọng: Nếu một hàm số là *strictly convex* và có điểm cực trị, thì điểm cực trị đó là duy nhất và cũng là *global minimum*.

1.3.2 Các tính chất cơ bản

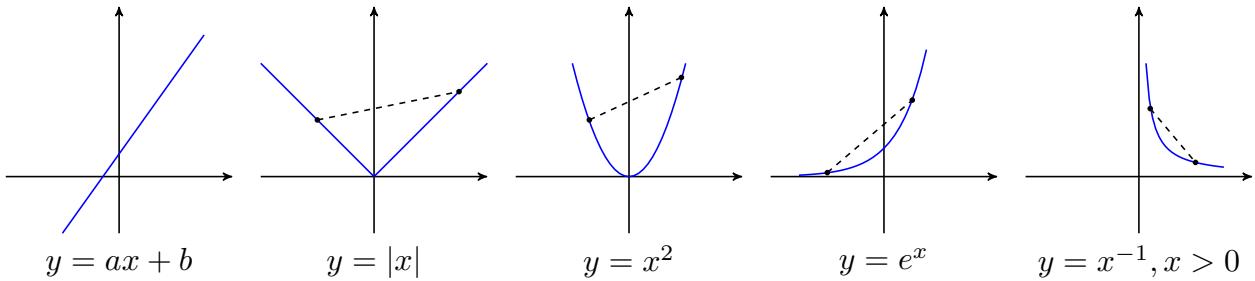
- Nếu $f(\mathbf{x})$ là *convex* thì $af(\mathbf{x})$ là *convex* nếu $a > 0$ và là *concave* nếu $a < 0$. Điều này có thể suy ra trực tiếp từ định nghĩa.
- Tổng của hai *hàm lồi* là một *hàm lồi*, với tập xác định là giao của hai tập xác định kia (nhắc lại rằng giao của hai tập lồi là một tập lồi)
- **Pointwise maximum and supremum:** Nếu các hàm số f_1, f_2, \dots, f_m là *convex* thì:

$$f(\mathbf{x}) = \max\{f_1(\mathbf{x}), f_2(\mathbf{x}), \dots, f_m(\mathbf{x})\}$$

cũng là *convex* trên tập xác định là giao của tất cả các tập xác định của các hàm số trên. Hàm max phía trên cũng có thể thay thế bằng **hàm sup**. Tính chất này có thể chứng minh



Hình 1.7: Ví dụ về Pointwise maximum.
Maximum của các hàm lồi là một hàm lồi.



Hình 1.8: Ví dụ về các convex functions một biến.

được theo Định nghĩa. Bạn cũng có thể nhận ra dựa vào hình ví dụ dưới đây. Mọi đoạn thẳng nối hai điểm bất kỳ trên đường màu xanh đều *không nằm dưới* đường màu xanh.

1.3.3 Ví dụ

Các hàm một biến

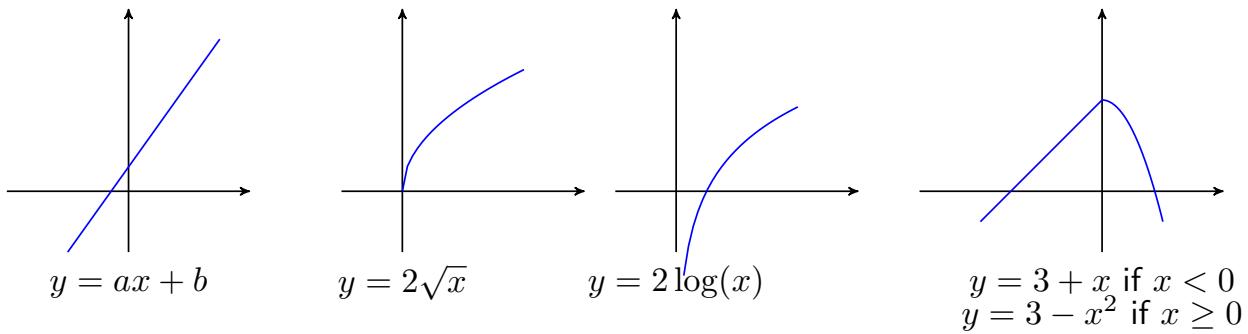
Ví dụ về các *convex functions* một biến:

- Hàm $y = ax + b$ là một *hàm lồi* vì đường nối hai điểm bất kỳ nằm trên chính đồ thị đó.
- Hàm $y = e^{ax}$ với $a \in \mathbb{R}$ bất kỳ.
- Hàm $y = x^a$ trên tập các số thực dương và $a \geq 1$ hoặc $a \leq 0$.
- Hàm *negative entropy* $y = x \log x$ trên tập các số thực dương.

Hình 1.8 minh họa đồ thị của một vài *convex functions*:

Ví dụ về các *concave functions* một biến:

- Hàm $y = ax + b$ là một *concave function* vì $-y$ là một *convex function*.



Hình 1.9: Ví dụ về các concave functions một biến.

- Hàm $y = x^a$ trên tập số dương và $0 \leq a \leq 1$.
- Hàm logarithm $y = \log(x)$ trên tập các số dương.

Hình 1.9 minh họa đồ thị của một vài *concave functions*.

Affine functions

Các hàm số dạng $f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} + b$ vừa là convex, vừa là concave.

Khi biến là một ma trận \mathbf{X} , các hàm affine được định nghĩa có dạng:

$$f(\mathbf{X}) = \text{trace}(\mathbf{A}^T \mathbf{X}) + b$$

trong đó trace là hàm số tính tổng các giá trị trên đường chéo của một ma trận vuông, \mathbf{A} là một ma trận có cùng chiều với \mathbf{X} (để đảm bảo phép nhân ma trận thực hiện được và kết quả là một ma trận vuông).

Quadratic forms

Hàm bậc hai một biến có dạng $f(x) = ax^2 + bx + c$ là convex nếu $a > 0$, là concave nếu $a < 0$.

Với biến là một vector $\mathbf{x} = [x_1, x_2, \dots, x_n]$, một quadratic form là một hàm số có dạng:

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$$

Với \mathbf{A} thường là một ma trận đối xứng, tức $a_{ij} = a_{ji}, \forall i, j$, có số hàng bằng số phần tử của \mathbf{x} , \mathbf{b} là một ma trận bất kỳ cùng chiều với \mathbf{x} và c là một hằng số bất kỳ.

Nếu \mathbf{A} là một ma trận (nửa) xác định dương thì $f(\mathbf{x})$ là một *convex function*.

Nếu \mathbf{A} là một ma trận (nửa) xác định âm, tức $\mathbf{x}^T \mathbf{A} \mathbf{x} \leq 0, \forall \mathbf{x}$, thì $f(\mathbf{x})$ là một *concave function*.

Các bạn có thể tìm đọc về ma trận xác định dương và các tính chất của nó trong sách *Dai số tuyến tính bất kỳ*. Nếu bạn gặp nhiều khó khăn trong phần này, hãy đọc lại kiến thức về *Dai số tuyến tính*, rất rất quan trọng trong *Tối Ưu và Machine Learning*.

Hàm mất mát trong Linear Regression có dạng:

$$\begin{aligned}\mathcal{L}(\mathbf{w}) &= \frac{1}{2} \|\mathbf{y} - \mathbf{Xw}\|_2^2 = \frac{1}{2} (\mathbf{y} - \mathbf{Xw})^T (\mathbf{y} - \mathbf{Xw}) \\ &= \frac{1}{2} \mathbf{w}^T \mathbf{X}^T \mathbf{Xw} - \mathbf{y}^T \mathbf{Xw} + \frac{1}{2} \mathbf{y}^T \mathbf{y}\end{aligned}$$

vì $\mathbf{X}^T \mathbf{X}$ là một ma trận xác định dương, hàm mất mát của Linear Regression chính là một convex function.

Norms

Vâng, lại là norms. Một hàm số bất kỳ thỏa mãn [ba điều kiện của norm](#) đều là một *convex function*. Bạn đọc có thể chứng minh điều này bằng định nghĩa.

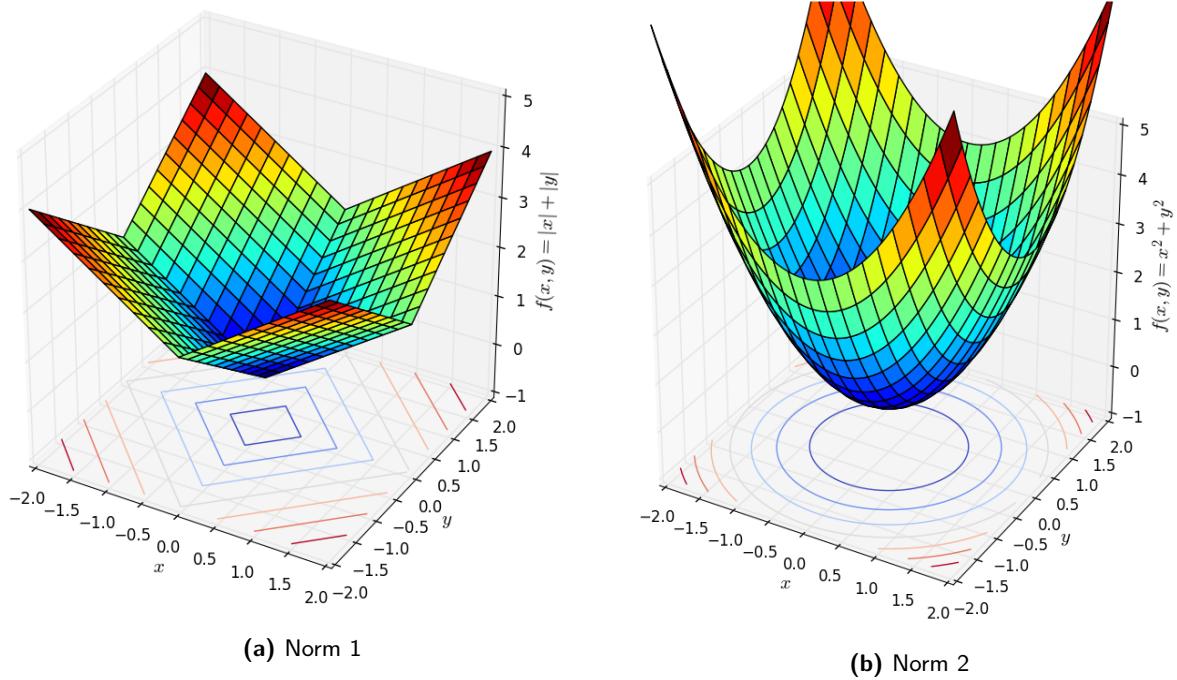
Hình 1.10 minh họa hai ví dụ về norm 1 (trái) và norm 2 (phải) với số chiều là 2 (chiều thứ ba trong hình dưới đây là giá trị của hàm số).

Nhận thấy rằng các bề mặt này đều có *một đáy duy nhất* tương ứng với gốc tọa độ (đây chính là điều kiện đầu tiên của norm). Các hàm *strictly convex* khác cũng có dạng tương tự, tức có một *đáy duy nhất*. Điều này cho thấy nếu ta *thả một hòn bi* ở vị trí bất kỳ trên các bề mặt này, cuối cùng nó sẽ *lăn về đáy*. Nếu liên tưởng tới thuật toán [Gradient Descent](#) thì việc áp dụng thuật toán này vào các bài toán không ràng buộc với *hàm mục tiêu* là *strictly convex* (và giả sử là khả vi, tức có đạo hàm) sẽ cho kết quả rất tốt nếu *learning rate* không quá lớn. Đây chính là một trong các lý do vì sao các *convex functions* là quan trọng, cũng là lý do vì sao tôi dành bài viết này chỉ để nói về *convexity*. (Bạn đọc được khuyến khích đọc hai bài về [Gradient Descent](#) trong blog này).

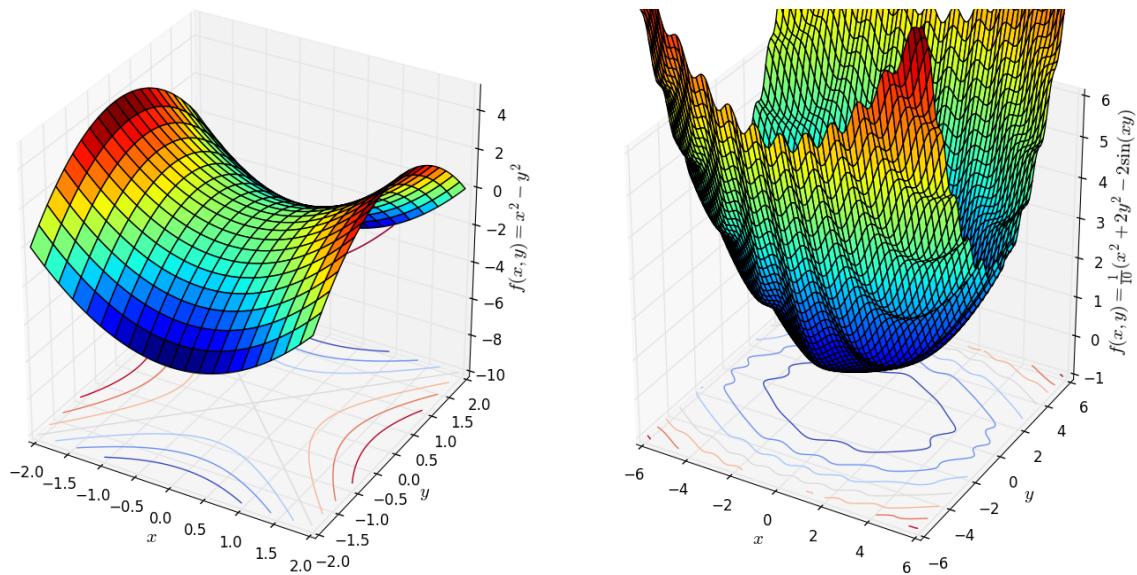
Tiện đây, tôi cũng lấy thêm hai ví dụ về các hàm không phải convex (cũng không phải concave). Hàm thứ nhất $f(x, y) = x^2 - y^2$ là một hyperbolic, hàm thứ hai $f(x, y) = \frac{1}{10}(x^2 + 2y^2 - 2\sin(xy))$.

Contours - level sets Với các hàm số phức tạp hơn, khi vẽ các mặt trong không gian ba chiều sẽ khó tưởng tượng hơn, tức khó nhìn được tính *convexity* của nó. Một phương pháp thường được sử dụng là dùng *contours* hay *level sets*. Tôi cũng đã đề cập đến khái niệm này trong Bài Gradient Descent, phần [đường đồng mức](#).

Contours là cách mô tả các mặt trong không gian ba chiều bằng cách chiếu nó xuống không gian hai chiều. Trong không gian hai chiều, các điểm thuộc cùng một *đường* tương ứng với

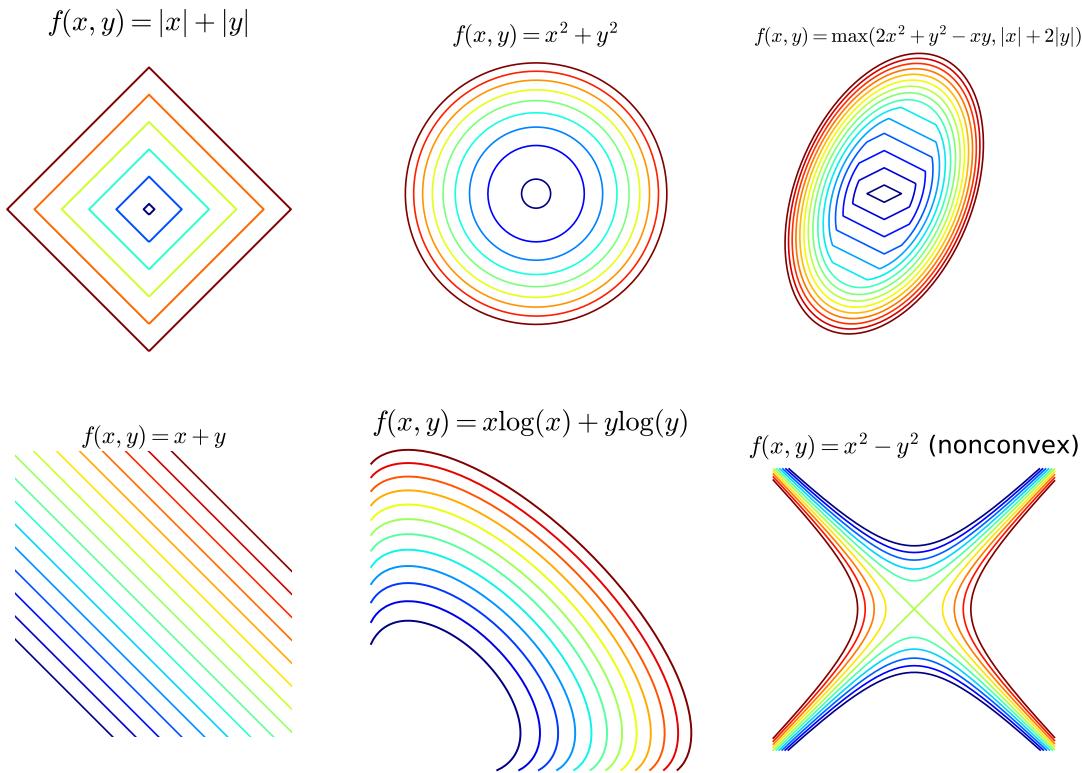


Hình 1.10: Ví dụ về mặt của các norm hai biến.



Hình 1.11: Ví dụ về các hàm hai biến không convex.

các điểm làm cho hàm số có giá trị bằng nhau. Mỗi đường đó còn được gọi là một *level set*. Trong Hình 1.10 và Hình 1.11, các đường của các mặt lên mặt phẳng Oxy chính là các *level sets*. Một cách hiểu khác, mỗi đường *level set* là một *vết cắt* nếu ta cắt các bề mặt bởi một mặt phẳng song song với mặt phẳng Oxy .



Hình 1.12: Ví dụ về Countours. Các đường màu càng xanh đậm thì tương ứng với các giá trị càng nhỏ, các đường màu càng đỏ đậm thì tương ứng các giá trị càng lớn.

Khi thể hiện một hàm số hai biến để kiểm tra tính convexity của nó, hoặc để tìm điểm cực trị của nó, người ta thường vẽ *contours* thay vì vẽ các mặt trong không gian ba chiều. Hình 5.6 minh họa một vài ví dụ về contours.

Ở hàng trên, các đường *level sets* là các đường khép kín (closed). Khi các đường kín này tập trung nhỏ dần ở một điểm thì các điểm đó là các điểm cực trị. Với các *convex functions* như trong ba ví dụ này, chỉ có 1 điểm cực trị và đó cũng là điểm làm cho hàm số đạt giá trị nhỏ nhất (global optimal). Nếu để ý, bạn sẽ thấy các đường khép kín này tạo thành một *vùng lồi!*.

Ở hàng dưới, các đường không phải khép kín. Hình bên trái tương ứng với một hàm tuyến tính $f(x, y) = x + y$ và đó là một *convex function*. Hình ở giữa cũng là một *convex function* (bạn có thể chứng minh điều này sau khi tính đạo hàm bậc hai, tôi sẽ nói ở phía dưới) nhưng các level sets là các *đường không kín*. Hàm này có log nên tập xác định là góc phần tư thứ nhất tương ứng với các tọa độ dương (chú ý rằng tập hợp các điểm có tọa độ dương cũng là một *tập lồi*). Các *đường không kín* này nếu kết hợp với trục Ox, Oy sẽ tạo thành biên của các *tập lồi*. Hình cuối cùng là contours của một hàm hyperbolic, hàm này không phải là *hàm lồi*.

1.3.4 α - sublevel sets

Định nghĩa: α - sublevel set của một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ được định nghĩa là:

$$\mathcal{C}_\alpha = \{\mathbf{x} \in \text{dom } f \mid f(\mathbf{x}) \leq \alpha\}$$

Tức tập hợp các điểm trong tập xác định của f mà tại đó hàm số đạt giá trị nhỏ hơn hoặc bằng α .

Quay lại với Hình 5.6, hàng trên, các α - sublevel sets chính là phần bị bao bởi các level sets.

Ở hàng dưới, bên trái, các α - sublevel sets chính là phần nửa mặt phẳng phía dưới xác định bởi các đường thẳng level sets. Ở hình giữa, các α - sublevel sets chính là các vùng bị giới hạn bởi các trục tọa độ và các level sets.

Hàng dưới, bên phải, các α - sublevel sets hơi khó tưởng tượng chút. Với $\alpha > 0$, các level sets là các đường màu vàng hoặc đỏ. Các α - sublevel sets tương ứng là phần *bị b López vào trong*, giới hạn bởi các đường đỏ cùng màu. Các vùng này, có thể dễ nhận thấy, là *không lồi*.

Định lý: Nếu một hàm số là lồi thì *mọi* α - sublevel sets của nó là lồi. Điều ngược lại chưa chắc đã đúng, tức nếu các α - sublevel sets của một hàm số là *lồi* thì hàm số đó chưa chắc đã *lồi*.

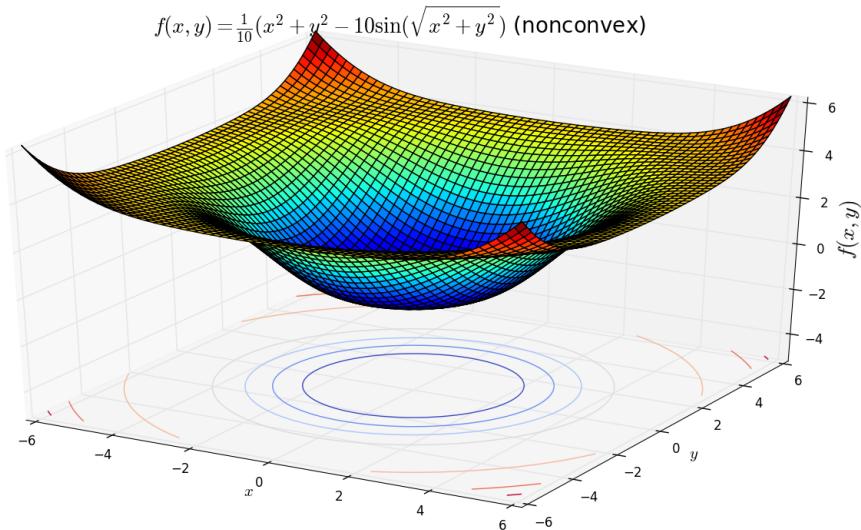
Điều này chỉ ra rằng nếu tồn tại một giá trị α sao cho một α - sublevel set của một hàm số là *không lồi*, thì hàm số đó là *không lồi* (*không lồi* nhưng không có nghĩa là *concave*, chú ý). Vậy nên Hyperbolic không phải là hàm lồi.

Các ví dụ ở Hình 5.6, trừ hình cuối cùng, đều tương ứng với các hàm lồi.

Một ví dụ về việc một hàm số không *convex* nhưng mọi α - sublevel sets là *convex* là hàm $f(x, y) = -e^{x+y}$. Hàm này có mọi α - sublevel sets là nửa mặt phẳng - là *convex*, nhưng nó không phải là *convex* (trong trường hợp này nó là *concave*).

Dưới đây là một ví dụ khác về việc một hàm số có mọi α - sublevel sets là *lồi* nhưng không phải *hàm lồi*.

Mọi α - sublevel sets của hàm số này đều là các hình tròn - *convex* nhưng hàm số đó không phải là *lồi*. Vì có thể tìm được hai điểm trên mặt này sao cho đoạn thẳng nối hai điểm nằm hoàn toàn phía dưới của mặt (một điểm ở *cạnh* và 1 điểm ở *đáy* chẳng hạn).



Hình 1.13: Mọi alpha-sublevel sets là convex sets nhưng hàm số là nonconvex.

Những hàm số có tập xác định là một *tập lồi* và có mọi α - sublevel sets là *lồi* được gọi chung là *quasiconvex*. Mọi *convex function* đều là *quasiconvex* nhưng ngược lại không đúng. Định nghĩa chính thức của *quasiconvex function* được phát biểu như sau:

Quasiconvex function: Một hàm số $f : \mathcal{C} \rightarrow \mathbb{R}$ với \mathcal{C} là một tập con *lồi* của \mathbb{R}^n được gọi là *quasiconvex* nếu với mọi $\mathbf{x}, \mathbf{y} \in \mathcal{C}$ và mọi $\theta \in [0, 1]$, ta có:

$$f(\theta\mathbf{x} + (1 - \theta)\mathbf{y}) \leq \max\{f(\mathbf{x}), f(\mathbf{y})\}$$

Định nghĩa này khác với định nghĩa về *convex function* một chút.

1.3.5 Kiểm tra tính chất lồi dựa vào đạo hàm.

Có một cách để nhận biết một hàm số khả vi có là hàm lồi hay không dựa vào các đạo hàm bậc nhất hoặc đạo hàm bậc hai của nó.

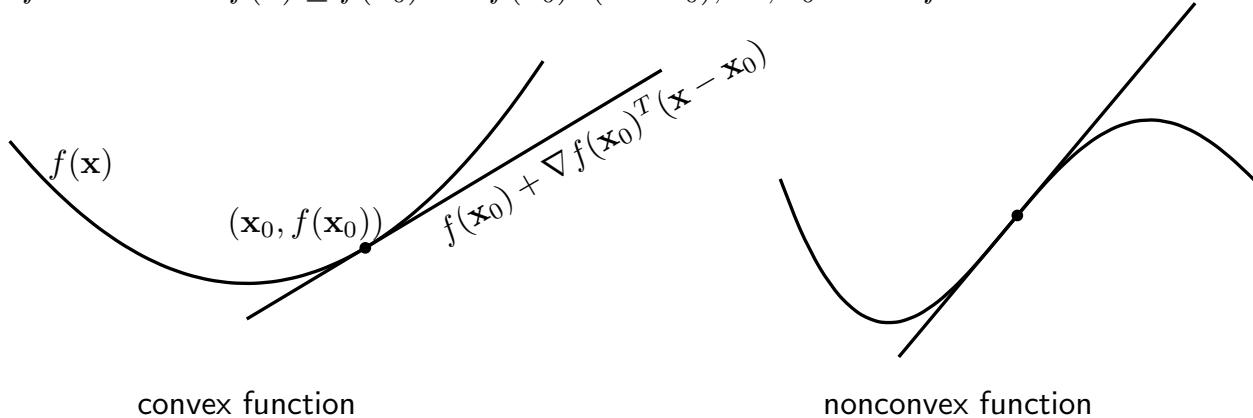
First-order condition

Trước hết chúng ta định nghĩa phương trình đường (mặt) tiếp tuyến của một hàm số f khả vi tại một điểm nằm trên đồ thị (mặt) của hàm số đó $(\mathbf{x}_0, f(\mathbf{x}_0))$. Với hàm một biến, bạn đọc đã quen thuộc:

$$y = f'(\mathbf{x}_0)(x - \mathbf{x}_0) + f(\mathbf{x}_0)$$

f is differentiable with convex domain

f is convex iff $f(\mathbf{x}) \geq f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0), \forall \mathbf{x}, \mathbf{x}_0 \in \text{dom } f$



Hình 1.14: Kiểm tra tính convexity dựa vào đạo hàm bậc nhất. Trái: hàm lồi vì tiếp tuyến tại mọi điểm đều nằm dưới đồ thị hàm số đó, phải: hàm không lồi.

Với hàm nhiều biến, đặt $\nabla f(\mathbf{x}_0)$ là gradient của hàm số f tại điểm \mathbf{x}_0 , phương trình mặt tiếp tuyến được cho bởi:

$$y = \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) + f(\mathbf{x}_0)$$

First-order condition nói rằng: Giả sử hàm số f có tập xác định là một tập lồi, có đạo hàm tại mọi điểm trên tập xác định đó. Khi đó, hàm số f là *lồi nếu và chỉ nếu* với mọi \mathbf{x}, \mathbf{x}_0 trên tập xác định của hàm số đó, ta có:

$$f(\mathbf{x}) \geq f(\mathbf{x}_0) + \nabla f(\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) \quad (1.4)$$

Tương tự như thế, một hàm số là *strictly convex* nếu dấu bằng trong (1.4) xảy ra khi và chỉ khi $\mathbf{x} = \mathbf{x}_0$.

Nói một cách trực quan hơn, một hàm số là lồi nếu đường (mặt) tiếp tuyến tại một điểm bất kỳ trên đồ thị (mặt) của hàm số đó **nằm dưới** đồ thị (mặt) đó.

(Đừng quên điều kiện về tập xác định là lồi.)

Dưới đây là ví dụ về *hàm lồi* và *hàm không lồi*.

Hàm bên trái là một hàm lồi. Hàm bên phải không phải là hàm lồi vì đồ thị của nó vừa nằm trên, vừa nằm dưới tiếp tuyến.

(*iff* là viết tắt của *if and only if*)

Ví dụ: Nếu ma trận đối xứng \mathbf{A} là xác định dương thì hàm số $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$ là *hàm lồi*.

Chứng minh: Đạo hàm bậc nhất của hàm số trên là:

$$\nabla f(\mathbf{x}) = 2\mathbf{A}\mathbf{x}$$

Vậy *first-order condition* có thể viết dưới dạng (chú ý rằng \mathbf{A} là một ma trận đối xứng):

$$\begin{aligned} \mathbf{x}^T \mathbf{A} \mathbf{x} &\geq 2(\mathbf{A}\mathbf{x}_0)^T(\mathbf{x} - \mathbf{x}_0) + \mathbf{x}_0^T \mathbf{A} \mathbf{x}_0 \\ \Leftrightarrow \mathbf{x}^T \mathbf{A} \mathbf{x} &\geq 2\mathbf{x}_0^T \mathbf{A} \mathbf{x} - \mathbf{x}_0^T \mathbf{A} \mathbf{x}_0 \\ \Leftrightarrow (\mathbf{x} - \mathbf{x}_0)^T \mathbf{A} (\mathbf{x} - \mathbf{x}_0) &\geq 0 \end{aligned}$$

Bất đẳng thức cuối cùng là đúng dựa trên định nghĩa của một ma trận xác định dương. Vậy hàm số $f(\mathbf{x}) = \mathbf{x}^T \mathbf{A} \mathbf{x}$ là *hàm lồi*.

First-order condition ít được sử dụng để tìm tính chất lồi của một hàm số, thay vào đó, người ta thường dùng *Second-order condition* với các hàm có đạo hàm tới bậc hai.

Second-order condition

Với hàm nhiều biến, tức biến là một vector, giả sử có chiều là d , đạo hàm bậc nhất của nó là một vector cũng có chiều là d . Đạo hàm bậc hai của nó là một ma trận vuông có chiều là $d \times d$. Đạo hàm bậc hai của hàm số $f(\mathbf{x})$ được ký hiệu là $\nabla^2 f(\mathbf{x})$. Đạo hàm bậc hai còn được gọi là *Hessian*.

Second-order condition: Một hàm số có đạo hàm bậc hai là *convex* nếu $\text{dom } f$ là *convex* và Hessian của nó là một ma trận *nửa xác định dương* với mọi \mathbf{x} trong tập xác định:

$$\nabla^2 f(\mathbf{x}) \succeq 0.$$

Nếu Hessian là một ma trận xác định dương thì hàm số đó *strictly convex*. Tương tự, nếu Hessian là một ma trận xác định âm thì hàm số đó là *strictly concave*.

Với hàm số một biến $f(x)$, điều kiện này tương đương với $f''(x) \geq 0$ với mọi x thuộc tập xác định (và tập xác định là *lồi*).

Ví dụ:

- Hàm *negative entropy* $f(x) = x \log(x)$ là *strictly convex* vì tập xác định là $x > 0$ là một tập lồi và $f''(x) = 1/x$ là một số dương với mọi x thuộc tập xác định.
- Hàm $f(x) = x^2 + 5 \sin(x)$ không là hàm lồi vì đạo hàm bậc hai $f''(x) = 2 - 5 \sin(x)$ có thể nhận giá trị âm.
- Hàm *cross entropy* là một hàm *strictly convex*. Xét ví dụ đơn giản với chỉ hai xác suất x và $1 - x$ với a là một hằng số thuộc đoạn $[0, 1]$ và $0 < x < 1$: $f(x) = -(a \log(x) + (1 - a) \log(1 - x))$ có đạo hàm bậc hai là $\frac{a}{x^2} + \frac{1-a}{(1-x)^2}$ là một số dương.

- Nếu \mathbf{A} là một ma trận xác định dương thì $f(\mathbf{x}) = \frac{1}{2}\mathbf{x}^T \mathbf{A} \mathbf{x}$ là lồi vì Hessian của nó chính là \mathbf{A} là một ma trận xác định dương.
- Xét hàm số *negative entropy* với hai biến: $f(x, y) = x \log(x) + y \log(y)$ trên tập các giá trị dương của x và y . Hàm số này có đạo hàm bậc nhất là $[\log(x) + 1, \log(y) + 1]^T$ và Hessian là $\begin{bmatrix} 1/x & 0 \\ 0 & 1/y \end{bmatrix}$, là một ma trận đường chéo với các thành phần trên đường chéo là dương nên là một ma trận xác định dương. Vậy *negative entropy* là một hàm *strictly convex*. (Chú ý rằng một ma trận là xác định dương nếu các trị riêng của nó đều dương. Với một ma trận là ma trận đường chéo thì các trị riêng của nó chính là các thành phần trên đường chéo.)

Ngoài ra còn nhiều tính chất thú vị của các *hàm lồi*, các bạn được khuyến khích đọc thêm Chương 3 của cuốn Convex Optimization trong phần tài liệu tham khảo.

1.4 Tóm tắt

- Machine Learning và Optimization có quan hệ mật thiết với nhau. Trong Optimization, Convex Optimization là quan trọng nhất. Một bài toán là convex optimization nếu *hàm mục tiêu* là convex và tập hợp các điểm thỏa mãn các điều kiện ràng buộc là một *convex set*.
- Trong *convex set*, mọi đoạn thẳng nối hai điểm bất kỳ trong tập đó sẽ nằm hoàn toàn trong tập đó. Tập hợp các giao điểm của các *convex sets* là một *convex set*.
- Một hàm số là *convex* nếu đoạn thẳng nối hai điểm bất kỳ trên đồ thị hàm số đó không nằm dưới đồ thị đó.
- Một hàm số khả vi là *convex* nếu tập xác định của nó là *convex* và đường (mặt) tiếp tuyến *không nằm phía trên* đồ thị (bề mặt) của hàm số đó.
- Các norms là các hàm lồi, được sử dụng nhiều trong tối ưu.

1.5 Tài liệu tham khảo

[1] Convex Optimization – Boyd and Vandenberghe, Cambridge University Press, 2004.

Convex Optimization Problems

Nội dung trong bài viết này chủ yếu được dịch từ Chương 4 của cuốn *Convex Optimization* trong phần Tài liệu tham khảo.

2.1 Giới thiệu

Tôi xin bắt đầu bài viết này bằng ba bài toán khá gần với thực tế:

2.1.1 Bài toán nhà xuất bản

Bài toán

Một nhà xuất bản (NXB) nhận được đơn hàng 600 bản của cuốn "Machine Learning cơ bản" tới Thái Bình và 400 bản tới Hải Phòng. NXB đó có 800 cuốn ở kho Nam Định và 700 cuốn ở kho Hải Dương. Giá chuyển phát một cuốn sách từ Nam Định tới Thái Bình là 50,000 VND (50k), tới Hải Phòng là 100k. Giá chuyển phát một cuốn từ Hải Dương tới Thái Bình là 150k, trong khi tới Hải Phòng chỉ là 40k. Hỏi để tốn ít chi phí chuyển phát nhất, công ty đó nên phân phối mỗi kho chuyển bao nhiêu cuốn tới mỗi địa điểm?

Phân tích

Để cho đơn giản, ta xây dựng bảng số lượng chuyển sách từ nguồn tới đích như sau:

Tổng chi phí (objective function) sẽ là $f(x, y, z, t) = 5x + 10y + 15z + 4t$. Các điều kiện ràng buộc (constraints) viết dưới dạng biểu thức toán học là:

- Chuyển 600 cuốn tới Thái Bình: $x + z = 600$.

Nguồn	Đích	Đơn giá ($\times 10k$)	Số lượng
Nam Định	Thái Bình	5	x
Nam Định	Hải Phòng	10	y
Hải Dương	Thái Bình	15	z
Hải Dương	Hải Phòng	4	t

- Chuyển 400 cuốn tới Hải Phòng: $y + t = 400$.
- Lấy từ kho Nam Định không quá 800: $x + y \leq 800$.
- Lấy từ kho Hải Dương không quá 700: $z + t \leq 700$.
- x, y, z, t là các số tự nhiên. Ràng buộc là số tự nhiên sẽ khiến cho bài toán rất khó giải nếu số lượng biến là rất lớn. Với bài toán này, ta giả sử rằng x, y, z, t là các số thực dương. Khi tìm được nghiệm, nếu chúng không phải là số tự nhiên, ta sẽ lấy các giá trị tự nhiên gần nhất.

Vậy ta cần giải bài toán tối ưu sau đây:

Bài toán NXB:

$$(x, y, z, t) = \arg \min_{x, y, z, t} 5x + 10y + 15z + 4t \quad (2.1)$$

$$\text{subject to:} \quad x + z = 600 \quad (2.2)$$

$$y + t = 400 \quad (2.3)$$

$$x + y \leq 800 \quad (2.4)$$

$$z + t \leq 700 \quad (2.5)$$

$$x, y, z, t \geq 0 \quad (2.6)$$

Nhận thấy rằng hàm mục tiêu (objective function) là một hàm tuyến tính của các biến x, y, z, t . Các điều kiện ràng buộc đều có dạng *hyperplanes* hoặc *halfspaces*, đều là các ràng buộc tuyến tính (linear constraints). Bài toán tối ưu với cả *objective function* và *constraints* đều là *linear* được gọi là **Linear Programming (LP)**. Dạng tổng quát và cách thức lập trình để giải một bài toán thuộc loại này sẽ được cho trong phần sau của bài viết này.

Nghiệm cho bài toán này có thể nhận thấy ngay là $x = 600, y = 0, z = 0, t = 400$. Nếu ràng buộc nhiều hơn và số biến nhiều hơn, chúng ta cần một lời giải có thể tính được bằng cách lập trình.

2.1.2 Bài toán canh tác

Bài toán

Một anh nông dân có tổng cộng 10ha (10 hecta) đất canh tác. Anh dự tính trồng cà phê và hồ tiêu trên số đất này với tổng chi phí cho việc trồng này là không quá 16T (triệu đồng). Chi phí để trồng cà phê là 2T cho 1ha, để trồng hồ tiêu là 1T/ha/. Thời gian trồng cà phê là 1 ngày/ha và hồ tiêu là 4 ngày/ha; trong khi anh chỉ có thời gian tổng cộng là 32 ngày. Sau khi trừ tất cả các chi phí (bao gồm chi phí trồng cây), mỗi ha cà phê mang lại lợi nhuận 5T, mỗi ha hồ tiêu mang lại lợi nhuận 3T. Hỏi anh phải trồng như thế nào để tối đa lợi nhuận? (*Các số liệu có thể vô lý vì chúng đã được chọn để bài toán ra nghiệm đẹp*)

Phân tích

Gọi x và y lần lượt là số ha cà phê và hồ tiêu mà anh nông dân nên trồng. Lợi nhuận anh ấy thu được là $f(x, y) = 5x + 3y$ (triệu đồng).

Các ràng buộc trong bài toán này là:

- Tổng diện tích trồng không vượt quá 10: $x + y \leq 10$.
- Tổng chi phí trồng không vượt quá 16T: $2x + y \leq 16$.
- Tổng thời gian trồng không vượt quá 32 ngày: $x + 4y \leq 32$.
- Diện tích cà phê và hồ tiêu là các số không âm: $x, y \geq 0$.

Vậy ta có bài toán tối ưu sau đây:

Bài toán canh tác:

$$(x, y) = \arg \max_{x,y} 5x + 3y \quad (2.7)$$

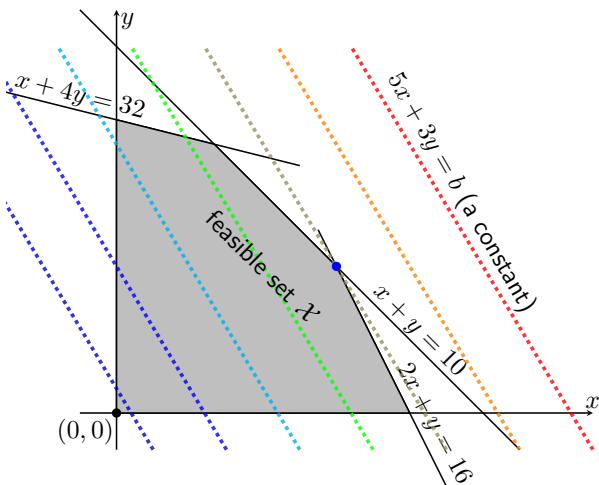
$$\text{subject to: } \begin{aligned} x + y &\leq 10 \\ 2x + y &\leq 16 \end{aligned} \quad (2.8)$$

$$2x + y \leq 16 \quad (2.9)$$

$$x + 4y \leq 32 \quad (2.10)$$

$$x, y \geq 0 \quad (2.11)$$

Bài toán này hơi khác một chút là ta cần *tối đa hàm mục tiêu* thay vì tối thiểu nó. Việc chuyển bài toán này về bài toán *tối thiểu* có thể được thực hiện đơn giản bằng cách đổi dấu



Hình 2.1: Minh họa nghiệm cho bài toán canh tách. Phần ngũ giác màu xám thể hiện tập hợp các điểm thoả mãn các ràng buộc. Các đường nét đứt thể hiện các đường đồng mức của hàm mục tiêu với màu càng đỏ tương ứng với giá trị càng cao. Nghiệm tìm được chính là điểm màu xanh, là giao điểm của hình ngũ giác xám và đường đồng mức ứng với giá trị cao nhất.

hàm mục tiêu. Khi đó hàm mục tiêu vẫn là *linear*, các ràng buộc vẫn là các *linear constraints*, ta lại có một bài toán **Linear Programming (LP)** nữa.

Bạn cũng có thể dựa vào Hình 2.1 để suy ra nghiệm của bài toán.

Vùng màu xám có dạng *polyhedron* (trong trường hợp này là đa giác) chính là tập hợp các điểm thoả mãn các ràng buộc từ (2.8) đến (2.11). Các đường nét đứt có màu chính là các đường đồng mức của hàm mục tiêu $5x + 3y$, mỗi đường ứng với một giá trị khác nhau với đường càng đỏ ứng với giá trị càng cao. Một cách trực quan, nghiệm của bài toán có thể tìm được bằng cách di chuyển đường nét đứt màu xanh về phía bên phải (phía làm cho giá trị của hàm mục tiêu lớn hơn) đến khi nó không còn điểm chung với phần đa giác màu xám nữa.

Có thể nhận thấy nghiệm của bài toán chính là điểm màu xanh là giao điểm của hai đường thẳng $x + y = 10$ và $2x + y = 16$. Giải hệ phương trình này ta có $x^* = 6$ và $y^* = 4$. Tức anh nông dân nên trồng 6ha cà phê và 4ha hồ tiêu. Lúc đó lợi nhuận thu được là $5x^* + 3y^* = 42$ triệu đồng, trong khi anh chỉ mất thời gian là 22 ngày. (*Chịu tính toán cái là khác ngay, làm ít, hưởng nhiều*).

Với nhiều biến hơn và nhiều ràng buộc hơn, chúng ta liệu có thể vẽ được hình như thế này để nhìn ra nghiệm hay không? Câu trả lời của tôi là nên tìm một công cụ để với nhiều biến hơn và với các ràng buộc khác nhau, chúng ta có thể tìm ra nghiệm gần như ngay lập tức.

2.1.3 Bài toán đóng thùng

Bài toán

Một công ty phải chuyển $400 m^3$ cát tới địa điểm xây dựng ở bên kia sông bằng cách thuê một chiếc xà lan. Ngoài chi phí vận chuyển một lượt đi về là 100k của chiếc xà lan, công ty đó phải thiết kế một thùng hình hộp chữ nhật đặt trên xà lan để đựng cát. Chiếc thùng này

không cần nắp, chi phí cho các mặt xung quanh là $1T/m^2$, cho mặt đáy là $2T/m^2$. Kích thước của chiếc thùng đó như thế nào để tổng chi phí vận chuyển là nhỏ nhất. Để cho đơn giản, giả sử cát chỉ được đổ ngang hoặc thấp hơn với phần trên của thành thùng, không có ngọn. Giả sử thêm rằng xà lan *rộng vô hạn* và chúa được sức nặng vô hạn, giả sử này khiến bài toán dễ giải hơn.

Phân tích

Giả sử chiếc thùng cần làm có chiều dài là x (m), chiều rộng là y và chiều cao là z . Thể tích của thùng là xyz (đơn vị là m^3). Có hai loại chi phí là:

- *Chi phí thuê xà lan:* số chuyến xà lan phải thuê là $\frac{400}{xyz}$ (ta hãy tạm giả sử rằng đây là một số tự nhiên, việc làm tròn này sẽ không thay đổi kết quả đáng kể vì chi phí vận chuyển một chuyến là nhỏ so với chi phí làm thùng). Số tiền phải trả cho xà lan sẽ là $0.1 \frac{400}{xyz} = \frac{40}{xyz}$.
- *Chi phí làm thùng:* Diện tích xung quanh của thùng là $2(x + y)z$. Diện tích đáy là xy . Vậy tổng chi phí làm thùng là $2(x + y)z + 2xy = 2(xy + yz + zx)$.

Tổng toàn bộ chi phí là $f(x, y, z) = 40x^{-1}y^{-1}z^{-1} + 2(xy + yz + zx)$. Điều kiện ràng buộc duy nhất là kích thước thùng phải là các số dương. Vậy ta có bài toán tối ưu sau:

Bài toán vận chuyển:

$$(x, y) = \arg \min_{x, y, z} 40x^{-1}y^{-1}z^{-1} + 2(xy + yz + zx)$$

subject to: $x, y, z > 0$ (2.12)

Bài toán này thuộc loại **Geometric Programming (GP)**. Định nghĩa của GP và cách dùng công cụ tối ưu sẽ được trình bày trong phần sau của bài viết.

Nhận thấy rằng bài này hoàn toàn có thể dùng bất đẳng thức Cauchy để giải được, nhưng tôi vẫn muốn một lời giải cho bài toán tổng quát sao cho có thể lập trình được.

(Lời giải:

$$f(x, y, z) = \frac{20}{xyz} + \frac{20}{xyz} + 2xy + 2yz + 2zx \geq 5\sqrt[5]{3200}$$

dấu bằng xảy ra khi và chỉ khi $x = y = z = \sqrt[5]{10}$. Bài này có lẽ hợp với các kỳ thi vì dữ kiện quá đẹp. Cá nhân tôi thích các đề bài ra kiểu này hơn là yêu cầu đi tìm giá trị nhỏ nhất của một biểu thức nhầm chán, nhiều học sinh cho rằng không biết học bất đẳng thức để làm gì!)

Nếu có các ràng buộc về kích thước của thùng và trọng lượng mà xà lan tải được thì có thể tìm được lời giải đơn giản như thế này không?

Những bài toán trên đây đều là các bài toán tối ưu. Chính xác hơn nữa, chúng đều là các bài toán tối ưu lồi (*convex optimization problems*) như các bạn sẽ thấy ở phần sau. Và việc tìm lời giải có thể không mấy khó khăn, thậm chí giải bằng tay cũng có thể ra kết quả. Tuy nhiên, mục đích của bài viết này không phải là hướng dẫn các bạn giải các bài toán trên *bằng tay*, mà là cách nhận diện các bài toán và đưa chúng về các dạng mà các toolboxes sẵn có có thể giúp chúng ta. Trên thực tế, lượng dữ kiện và số biến cần tối ưu lớn hơn nhiều, chúng ta không thể giải các bài toán trên *bằng tay* được.

Trước hết, chúng ta cần hiểu các khái niệm về *convex optimization problems* và tại sao *convex* lại quan trọng. (Bạn đọc có thể đọc tới [phần 4](#) nếu không muốn biết các khái niệm và định lý toán trong phần 2 và 3.)

2.2 Nhắc lại bài toán tối ưu

2.2.1 Các khái niệm cơ bản

Tôi xin nhắc lại bài toán tối ưu ở dạng tổng quát:

$$\begin{aligned} \mathbf{x}^* = \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned} \tag{2.13}$$

Phát biểu bằng lời: Tìm giá trị của biến \mathbf{x} để tối thiểu hàm $f_0(\mathbf{x})$ trong số các giá trị của \mathbf{x} thoả mãn các điều kiện ràng buộc. Ta có bảng các tên gọi tiếng Anh và tiếng Việt như trong [Bảng 2.1](#).

Ngoài ra:

- Khi $m = p = 0$, bài toán (2.13) được gọi là *unconstrained optimization problem* (bài toán tối ưu không ràng buộc).
- \mathcal{D} chỉ là tập xác định, tức giao của tất cả các tập xác định của mọi hàm số xuất hiện trong bài toán. Tập hợp các điểm thoả mãn mọi điều kiện ràng buộc, thông thường, là một tập con của \mathcal{D} được gọi là *feasible set* hoặc *constraint set*. Khi *feasible set* là một tập rỗng thì ta nói bài toán tối ưu (2.13) là *infeasible*. Nếu một điểm nằm trong *feasible set*, ta gọi điểm đó là *feasible*.
- *Optimal value* (*giá trị tối ưu*) của bài toán tối ưu (2.13) được định nghĩa là:

Ký hiệu	Tiếng Anh	Tiếng Việt
$\mathbf{x} \in \mathbb{R}^n$	optimization variable	biến tối ưu
$f_0 : \mathbb{R}^n \rightarrow \mathbb{R}$	objective/loss/cost/function	hàm mục tiêu
$f_i(\mathbf{x}) \leq 0$	inequality constraint	bất đẳng thức ràng buộc
$f_i : \mathbb{R}^n \rightarrow \mathbb{R}$	inequality constraint function	hàm bất đẳng thức ràng buộc
$h_j(\mathbf{x}) = 0$	equality constraint	đẳng thức ràng buộc
$h_j : \mathbb{R}^n \rightarrow \mathbb{R}$	equality constraint function	hàm đẳng thức ràng buộc
$\mathcal{D} = \bigcap_{i=0}^m \text{dom } f_i \cap \bigcap_{j=1}^p \text{dom } h_j$	domain	tập xác định

Bảng 2.1: Bảng các thuật ngữ trong các bài toán tối ưu.

$$p^* = \inf \{f_0(\mathbf{x}) | f_i(\mathbf{x}) \leq 0, i = 1, \dots, m; h_j(\mathbf{x}) = 0, j = 1, \dots, p\}$$

trong đó \inf là viết tắt của hàm **infimum**. p^* có thể nhận các giá trị $\pm\infty$. Nếu bài toán là *infeasible*, ta coi $p^* = +\infty$, Nếu hàm mục tiêu không bị chặn dưới (*unbounded below*) trong tập xác định, ta coi $p^* = -\infty$.

2.2.2 Optimal and locally optimal points

Một điểm \mathbf{x}^* được gọi là một điểm *optimal point* (*điểm tối ưu*), hoặc là *nghiệm* của bài toán (2.13) nếu \mathbf{x}^* là *feasible* và $f_0(\mathbf{x}^*) = p^*$. Tất hợp tất cả các *optimal points* được gọi là *optimal set*.

Nếu *optimal set* là một tập *không rỗng*, ta nói bài toán (2.13) là *solvable* (*giải được*). Ngược lại, nếu *optimal set* là một tập rỗng, ta nói *optimal value* là *không thể đạt được* (*not attained/not achieved*).

Ví dụ: xét hàm mục tiêu $f(x) = 1/x$ với ràng buộc $x > 0$. *Optimal value* của bài toán này là $p^* = 0$ nhưng *optimal set* là một tập rỗng vì không có giá trị nào của x để hàm mục tiêu đạt giá trị 0. Lúc này ta nói *giá trị tối ưu* là *không đạt được*.

Với hàm một biến, một điểm là *cực tiểu* của một hàm số nếu tại đó, hàm số đạt giá trị nhỏ nhất trong một lân cận (và lân cận này thuộc tập xác định của hàm số). Trong không gian 1 chiều, *lân cận* được hiểu là trị tuyệt đối của hiệu 2 điểm nhỏ hơn một giá trị nào đó.

Trong toán tối ưu (thường là không gian nhiều chiều), ta gọi một điểm \mathbf{x} là **locally optimal** (*cực tiểu*) nếu tồn tại một giá trị (thường được gọi là bán kính) R sao cho:

$$\begin{aligned} f_0(\mathbf{x}) &= \inf \{f_0(\mathbf{z}) | f_i(\mathbf{z}) \leq 0, i = 1, \dots, m, \\ h_j(\mathbf{z}) &= 0, j = 1, \dots, p, \|\mathbf{z} - \mathbf{x}\|_2 \leq R\} \end{aligned} \quad (2.14)$$

Nếu một điểm *feasible* \mathbf{x} thoả mãn $f_i(\mathbf{x}) = 0$, ta nói rằng bất đẳng thức ràng buộc thứ $i : f_i(\mathbf{x}) = 0$ là *active*. Nếu $f_i(\mathbf{x}) < 0$, ta nói rằng ràng buộc này là *inactive* tại \mathbf{x} .

2.2.3 Một vài lưu ý

Mặc dù trong định nghĩa bài toán tối ưu (2.13) là cho bài toán *tối thiểu hàm mục tiêu* với các ràng buộc thoả mãn các điều kiện nhỏ hơn hoặc bằng 0, các bài toán tối ưu với *tối đa hàm mục tiêu* và điều kiện ràng buộc ở dạng khác đều có thể đưa về được dạng này:

- $\max f_0(\mathbf{x}) \Leftrightarrow \min -f_0(\mathbf{x})$.
- $f_i(\mathbf{x}) \leq g(\mathbf{x}) \Leftrightarrow f_i(\mathbf{x}) - g(\mathbf{x}) \leq 0$.
- $f_i(\mathbf{x}) \geq 0 \Leftrightarrow -f_i(\mathbf{x}) \leq 0$.
- $a \leq f_i(\mathbf{x}) \leq b \Leftrightarrow f_i(\mathbf{x}) - b \leq 0$ và $a - f_i(\mathbf{x}) \leq 0$.
- $f_i(\mathbf{x}) \leq 0 \Leftrightarrow f_i(\mathbf{x}) + s_i = 0$ và $s_i \geq 0$. s_i được gọi là *slack variable*. Phép biến đổi đơn giản này trong nhiều trường hợp lại tỏ ra hiệu quả vì bất đẳng thức $s_i \geq 0$ thường dễ giải quyết hơn là $f_i(\mathbf{x}) \leq 0$.

2.3 Bài toán tối ưu lồi

Trong toán tối ưu, chúng ta đặc biệt quan tâm tới những bài toán mà hàm mục tiêu là một hàm lồi, và *feasible set* cũng là một tập lồi.

2.3.1 Định nghĩa

Một *bài toán tối ưu lồi* (*convex optimization problem*) là một bài toán tối ưu có dạng:

$$\begin{aligned} \mathbf{x}^* = \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ \mathbf{a}_j^T \mathbf{x} - b_j = 0, \quad j = 1, \dots, n \end{aligned} \tag{2.15}$$

trong đó f_0, f_1, \dots, f_m là các hàm lồi.

So với bài toán tối ưu (2.13), bài toán tối ưu lồi (2.15) có thêm ba điều kiện nữa:

- *Hàm mục tiêu* là một *hàm lồi*.

- Các hàm bất đẳng thức ràng buộc f_i là các hàm lồi.
- Hàm đẳng thức ràng buộc h_j là *affine* (hàm *linear* cộng với một hằng số nữa được gọi là *affine*).

Một vài nhận xét:

- Tập hợp các điểm thoả mãn $h_j(\mathbf{x}) = 0$ là một tập lồi vì nó có dạng một *hyperplane*.
- Khi f_i là một *hàm lồi* thì tập hợp các điểm thoả mãn $f_i(\mathbf{x}) \leq 0$ chính là *0-sublevel set* của f_i và là một *tập lồi*.
- Như vậy tập hợp các điểm thoả mãn mọi điều kiện ràng buộc chính là *giao điểm* của các *tập lồi*, vì vậy nó là một *tập lồi*.

Vậy, trong một bài toán tối ưu lồi, ta *tối thiểu một hàm mục tiêu lồi* trên một *tập lồi*.

2.3.2 Cực tiểu của bài toán tối ưu lồi chính là điểm tối ưu.

Tính chất quan trọng nhất của bài toán tối ưu lồi chính là bất kỳ *locally optimal point* chính là một điểm *(globally) optimal point*.

Tính chất quan trọng này có thể chứng minh bằng phản chứng như sau. Gọi \mathbf{x}_0 là một điểm *locally optimal*, tức:

$$f_0(\mathbf{x}_0) = \inf\{f_0(\mathbf{x}) \mid \mathbf{x} \text{ is feasible}, \|\mathbf{x} - \mathbf{x}_0\|_2 \leq R\}$$

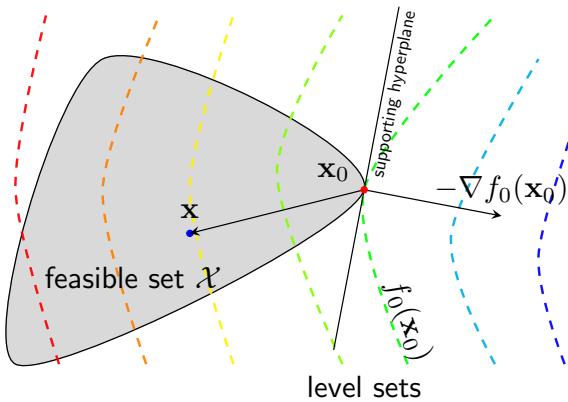
với $R > 0$ nào đó. Giả sử \mathbf{x}_0 không phải là *globally optimal point*, tức tồn tại một *feasible point* \mathbf{y} sao cho $f(\mathbf{y}) < f(\mathbf{x}_0)$ (hiển nhiên rằng \mathbf{y} không nằm trong lân cận đang xét). Ta có thể tìm được $\theta \in [0, 1]$ đủ nhỏ sao cho $\mathbf{z} = (1 - \theta)\mathbf{x}_0 + \theta\mathbf{y}$ nằm trong lân cận của \mathbf{x}_0 , tức $\|\mathbf{z} - \mathbf{x}_0\|_2 < R$. Chú ý rằng \mathbf{z} cũng là một *feasible point* vì *feasible set* là một *tập lồi*. Hơn nữa, vì *hàm mục tiêu* f_0 là một hàm lồi, ta có:

$$f_0(\mathbf{z}) = f_0((1 - \theta)\mathbf{x}_0 + \theta\mathbf{y}) \quad (2.16)$$

$$\leq (1 - \theta)f_0(\mathbf{x}_0) + \theta f_0(\mathbf{y}) \quad (2.17)$$

$$< (1 - \theta)f_0(\mathbf{x}_0) + \theta f_0(\mathbf{x}_0) \quad (2.18)$$

$$= f_0(\mathbf{x}_0) \quad (2.19)$$



Hình 2.2: Biểu diễn hình học của điều kiện tối ưu cho hàm mục tiêu khả vi. Các đường nét đứt có màu tương ứng với các level sets (đường đồng mức).

điều này mâu thuẫn với giả thiết \mathbf{x}_0 là một điểm cực tiểu. Vậy giả sử sai, tức \mathbf{x}_0 chính là *globally optimal point* và ta có điều phải chứng minh.

Chứng minh bằng lời: giả sử một điểm cực tiểu không phải là điểm làm cho hàm số đạt giá trị nhỏ nhất. Với điều kiện *feasible set* và *hàm mục tiêu* là lồi, ta luôn tìm được một điểm khác trong lân cận của điểm cực tiểu đó sao cho giá trị của hàm mục tiêu tại điểm mới này nhỏ hơn giá trị của hàm mục tiêu tại điểm cực tiểu. Sự mâu thuẫn này chỉ ra rằng với một bài toán tối ưu lồi, điểm cực tiểu phải là điểm làm cho hàm số đạt giá trị nhỏ nhất.

2.3.3 Điều kiện tối ưu cho hàm mục tiêu khả vi

Nếu hàm mục tiêu f_0 là khả vi (differentiable), theo [first-order condition](#), với mọi $\mathbf{x}, \mathbf{y} \in \text{dom } f_0$, ta có:

$$f_0(\mathbf{x}) \geq f_0(\mathbf{x}_0) + \nabla f_0(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \quad (2.20)$$

Đặt \mathcal{X} là *feasible set*. **Điều kiện cần và đủ** để một điểm $\mathbf{x}_0 \in \mathcal{X}$ là *optimal point* là:

$$\nabla f_0(\mathbf{x}_0)^T (\mathbf{x} - \mathbf{x}_0) \geq 0, \quad \forall \mathbf{x} \in \mathcal{X} \quad (2.21)$$

Tôi xin được bỏ qua việc chứng minh điều kiện cần và đủ này, bạn đọc có thể tìm trong trang 139-140 của cuốn Convex Optimization trong Tài liệu tham khảo.

Một cách hình học, điều kiện này nói rằng: Nếu \mathbf{x}_0 là điểm *optimal* thì với mọi $\mathbf{x} \in \mathcal{X}$, vector đi từ \mathbf{x}_0 tới \mathbf{x} hợp với vector $-\nabla f_0(\mathbf{x}_0)$ một góc tù (Xem [Hình 2.2](#)). Nói cách khác, nếu ta vẽ *mặt tiếp tuyến* của hàm mục tiêu tại \mathbf{x}_0 thì mọi điểm *feasible* nằm về một phía so với *mặt tiếp tuyến* này. Hơn nữa, *feasible set* nằm về phía làm cho hàm mục tiêu đạt giá trị cao hơn $f_0(\mathbf{x}_0)$. *Mặt tiếp tuyến* này chính là *supporting hyperplane* của *feasible set* tại điểm \mathbf{x}_0 . Nhắc lại rằng khi vẽ các *level set*, tôi thường dùng màu lam để chỉ giá trị nhỏ, màu đỏ để chỉ giá trị lớn của hàm số.

(Một mặt phẳng đi qua một điểm trên biên của một tập hợp sao cho mọi điểm trong tập hợp đó nằm về một phía (hoặc nằm trên) so với mặt phẳng đó được gọi là *supporting hyperplane*

(siêu phẳng hỗ trợ). Nếu một tập hợp là *lồi*, tồn tại *supporting hyperplane* tại mọi điểm trên biên của nó.)

Nếu tồn tại một điểm \mathbf{x}_0 trong *feasible set* sao cho $\nabla f_0(\mathbf{x}_0) = 0$, đây chính là *optimal point*. Điều này dễ hiểu vì đó chính là điểm làm cho gradient bằng 0, tức điểm cực tiểu của hàm mục tiêu. Nếu $\nabla f_0(\mathbf{x}_0) \neq 0$, vector $-\nabla f_0(\mathbf{x}_0)$ chính là *vector pháp tuyến* của *supporting hyperplane* tại \mathbf{x}_0 .

2.3.4 Giới thiệu thư viện CVXOPT

[CVXOPT](#) là một thư viện miễn phí trên Python giúp giải rất nhiều các bài toán trong cuốn sách Convex Optimization ở phần Tài liệu tham khảo. Tác giả thứ hai của cuốn sách này, Lieven Vandenberghe, chính là đồng tác giả của thư viện này. Hướng dẫn cài đặt, tài liệu hướng dẫn, và các ví dụ mẫu của thư viện này cũng có đầy đủ trên trang web [CVXOPT](#).

Trong phần còn lại của bài viết, tôi sẽ giới thiệu 3 bài toán rất cơ bản trong Convex Optimization: Linear Programming, Quadratic Programming, và Geometric Programming. Tôi cũng sẽ cùng các bạn lập trình để giải các ví dụ đã nêu ở phần đầu bài viết dựa trên thư viện CVXOPT này.

2.4 Linear Programming

Chúng ta cùng bắt đầu với lớp các bài toán đơn giản nhất trong Convex Optimization - Linear Programming (LP, một số tài liệu cũng gọi là *Linear Program*), trong đó hàm mục tiêu f_0 và hàm bất đẳng thức ràng buộc $f_i, i = 1, \dots, m$ đều là các hàm tuyến tính cộng với một hằng số (tức [hàm affine](#)).

2.4.1 Dạng tổng quát của LP

A general LP:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} + d \\ \text{subject to: } & \mathbf{Gx} \leq \mathbf{h} \\ & \mathbf{Ax} = \mathbf{b} \end{aligned} \tag{2.22}$$

Trong đó $\mathbf{G} \in \mathbb{R}^{m \times n}$, $\mathbf{h} \in \mathbb{R}^m$ và, $\mathbf{A} \in \mathbb{R}^{p \times n}$, $\mathbf{b} \in \mathbb{R}^p$, $\mathbf{c}, \mathbf{x} \in \mathbb{R}^n$ và d là một số vô hướng (số vô hướng này có thể bỏ qua vì nó không ảnh hưởng tới nghiệm của bài toán tối ưu, nó chỉ

làm thay đổi giá trị của hàm mục tiêu). Nhắc lại rằng ký hiệu \preceq nghĩa là mỗi phần tử trong vector (ma trận) ở về trái nhỏ hơn hoặc bằng phần tử tương ứng trong vector (ma trận) ở về phải.

Chú ý rằng nhiều bất đẳng thức dạng $\mathbf{g}_i \mathbf{x} \leq h_i$, với \mathbf{g}_i là các vector hàng, có thể viết gộp dưới dạng $\mathbf{Gx} \preceq \mathbf{h}$ trong đó mỗi hàng của \mathbf{G} ứng với một \mathbf{g}_i , mỗi phần tử của \mathbf{h} tương ứng với một h_i .

2.4.2 Dạng tiêu chuẩn của LP

Trong dạng tiêu chuẩn (*standard form*) LP, các bất đẳng thức ràng buộc chỉ là điều kiện các nghiệm có thành phần không âm:

A standard form LP:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \\ \text{subject to: } & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{x} \succeq \mathbf{0} \end{aligned} \tag{2.23}$$

Bài toán (2.22) có thể đưa về bài toán (2.23) bằng cách đặt thêm biến *slack* s .

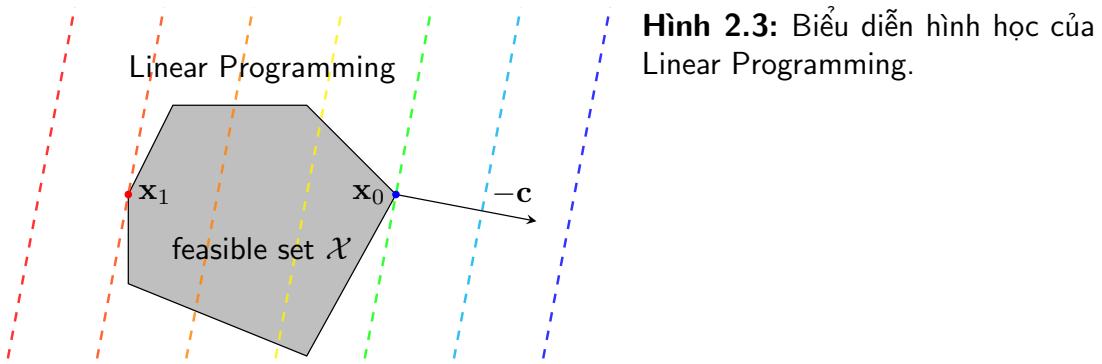
$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}, \mathbf{s}} \mathbf{c}^T \mathbf{x} \\ \text{subject to: } & \mathbf{Ax} = \mathbf{b} \\ & \mathbf{Gx} + \mathbf{s} = \mathbf{h} \\ & \mathbf{s} \succeq \mathbf{0} \end{aligned} \tag{2.24}$$

Tiếp theo, nếu ta biểu diễn \mathbf{x} dưới dạng hiệu của hai vector mà thành phần của nó đều không âm, tức: $\mathbf{x} = \mathbf{x}^+ - \mathbf{x}^-$, với $\mathbf{x}^+, \mathbf{x}^- \succeq \mathbf{0}$. Ta có thể tiếp tục viết lại (2.24) dưới dạng:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}^+, \mathbf{x}^-, \mathbf{s}} \mathbf{c}^T \mathbf{x}^+ - \mathbf{c}^T \mathbf{x}^- \\ \text{subject to: } & \mathbf{Ax}^+ - \mathbf{Ax}^- = \mathbf{b} \\ & \mathbf{Gx}^+ - \mathbf{Gx}^- + \mathbf{s} = \mathbf{h} \\ & \mathbf{x}^+ \succeq \mathbf{0}, \mathbf{x}^- \succeq \mathbf{0}, \mathbf{s} \succeq \mathbf{0} \end{aligned} \tag{2.25}$$

Tới đây, bạn đọc có thể thấy rằng (2.25) có thể viết gọn lại như (2.23).

Bài toán nhà xuất bản và Bài toán canh tác trong phần đầu của bài viết này chính là các LP.



2.4.3 Minh họa bằng hình học của bài toán LP

Các bài toán LP có thể được minh họa như Hình 2.3.

Điểm \mathbf{x}_0 chính là điểm làm cho hàm mục tiêu đạt giá trị nhỏ nhất, điểm \mathbf{x}_1 chính là điểm làm cho hàm mục tiêu đạt giá trị lớn nhất. Với các bài toán LP, nghiệm, nếu có, thường là một điểm ở *dỉnh* của polyheron hoặc là một *mặt* của polyhedron đó (trong trường hợp các đường level sets của hàm mục tiêu song song với mặt đó, và trên mặt đó, hàm mục tiêu đạt giá trị tối ưu).

Về LP, các bạn có thể tìm thấy rất nhiều tài liệu cả tiếng Việt (Quy hoạch tuyến tính) và tiếng Anh. Có rất nhiều các bài toán trong thực tế có thể đưa về dạng LP. Phương pháp thường được dùng để giải bài toán này có tên là *simplex* (*đơn hình*). Tôi sẽ không đề cập đến các phương pháp này, thay vào đó, tôi sẽ hướng dẫn các bạn dùng thư viện CVXOPT để giải quyết các bài toán thuộc dạng này.

2.4.4 Giải LP bằng CVXOPT

Tôi sẽ dùng thư viện CVPOPT để giải Bài toán canh tác ở phía trên. Nhắc lại bài toán này:

Bài toán canh tác:

$$\begin{aligned}
 (x, y) = \arg \max_{x,y} & 5x + 3y \\
 \text{subject to:} & x + y \leq 10 \\
 & 2x + y \leq 16 \\
 & x + 4y \leq 32 \\
 & x, y \geq 0
 \end{aligned} \tag{2.26}$$

Các điều kiện ràng buộc có thể viết lại dưới dạng $\mathbf{G}\mathbf{x} \preceq \mathbf{h}$, trong đó:

$$\mathbf{G} = \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 4 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \quad \mathbf{h} = \begin{bmatrix} 10 \\ 16 \\ 32 \\ 0 \\ 0 \end{bmatrix}$$

Lời giải cho bài toán này khi dùng CVXOPT là:

```
from cvxopt import matrix, solvers
c = matrix([-5., -3.])
G = matrix([[1., 2., 1., -1., 0.], [1., 1., 4., 0., -1.]])
h = matrix([10., 16., 32., 0., 0.])

solvers.options['show_progress'] = False
sol = solvers.lp(c, G, h)

print('Solution')
print(sol['x'])
```

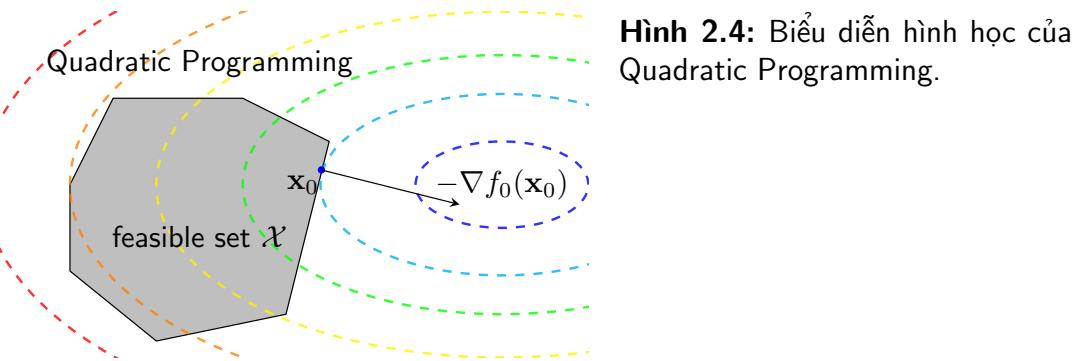
```
Solution:
[ 6.00e+00]
[ 4.00e+00]
```

Nghiệm này chính là nghiệm mà tôi đã tìm được trong phần đầu của bài viết.

Một vài lưu ý:

- Hàm `solvers.lp` của `cvxopt` giải bài toán (2.24).
- Trong bài toán của chúng ta, vì ta cần tìm giá trị lớn nhất nên ta phải đổi hàm mục tiêu về dạng $-5x - 3y$. Chính vì vậy mà `c = matrix([-5., -3.])`.
- Hàm `matrix` nhận đầu vào là một `list` (trong Python), `list` này thể hiện một vector cột. Nếu muốn biểu diễn một ma trận, đầu vào của `matrix` là một `list` của `list`, trong đó mỗi `list` bên trong thể hiện một vector cột của ma trận đó.
- Các hằng số trong bài toán cần ở dạng số thực. Nếu chúng là các số nguyên, ta cần thêm dấu `.` vào sau các số đó để thể hiện đó là số thực. (Tôi thấy điểm này hơi thừa, nhưng nếu không có dấu `.` thì chương trình sẽ báo lỗi.)
- Với đẳng thức ràng buộc $\mathbf{Ax} = \mathbf{b}$, `solvers.lp` lấy giá trị mặc định của `A` và `b` là `None`, tức nếu không khai báo thì nghĩa là không có đẳng thức ràng buộc nào.
- Với các tùy chọn khác, bạn đọc có thể tìm trong Tài liệu của CVXOPT.

Việc giải Bài toán nhà xuất bản bằng CVXOPT xin nhường lại cho bạn đọc như một bài tập đơn giản.



Hình 2.4: Biểu diễn hình học của Quadratic Programming.

2.5 Quadratic Programming

2.5.1 Định nghĩa bài toán Quadratic Programming

Một dạng Convex Optimization mà các bạn sẽ gặp rất nhiều trong các bài sau của blog là *Quadratic Programming* (QP, hoặc *Quadratic Program*). Khác biệt duy nhất của QP so với LP là hàm mục tiêu có dạng *Quadratic form*:

Quadratic Programming:

$$\begin{aligned} \mathbf{x} = \arg \min_{\mathbf{x}} & \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + \mathbf{r} \\ \text{subject to: } & \mathbf{G} \mathbf{x} \preceq \mathbf{h} \\ & \mathbf{A} \mathbf{x} = \mathbf{b} \end{aligned} \quad (2.27)$$

Trong đó $\mathbf{P} \in \mathbb{S}_+^n$ (tập các ma trận vuông nửa xác định dương có số cột là n), $\mathbf{G} \in \mathbb{R}^{m \times n}$, $\mathbf{A} \in \mathbb{R}^{p \times n}$. **Điều kiện \mathbf{P} là nửa xác định dương để đảm bảo hàm mục tiêu là convex.**

Chúng ta có thể thấy rằng LP chính là một trường hợp đặc biệt của QP với $\mathbf{P} = \mathbf{0}$.

Diễn đạt bằng lời: trong QP, chúng ta tối thiểu một hàm quadratic lồi trên một *polyhedron* (Xem Hình 2.4).

2.5.2 Ví dụ về QP

Bài toán vui: Có một hòn đảo mà hình dạng của nó có dạng một đa giác lồi. Một con thuyền ở ngoài biển thì cần đi theo hướng nào để tới đảo nhanh nhất, giả sử rằng tốc độ của sóng và gió bằng 0.

Bài toán khoảng cách từ một điểm tới một polyhedron được phát biểu như sau:

Cho một polyhedron được biểu diễn bởi $\mathbf{Ax} \preceq \mathbf{b}$ và một điểm \mathbf{u} , tìm điểm \mathbf{x} thuộc polyhedron đó sao cho khoảng cách Euclidean giữa \mathbf{x} và \mathbf{u} là nhỏ nhất.

Bài toán này có thể phát biểu như sau:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \frac{1}{2} \|\mathbf{x} - \mathbf{u}\|_2^2 \\ \text{subject to: } &\quad \mathbf{Gx} \preceq \mathbf{h} \end{aligned}$$

Hàm mục tiêu đạt giá trị nhỏ nhất bằng 0 nếu \mathbf{u} nằm trong polyheron đó và *optimal point* chính là $\mathbf{x} = \mathbf{u}$. Khi \mathbf{u} không nằm trong polyhedron, ta viết:

$$\frac{1}{2} \|\mathbf{x} - \mathbf{u}\|_2^2 = \frac{1}{2} (\mathbf{x} - \mathbf{u})^T (\mathbf{x} - \mathbf{u}) = \frac{1}{2} \mathbf{x}^T \mathbf{x} - \mathbf{u}^T \mathbf{x} + \frac{1}{2} \mathbf{u}^T \mathbf{u}$$

Biểu thức này có dạng hàm mục tiêu như trong (2.27) với $\mathbf{P} = \mathbf{I}$, $\mathbf{q} = -\mathbf{u}$, $\mathbf{r} = \frac{1}{2} \mathbf{u}^T \mathbf{u}$, trong đó \mathbf{I} là ma trận đơn vị.

2.5.3 Ví dụ về giải QP bằng CVXOPT

Xét bài toán sau đây (xem Hình 2.5):

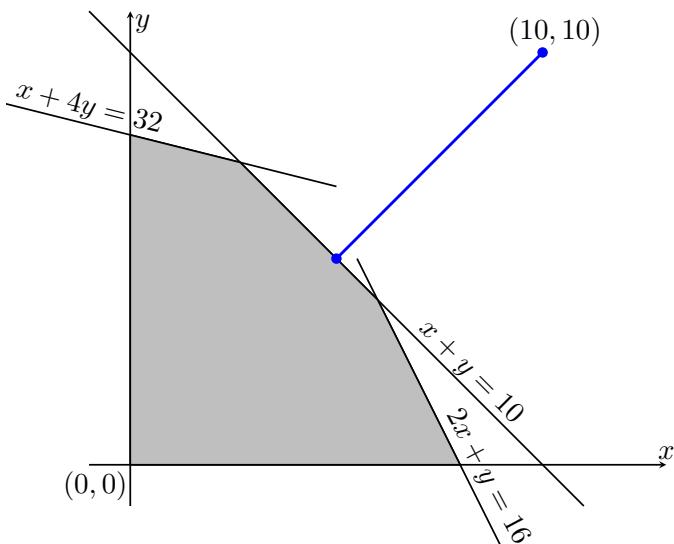
$$\begin{aligned} (x, y) &= \arg \min_{x, y} (x - 10)^2 + (y - 10)^2 \\ \text{subject to: } &\quad \begin{bmatrix} 1 & 1 \\ 2 & 1 \\ 1 & 4 \\ -1 & 0 \\ 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} \preceq \begin{bmatrix} 10 \\ 16 \\ 32 \\ 0 \\ 0 \end{bmatrix} \end{aligned}$$

Feasible set trong bài toán này tôi lấy trực tiếp từ Bài toán canh tác và $\mathbf{u} = [10, 10]^T$. Bài toán này có thể được giải bằng CVXOPT như sau:

```
from cvxopt import matrix, solvers
P = matrix([[1., 0.], [0., 1.]])
q = matrix([-10., -10.])
G = matrix([[1., 2., 1., -1., 0.], [1., 1., 4., 0., -1.]])
h = matrix([10., 16., 32., 0., 0.])

solvers.options['show_progress'] = False
sol = solvers.qp(P, q, G, h)

print('Solution:')
print(sol['x'])
```



Hình 2.5: Ví dụ về khoảng cách giữa một điểm và một polyhedron.

Solution:
[5.00e+00]
[5.00e+00]

Trong các thuật toán Machine Learning, các bạn sẽ gặp các bài toán về tìm *hình chiếu* (projection) của một điểm lên một *tập lồi* nói chung rất nhiều. Tới từng phần, tôi sẽ đề cập hướng giải quyết của các bài toán đó.

2.6 Geometric Programming

Trong mục này, chúng ta sẽ thấy một lớp các bài toán *không lồi* khi nhìn vào hàm mục tiêu và các hàm ràng buộc, nhưng có thể được biến đổi về dạng *lồi* bằng một vài kỹ thuật.

Trước hết, chúng ta cần có một vài định nghĩa:

2.6.1 Monomials và posynomials

Một hàm số $f : \mathbb{R}^n \rightarrow \mathbb{R}$ với tập xác định $\text{dom } f = \mathbf{R}_{++}^n$ (tất cả các phần tử đều là số dương) có dạng:

$$f(\mathbf{x}) = c x_1^{a_1} x_2^{a_2} \dots x_n^{a_n} \quad (2.28)$$

trong đó $c > 0$ và $a_i \in \mathbb{R}$, được gọi là một *monomial function* (khái niệm này khá giống với *đơn thức* khi tôi học lớp 8, nhưng khi đó SGK định nghĩa với c bất kỳ và a_i là các số tự nhiên).

Tổng của các monomials:

$$f(\mathbf{x}) = \sum_{k=1}^K c_k x_1^{a_{1k}} x_2^{a_{2k}} \dots x_n^{a_{nk}}$$

trong đó các $c_k > 0$ được gọi là *posynomial function* (*đa thức*), hoặc đơn giản là *posynomial*.

2.6.2 Geometric Programming

Một bài toán tối ưu có dạng:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } & f_i(x) \leq 1, \quad i = 1, 2, \dots, m \\ & h_j(x) = 1, \quad j = 1, 2, \dots, p \end{aligned} \tag{2.29}$$

trong đó f_0, f_1, \dots, f_m là các *posynomials* và h_1, \dots, h_p là các *monomials*, được gọi là *Geometric Programming* (GP). Điều kiện $\mathbf{x} \succ 0$ được ẩn đi.

Chú ý rằng nếu f là một *posynomial*, h là một *monomial* thì f/h là một *posynomial*.

Ví dụ:

$$\begin{aligned} (x, y, z) &= \arg \min_{x, y, z} x/y \\ \text{subject to: } & 1 \leq x \leq 2 \\ & x^3 + 2y/z \leq \sqrt{y} \\ & x/y = z \end{aligned} \tag{2.30}$$

Có thể được viết lại dưới dạng GP:

$$\begin{aligned} (x, y, z) &= \arg \min_{x, y, z} xy^{-1} \\ \text{subject to: } & x^{-1} \leq 1 \\ & (1/2)x \leq 1 \\ & x^3 y^{-1/2} + 2y^{1/2} z^{-1} \leq 1 \\ & xy^{-1} z^{-1} = 1 \end{aligned} \tag{2.31}$$

Bài toán này rõ ràng là *nonconvex* vì cả hàm mục tiêu và điều kiện ràng buộc đều không lồi.

2.6.3 Biến đổi GP về dạng convex

GP có thể được biến đổi về dạng lồi như sau:

Đặt $y_i = \log(x_i)$, tức $x_i = \exp(y_i)$. Nếu f là một *monomial function* của \mathbf{x} thì:

$$f(\mathbf{x}) = c(\exp(y_1))^{a_1} \dots (\exp(y_n))^{a_n} = \exp(\mathbf{a}^T \mathbf{y} + b)$$

với $b = \log(c)$. Lúc này, hàm số $g(y) = \exp(\mathbf{a}^T \mathbf{y} + b)$ là một hàm lồi theo \mathbf{y} . (Bạn đọc có thể chứng minh theo định nghĩa rằng hợp của hai hàm lồi là một hàm lồi. Trong trường hợp này, hàm \exp và hàm *affine* trên đều là các hàm lồi.)

Tương tự như thế, *posynomial* trong đẳng thức (24) có thể viết dưới dạng:

$$f(\mathbf{x}) = \sum_{k=1}^K \exp(\mathbf{a}_k^T \mathbf{y} + b_k)$$

trong đó $\mathbf{a}_k = [a_{1k}, \dots, a_{nk}]^T$ và $b_k = \log(c_k)$. Lúc này, *posynomial* đã được viết dưới dạng tổng của các hàm \exp của các hàm *affine* (và vì vậy là một hàm lồi, nhớ lại rằng tổng của các hàm lồi là một hàm lồi).

Bài toán GP (2.29) được viết lại dưới dạng:

$$\begin{aligned} \mathbf{y} &= \arg \min_{\mathbf{y}} \sum_{k=1}^{K_0} \exp(\mathbf{a}_{0k}^T \mathbf{y} + b_{0k}) \\ \text{subject to: } &\sum_{k=1}^{K_i} \exp(\mathbf{a}_{ik}^T \mathbf{y} + b_{ik}) \leq 1, \quad i = 1, \dots, m \\ &\exp(\mathbf{g}_j^T \mathbf{y} + h_j) = 1, \quad j = 1, \dots, p \end{aligned} \tag{2.32}$$

với $\mathbf{a}_{ik} \in \mathbb{R}^n, i = 1, \dots, p$ và $\mathbf{g}_i \in \mathbb{R}^n$.

Với chú ý rằng hàm số $\log \sum_{i=1}^m \exp(g_i(\mathbf{x}))$ là một hàm *lồi* nếu g_i là các hàm *lồi* (tôi xin bỏ qua phần chứng minh), ta có thể viết lại bài toán (2.32) dưới dạng *lồi* bằng cách lấy log của các hàm như sau:

GP in convex form:

$$\begin{aligned} \text{minimize}_{\mathbf{y}} \tilde{f}_0(\mathbf{y}) &= \log \left(\sum_{k=1}^{K_0} \exp(\mathbf{a}_{0k}^T \mathbf{y} + b_{0k}) \right) \\ \text{subject to: } \tilde{f}_i(\mathbf{y}) &= \log \left(\sum_{k=1}^{K_i} \exp(\mathbf{a}_{ik}^T \mathbf{y} + b_{ik}) \right) \leq 0, \quad i = 1, \dots, m \\ \tilde{h}_j(\mathbf{y}) &= \mathbf{g}_j^T \mathbf{y} + h_j = 0, \quad j = 1, \dots, p \end{aligned} \tag{2.33}$$

Lúc này, ta có thể nói rằng GP tương đương với một bài toán tối ưu lồi vì hàm mục tiêu và các hàm bất đẳng thức ràng buộc trong (2.33) đều là hàm lồi, đồng thời điều kiện đẳng thức cuối cùng chính là dạng *affine*. Dạng này thường được gọi là *geometric program in convex form* (để phân biệt nó với định nghĩa của GP).

2.6.4 Giải GP bằng CVXOPT

Chúng ta quay lại ví dụ về Bài toán đóng thùng *không có ràng buộc* và hàm mục tiêu là $f(x, y, z) = 40x^{-1}y^{-1}z^{-1} + 2xy + 2yz + 2zx$ là một posynomial. Vậy đây là một GP.

Code cho việc tìm *optimal point* của bài toán này bằng CVXOPT như sau:

```
from cvxopt import matrix, solvers
from math import log, exp# gp
from numpy import array
import numpy as np

K = [4]
F = matrix([[-1., 1., 1., 0.],
            [-1., 1., 0., 1.],
            [-1., 0., 1., 1.]])
g = matrix([log(40.), log(2.), log(2.), log(2.)])
solvers.options['show_progress'] = False
sol = solvers.gp(K, F, g)

print('Solution:')
print(np.exp(np.array(sol['x'])))

print('\nchecking sol^5')
print(np.exp(np.array(sol['x'])))**5)
```

```
Solution:
[[ 1.58489319]
 [ 1.58489319]
 [ 1.58489319]]

checking sol^5
[[ 9.9999998]
 [ 9.9999998]
 [ 9.9999998]]
```

Nghiệm thu được chính là $x = y = z = \sqrt[5]{10}$. Bạn đọc được khuyến khích đọc thêm chỉ dẫn của hàm `solvers.gp` để hiểu cách thiết lập bài toán.

2.7 Tóm tắt

- Các bài toán tối ưu xuất hiện rất nhiều trong thực tế, trong đó Tối Ưu Lồi đóng một vai trò quan trọng. Trong bài toán Tối Ưu Lồi, nếu tìm được cực trị thì cực trị đó chính là một điểm *optimal* của bài toán (nghiệm của bài toán).
- Có nhiều bài toán tối ưu không được viết dưới dạng *convex* nhưng có thể biến đổi về dạng *convex*, ví dụ như bài toán Geometric Programming.
- Linear Programming và Quadratic Programming đóng một vài trò quan trọng trong toán tối ưu, được sử dụng nhiều trong các thuật toán Machine Learning.
- Thư viện CVXOPT được dùng để tối ưu nhiều bài toán tối ưu lồi, rất dễ sử dụng và thời gian chạy tương đối nhanh.

2.8 Tài liệu tham khảo

- [1] Convex Optimization – Boyd and Vandenberghe, Cambridge University Press, 2004.
- [2] CVXOPT.

3

Duality

Trong bài viết này, chúng ta giả sử rằng các đạo hàm đều tồn tại.

Bài viết này chủ yếu được dịch lại từ Chương 5 của cuốn *Convex Optimization* trong tài liệu tham khảo.

3.1 Giới thiệu

Trong [Bài 16](#), chúng ta đã làm quen với các khái niệm về tập hợp lồi và hàm số lồi. Tiếp theo đó, trong [Bài 17](#), tôi cũng đã trình bày về các bài toán tối ưu lồi, cách nhận dạng và cách sử dụng thư viện để giải các bài toán lồi cơ bản. Trong bài này, chúng ta sẽ tiếp tục tiếp cận một cách sâu hơn: các điều kiện về nghiệm của các bài toán tối ưu, cả lồi và không lồi; bài toán đối ngẫu (dual problem) và điều kiện KKT.

Trước tiên, chúng ta lại bắt đầu bằng những kỹ thuật đơn giản cho các bài toán cơ bản. Kỹ thuật này có lẽ các bạn đã từng nghe đến: Phương pháp nhân tử Lagrange (method of [Lagrange multipliers](#)). Đây là một phương pháp giúp tìm các điểm cực trị của hàm mục tiêu trên feasible set của bài toán.

Nhắc lại rằng giá trị lớn nhất và nhỏ nhất (nếu có) của một hàm số $f_0(\mathbf{x})$ khả vi (và tập xác định là một [tập mở](#)) đạt được tại một trong các điểm cực trị của nó. Và điều kiện cần để một điểm là điểm cực trị là đạo hàm của hàm số tại điểm này $f'_0(x) = 0$. Chú ý rằng một điểm thoả mãn $f'_0(\mathbf{x}) = 0$ thì được gọi là *điểm dừng* hay *stationary point*. Điểm cực trị là một điểm dừng nhưng không phải điểm dừng nào cũng là điểm cực trị. Ví dụ hàm $f(x) = x^3$ có 0 là một điểm dừng nhưng không phải là điểm cực trị.

Với hàm nhiều biến, ta cũng có thể áp dụng quan sát này. Tức chúng ta cần đi tìm nghiệm của phương trình đạo hàm *theo mỗi biến* bằng 0. Tuy nhiên, đó là với các bài toán không ràng buộc (unconstrained optimization problems), với các bài toán có ràng buộc như chúng ta đã gặp trong Bài 17 thì sao?

Trước tiên chúng ta xét bài toán mà ràng buộc chỉ là một phương trình:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_1(\mathbf{x}) = 0 \end{aligned} \tag{3.1}$$

Bài toán này là bài toán tổng quát, không nhất thiết phải lồi. Tức hàm mục tiêu và hàm ràng buộc không nhất thiết phải lồi.

3.2 Phương pháp nhân tử Lagrange

Nếu chúng ta đưa được bài toán này về một bài toán không ràng buộc thì chúng ta có thể tìm được nghiệm bằng cách giải hệ phương trình đạo hàm theo từng thành phần bằng 0 (giả sử rằng việc giải hệ phương trình này là khả thi).

Điều này là động lực để nhà toán học [Lagrange](#) sử dụng hàm số: $\mathcal{L}(\mathbf{x}, \lambda) = f_0(\mathbf{x}) + \lambda f_1(\mathbf{x})$. Chú ý rằng, trong hàm số này, chúng ta có thêm một biến nữa là λ , biến này được gọi là nhân tử Lagrange (Lagrange multiplier). Hàm số $\mathcal{L}(\mathbf{x}, \lambda)$ được gọi là *hàm hỗ trợ* (*auxiliary function*), hay *the Lagrangian*. Người ta đã chứng minh được rằng, điểm *optimal value* của bài toán (3.1) thoả mãn điều kiện $\nabla_{\mathbf{x}, \lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$ (tôi xin được bỏ qua chứng minh của phần này). Điều này tương đương với:

$$\nabla_{\mathbf{x}} f_0(\mathbf{x}) + \lambda \nabla_{\mathbf{x}} f_1(\mathbf{x}) = 0 \tag{3.2}$$

$$f_1(\mathbf{x}) = 0 \tag{3.3}$$

Để ý rằng điều kiện thứ hai chính là $\nabla_{\lambda} \mathcal{L}(\mathbf{x}, \lambda) = 0$, và cũng chính là ràng buộc trong bài toán (3.1).

Việc giải hệ phương trình (3.2) - (3.3), trong nhiều trường hợp, đơn giản hơn việc trực tiếp đi tìm *optimal value* của bài toán (3.1).

Xét các ví dụ đơn giản sau đây.

3.2.1 Ví dụ

Ví dụ 1: Tìm giá trị lớn nhất và nhỏ nhất của hàm số $f_0(x, y) = x + y$ thoả mãn điều kiện $f_1(x, y) = x^2 + y^2 = 2$. Ta nhận thấy rằng đây không phải là một bài toán tối ưu lồi vì *feasible set* $x^2 + y^2 = 2$ không phải là một tập lồi (nó chỉ là một đường tròn).

Lời giải:

Lagrangian của bài toán này là: $\mathcal{L}(x, y, \lambda) = x + y + \lambda(x^2 + y^2 - 2)$. Các điểm cực trị của hàm số Lagrange phải thoả mãn điều kiện:

$$\nabla_{x,y,\lambda} \mathcal{L}(x, y, \lambda) = 0 \Leftrightarrow \begin{cases} 1 + 2\lambda x = 0 \\ 1 + 2\lambda y = 0 \\ x^2 + y^2 = 2 \end{cases} \quad (3.4)$$

Từ hai phương trình đầu của (3.4) ta suy ra $x = y = \frac{-1}{2\lambda}$. Thay vào phương trình ta sẽ có $\lambda^2 = \frac{1}{4} \Rightarrow \lambda = \pm\frac{1}{2}$. Vậy ta được 2 cặp nghiệm $(x, y) \in \{(1, 1), (-1, -1)\}$. Bằng cách thay các giá trị này vào hàm mục tiêu, ta tìm được giá trị nhỏ nhất và lớn nhất của hàm số cần tìm.

Ví dụ 2: Cross-entropy. Trong [Chương 10](#) và [Chương 13](#), chúng ta đã được biết đến hàm mất mát ở dạng [cross entropy](#). Chúng ta cũng đã biết rằng hàm cross entropy được dùng để đo sự giống nhau của hai phân phối xác suất với giá trị của hàm số này càng nhỏ thì hai xác suất càng gần nhau. Chúng ta cũng đã phát biểu rằng giá trị nhỏ nhất của hàm cross entropy đạt được khi từng gấp xác suất là giống nhau. Nay giờ, tôi xin phát biểu lại và chứng minh nhận định trên.

Cho một phân bố xác xuất $\mathbf{p} = [p_1, p_2, \dots, p_n]^T$ với $p_i \in [0, 1]$ và $\sum_{i=1}^n p_i = 1$. Với một phân bố xác suất bất kỳ $\mathbf{q} = [q_1, q_2, \dots, q_n]$ và giả sử rằng $q_i \neq 0, \forall i$, hàm số cross entropy được định nghĩa là:

$$f_0(\mathbf{q}) = - \sum_{i=1}^n p_i \log(q_i)$$

Hãy tìm \mathbf{q} để hàm cross entropy đạt giá trị nhỏ nhất.

Trong bài toán này, ta có ràng buộc là $\sum_{i=1}^n q_i = 1$. *Lagrangian* của bài toán là:

$$\mathcal{L}(q_1, q_2, \dots, q_n, \lambda) = - \sum_{i=1}^n p_i \log(q_i) + \lambda(\sum_{i=1}^n q_i - 1)$$

Ta cần giải hệ phương trình:

$$\nabla_{q_1, \dots, q_n, \lambda} \mathcal{L}(q_1, \dots, q_n, \lambda) = 0 \Leftrightarrow \begin{cases} -\frac{p_i}{q_i} + \lambda = 0, & i = 1, \dots, n \\ q_1 + q_2 + \dots + q_n = 1 \end{cases}$$

Từ phương trình thứ nhất ta có $p_i = \lambda q_i$. Vậy nên: $1 = \sum_{i=1}^n p_i = \lambda \sum_{i=1}^n q_i = \lambda \Rightarrow \lambda = 1 \Rightarrow q_i = p_i, \forall i$.

Qua đây, chúng ta đã hiểu rằng vì sao hàm số cross entropy được dùng để ép hai xác suất *gần nhau*.

3.3 Hàm đối ngẫu Lagrange (The Lagrange dual function)

3.3.1 Lagrangian

Với bài toán tối ưu tổng quát:

$$\begin{aligned} \mathbf{x}^* &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, \quad i = 1, 2, \dots, m \\ &h_j(\mathbf{x}) = 0, \quad j = 1, 2, \dots, p \end{aligned} \tag{3.5}$$

với miền xác định $\mathcal{D} = (\cap_{i=0}^m \text{dom } f_i) \cap (\cap_{j=1}^p \text{dom } h_j)$. Chú ý rằng, chúng ta đang không giả sử về tính chất lồi của hàm tối ưu hay các hàm ràng buộc ở đây. Giả sử duy nhất ở đây là $\mathcal{D} \neq \emptyset$ (tập rỗng).

Lagrangian cũng được xây dựng tương tự với mỗi nhân tử Lagrange cho một (bất) phương trình ràng buộc:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j h_j(\mathbf{x})$$

với $\boldsymbol{\lambda} = [\lambda_1, \lambda_2, \dots, \lambda_m]$; $\boldsymbol{\nu} = [\nu_1, \nu_2, \dots, \nu_p]$ là các vectors và được gọi là *dual variables* (*biến đối ngẫu*) hoặc *Lagrange multiplier vectors* (vector nhân tử Lagrange). Lúc này nếu biến chính $\mathbf{x} \in \mathbb{R}^n$ thì tổng số biến của hàm số này sẽ là $n + m + p$.

3.3.2 Hàm đối ngẫu Lagrange

Hàm đối ngẫu Lagrange của bài toán tối ưu (hoặc gọn là *hàm số đối ngẫu*) (3.5) là một hàm của các biến đối ngẫu, được định nghĩa là giá trị nhỏ nhất theo \mathbf{x} của *Lagrangian*:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) \tag{3.6}$$

$$= \inf_{\mathbf{x} \in \mathcal{D}} \left(f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j h_j(\mathbf{x}) \right) \tag{3.7}$$

Nếu *Lagrangian* không bị chặn dưới, hàm đối ngẫu tại $\boldsymbol{\lambda}, \boldsymbol{\nu}$ sẽ lấy giá trị $-\infty$.

Đặc biệt quan trọng:

- \inf được lấy trên miền $x \in \mathcal{D}$, tức miền xác định của bài toán (là giao của miền xác định của mọi hàm trong bài toán). Miền xác định này khác với *feasible set*. Thông thường, *feasible set* là tập con của miền xác định \mathcal{D} .

- Với mỗi \mathbf{x} , Lagrangian là một hàm *affine* của $(\boldsymbol{\lambda}, \boldsymbol{\nu})$, tức là một *hàm concave*. Vậy, *hàm đối ngẫu* chính là *pointwise infimum* của (có thể vô hạn) các hàm concave, tức là một hàm concave. Vậy **hàm đối ngẫu của một bài toán tối ưu bất kỳ là một hàm concave, bất kể bài toán ban đầu có phải là convex hay không**. Nhắc lại rằng *pointwise supremum* của các hàm *convex* là một hàm *convex*, và một hàm là *concave* nếu khi đổi dấu hàm đó, ta được một hàm *convex*.

3.3.3 Chặn dưới của giá trị tối ưu

Nếu p^* là *optimal value* (giá trị tối ưu) của bài toán (3.5), thì với các biến đối ngẫu $\lambda_i \geq 0, \forall i$ và ν bất kỳ, chúng ta sẽ có:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \leq p^* \quad (3.8)$$

Tính chất này có thể được chứng minh dễ dàng. Giả sử \mathbf{x}_0 là một điểm *feasible* bất kỳ của bài toán (3.5), tức thoả mãn các điều kiện ràng buộc $f_i(\mathbf{x}_0) \leq 0, \forall i = 1, \dots, m; h_j(\mathbf{x}_0) = 0, \forall j = 1, \dots, p$, ta sẽ có:

$$\sum_{i=1}^m \lambda_i f_i(\mathbf{x}_0) + \sum_{j=1}^p \nu_j h_j(\mathbf{x}_0) \leq 0 \Rightarrow \mathcal{L}(\mathbf{x}_0, \boldsymbol{\lambda}, \boldsymbol{\nu}) \leq f_0(\mathbf{x}_0)$$

Vì điều này đúng với mọi \mathbf{x}_0 *feasible*, ta sẽ có tính chất quan trọng sau đây:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x} \in \mathcal{D}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) \leq \mathcal{L}(\mathbf{x}_0, \boldsymbol{\lambda}, \boldsymbol{\nu}) \leq f_0(\mathbf{x}_0).$$

Khi $\mathbf{x}_0 = \mathbf{x}^*$, ta có bất đẳng thức (3.8).

3.3.4 Ví dụ

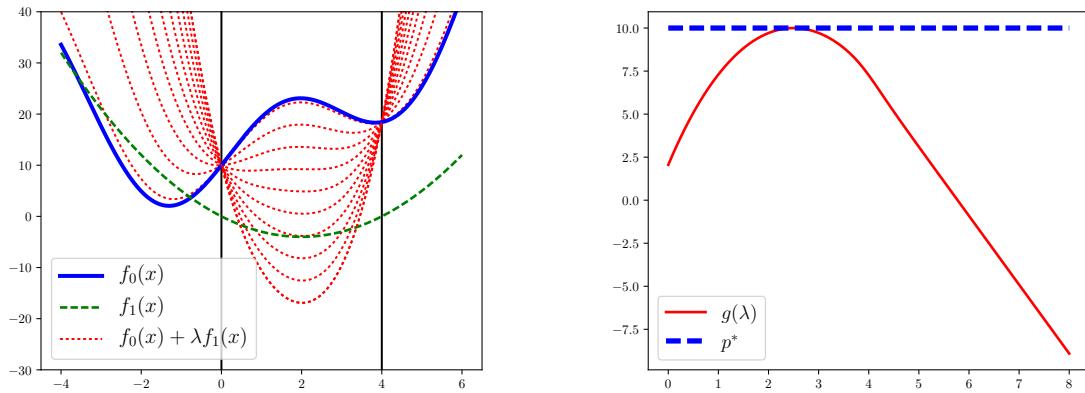
Ví dụ 1

Xét bài toán tối ưu sau:

$$\begin{aligned} x &= \arg \min_x x^2 + 10 \sin(x) + 10 \\ \text{subject to: } &(x - 2)^2 \leq 4 \end{aligned} \quad (3.9)$$

Chú ý: Với bài toán này, miền xác định $\mathcal{D} = \mathbb{R}$ nhưng *feasible set* là $0 \leq x \leq 4$.

Với hàm mục tiêu là đường đậm màu xanh lam trong Hình 3.1. Ràng buộc thực ra $0 \leq x \leq 4$, nhưng tôi viết ở dạng này để bài toán thêm phần thú vị. Hàm số ràng buộc $f_1(x) = (x-2)^2 - 4$ được cho bởi đường nét đứt màu xanh lục. Optimal value của bài toán này có thể được nhận



Hình 3.1: Ví dụ về dual function. Trái: Đường màu lam đậm thể hiện hàm mục tiêu. Đường nét đứt màu lục thể hiện hàm số ràng buộc. Các đường nét đứt màu đỏ thể hiện dual function ứng với các λ khác nhau. Phải: Đường nét đứt thể hiện giá trị tối ưu của bài toán. Đường màu đỏ thể hiện dual function. Với mọi λ , giá trị của hàm dual function nhỏ hơn hoặc bằng giá trị tối ưu của bài toán gốc.

ra là điểm trên đồ thị có hoành độ bằng 0. Chú ý rằng hàm mục tiêu ở đây không phải là hàm lồi nên bài toán tối ưu này cũng không phải là lồi, mặc dù hàm bất phương trình ràng buộc $f_1(x)$ là lồi.

Lagrangian của bài toán này có dạng:

$$\mathcal{L}(x, \lambda) = x^2 + 10 \sin(x) + 10 + \lambda((x - 2)^2 - 4)$$

Các đường dấu chấm màu đỏ trong Hình 1 là các đường ứng với các λ khác nhau. Vùng bị chặn giữa hai đường thẳng đứng màu đen thể hiện miền *feasible* của bài toán tối ưu.

Với mỗi λ , *dual function* được định nghĩa là:

$$g(\lambda) = \inf_x (x^2 + 10 \sin(x) + 10 + \lambda((x - 2)^2 - 4)), \quad \lambda \geq 0.$$

Từ hình 1 bên trái, ta có thể thấy ngay rằng với các λ khác nhau, $g(\lambda)$ hoặc tại điểm có hoành độ bằng 0, hoặc tại một điểm thấp hơn điểm tối ưu của bài toán. Đồ thị của hàm $g(\lambda)$ được cho bởi đường liền màu đỏ ở Hình 1 bên phải. Đường nét đứt màu lam thể hiện *optimal value* của bài toán tối ưu ban đầu. Ta có thể thấy ngay hai điều:

- Đường liền màu đỏ luôn nằm dưới (hoặc có đoạn trùng) với đường nét đứt màu lam.
- Hàm $g(\lambda)$ có dạng một hàm *concave*, tức nếu ta *lật* đồ thị này theo hướng trên-dưới thì ta sẽ có đồ thị của một hàm *convex*. (Mặc dù bài toán tối ưu gốc là không phải là một bài toán lồi.)

(Để vẽ được hình bên phải, tôi đã dùng *Gradient Descent* để tìm giá trị nhỏ nhất ứng với mỗi λ .)

Ví dụ 2

Xét một bài toán Linear Programming:

$$\begin{aligned} x &= \arg \min_{\mathbf{x}} \mathbf{c}^T \mathbf{x} \\ \text{s.t.: } &\mathbf{Ax} = \mathbf{b} \\ &\mathbf{x} \succeq 0 \end{aligned} \tag{3.10}$$

Hàm ràng buộc cuối cùng có thể được viết lại là: $f_i(\mathbf{x}) = -x_i, i = 1, \dots, n$. Lagrangian của bài toán này là:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) = \mathbf{c}^T \mathbf{x} - \sum_{i=1}^n \lambda_i x_i + \boldsymbol{\nu}^T (\mathbf{Ax} - \mathbf{b}) = -\mathbf{b}^T \boldsymbol{\nu} + (\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda})^T \mathbf{x}$$

(đừng quên điều kiện $\boldsymbol{\lambda} \succeq 0$.)

Dual function là:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \inf_{\mathbf{x}} \mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}, \boldsymbol{\nu}) \tag{3.11}$$

$$= -\mathbf{b}^T \boldsymbol{\nu} + \inf_{\mathbf{x}} (\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda})^T \mathbf{x} \tag{3.12}$$

Nhận thấy rằng một hàm tuyến tính $\mathbf{d}^T \mathbf{x}$ của \mathbf{x} bị chặn dưới khi vào chỉ khi $\mathbf{d} = 0$. Vì chỉ nếu một phần tử d_i của \mathbf{d} khác 0, ta chỉ cần chọn x_i rất lớn và ngược dấu với d_i , ta sẽ có một giá trị nhỏ tùy ý.

Nói cách khác, $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = -\infty$ trừ khi $\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0$. Tóm lại:

$$g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = \begin{cases} -\mathbf{b}^T \boldsymbol{\nu} & \text{if } \mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0 \\ -\infty & \text{otherwise} \end{cases} \tag{3.13}$$

Trường hợp thứ hai khi $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = -\infty$ các bạn sẽ gặp rất nhiều sau này. Trường hợp này không nhiều thú vị vì hiển nhiên $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \leq p^*$. Vì mục đích chính là đi tìm chặn dưới của p^* nên ta sẽ chỉ quan tâm tới các giá trị của $\boldsymbol{\lambda}$ và $\boldsymbol{\nu}$ sao cho $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ càng lớn càng tốt. Trong bài toán này, ta sẽ quan tâm tới các $\boldsymbol{\lambda}$ và $\boldsymbol{\nu}$ sao cho $\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0$.

3.4 Bài toán đối ngẫu Lagrange (The Lagrange dual problem)

Với mỗi cặp $(\boldsymbol{\lambda}, \boldsymbol{\nu})$, hàm đối ngẫu Lagrange cho chúng ta một chặn dưới cho *optimal value* p^* của bài toán gốc (3.5). Câu hỏi đặt ra là: với cặp giá trị nào của $(\boldsymbol{\lambda}, \boldsymbol{\nu})$, chúng ta sẽ có một chặn dưới tốt nhất của p^* ? Nói cách khác, ta đi cần giải bài toán:

$$\begin{aligned} \boldsymbol{\lambda}^*, \boldsymbol{\nu}^* &= \arg \max_{\boldsymbol{\lambda}, \boldsymbol{\nu}} g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \\ \text{subject to: } \boldsymbol{\lambda} &\succeq 0 \end{aligned} \quad (3.14)$$

Một điểm quan trọng: vì $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ là *concave* và hàm ràng buộc $f_i(\boldsymbol{\lambda}) = -\lambda_i$ là các hàm *convex*. Vậy bài toán (3.14) chính là một bài toán lồi. Vì vậy trong nhiều trường hợp, lời giải có thể dễ tìm hơn là bài toán gốc. Chú ý rằng, bài toán đối ngẫu (3.14) là lồi bất kể bài toán gốc (3.5) có là lồi hay không.

Bài toán này được gọi là *Lagrange dual problem* (bài toán đối ngẫu Largange) ứng với bài toán (3.5). Bài toán (3.5) còn có tên gọi khác là *primal problem* (bài toán gốc). Ngoài ra, có một khái niệm nữa, gọi là *dual feasible* tức là *feasible set* của bài toán đối ngẫu, bao gồm điều kiện $\boldsymbol{\lambda} \succeq 0$ và điều kiện ẩn $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) > -\infty$ (vì ta đang đi tìm giá trị lớn nhất của hàm số nên $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) = -\infty$ rõ ràng là không thú vị).

Nghiệm của bài toán (3.14), ký hiệu là $\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ được gọi là *dual optimal* hoặc *optimal Lagrange multipliers*.

Chú ý rằng điều kiện ẩn $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) > -\infty$, trong nhiều trường hợp, cũng có thể được viết cụ thể. Quay lại với ví dụ phía trên, điều kiện ẩn có thể được viết thành $\mathbf{c} + \mathbf{A}^T \boldsymbol{\nu} - \boldsymbol{\lambda} = 0$. Đây là một hàm affine. Vì vậy, khi có thêm ràng buộc này, ta vẫn được một bài toán lồi.

3.4.1 Weak duality

Ký hiệu giá trị tối ưu của bài toán đối ngẫu (3.14) là d^* . Theo (3.14), ta đã biết rằng:

$$d^* \leq p^*$$

ngay cả khi bài toán gốc không phải là lồi.

Tính chất đơn giản này được gọi là *weak duality*. Tuy đơn giản nhưng nó cực kỳ quan trọng.

Từ đây ta quan sát thấy hai điều:

- Nếu bài toán gốc không bị chặn dưới, tức $p^* = -\infty$, ta phải có $d^* = -\infty$, tức là bài toán đối ngẫu Lagrange là *infeasible* (tức không có giá trị nào thoả mãn ràng buộc).
- Nếu bài toán đối ngẫu là không bị chặn trên, tức $d^* = +\infty$, chúng ta phải có $p^* = +\infty$, tức bài toán gốc là *infeasible*.

Giá trị $p^* - d^*$ được gọi là *optimal duality gap* (dịch thô là *khoảng cách đối ngẫu tối ưu*). Khoảng cách này luôn luôn là một số không âm.

Đôi khi có những bài toán (lồi hoặc không) rất khó giải, nhưng ít nhất nếu ta có thể tìm được d^* , ta có thể biết được chặn dưới của bài toán gốc. Việc tìm d^* thường có thể thực hiện được vì bài toán đối ngẫu luôn luôn là lồi.

3.4.2 Strong duality và Slater's constraint qualification

Nếu đẳng thức $p^* = d^*$ thoả mãn, *the optimal duality gap* bằng không, ta nói rằng *strong duality* xảy ra. Lúc này, việc giải bài toán đối ngẫu đã giúp ta tìm được *chính xác* giá trị tối ưu của bài toán gốc.

Thật không may, *strong duality* không thường xuyên xảy ra trong các bài toán tối ưu. Tuy nhiên, nếu bài toán gốc là lồi, tức có dạng:

$$\begin{aligned} x &= \arg \min_{\mathbf{x}} f_0(\mathbf{x}) \\ \text{subject to: } &f_i(\mathbf{x}) \leq 0, i = 1, 2, \dots, m \\ &\mathbf{Ax} = \mathbf{b} \end{aligned} \tag{3.15}$$

trong đó f_0, f_1, \dots, f_m là các hàm lồi, chúng ta *thường* (không luôn luôn) có *strong duality*. Có rất nhiều nghiên cứu thiết lập các điều kiện, ngoài tính chất lồi, để *strong duality* xảy ra. Những điều kiện đó thường có tên là *constraint qualifications*.

Một trong các *constraint qualification* đơn giản nhất là *Slater's condition*.

Định nghĩa: Một điểm *feasible* của bài toán (3.15) được gọi là *strictly feasible* nếu:

$$f_i(\mathbf{x}) < 0, \quad i = 1, 2, \dots, m, \quad \mathbf{Ax} = \mathbf{b}$$

Định lý Slater: Nếu tồn tại một điểm *strictly feasible* (và bài toán gốc là lồi), thì *strong duality* xảy ra.

Điều kiện khá đơn giản sẽ giúp ích cho nhiều bài toán tối ưu sau này.

Chú ý:

- *Strong duality* không thường xuyên xảy ra. Với các bài toán lồi, việc này xảy ra thường xuyên hơn. Tồn tại những bài toán lồi mà *strong duality* không xảy ra.
- Có những bài toán không lồi nhưng *strong duality* vẫn xảy ra. Ví dụ như bài toán trong Hình 1 phía trên.

3.5 Optimality conditions

3.5.1 Complementary slackness

Giả sử rằng *strong duality* xảy ra. Gọi \mathbf{x}^* là một điểm *optimal* của bài toán gốc và (λ^*, ν^*) là cặp điểm *optimal* của bài toán đối ngẫu. Ta có:

$$f_0(\mathbf{x}^*) = g(\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*) \quad (3.16)$$

$$= \inf_{\mathbf{x}} \left(f_0(\mathbf{x}) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}) \right) \quad (3.17)$$

$$\leq f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* h_j(\mathbf{x}^*) \quad (3.18)$$

$$\leq f_0(\mathbf{x}^*) \quad (3.19)$$

- Dòng đầu là do chính là *strong duality*.
- Dòng hai là do định nghĩa của hàm đối ngẫu.
- Dòng ba là hiển nhiên vì infimum của một hàm nhỏ hơn giá trị của hàm đó tại bất kỳ một điểm nào khác.
- Dòng bốn là vì các ràng buộc $f_i(\mathbf{x}^*) \leq 0, \lambda_i \geq 0, i = 1, 2, \dots, m$ và $h_j(\mathbf{x}^*) = 0$.

Từ đây có thể thấy rằng dấu đẳng thức ở dòng ba và dòng bốn phải đồng thời xảy ra. Và ta lại có thêm hai quan sát thú vị nữa:

- \mathbf{x}^* chính là một điểm *optimal* của $g(\lambda^*, \nu^*)$.
- Thú vị hơn:

$$\sum_{i=1}^m \lambda_i^* f_i(\mathbf{x}^*) = 0$$

Vì mỗi phần tử trong tổng trên là không dương do $\lambda_i^* \geq 0, f_i \leq 0$, ta kết luận rằng:

$$\lambda_i^* f_i(\mathbf{x}^*) = 0, i = 1, 2, \dots, m$$

Điều kiện cuối cùng này được gọi là *complementary slackness*. Từ đây có thể suy ra:

$$\lambda_i^* > 0 \Rightarrow f_i(\mathbf{x}^*) = 0 \quad (3.20)$$

$$f_i(\mathbf{x}^*) < 0 \Rightarrow \lambda_i^* = 0 \quad (3.21)$$

Tức ta luôn có một trong hai giá trị này bằng 0.

3.5.2 KKT optimality conditions

Chúng ta vẫn giả sử rằng các hàm đang xét có đạo hàm và bài toán tối ưu không nhất thiết là lồi.

KKT conditions cho bài toán *không* lồi

Giả sử rằng *strong duality* xảy ra. Gọi \mathbf{x}^* và $(\boldsymbol{\lambda}^*, \boldsymbol{\nu}^*)$ là *điều kiện primal và dual optimal points*. Vì \mathbf{x}^* tối ưu hàm khả vi $\mathcal{L}(\mathbf{x}, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*)$, ta có đạo hàm của Lagrangian tại \mathbf{x}^* phải bằng 0.

Điều kiện Karush-Kuhn-Tucker (KKT) nói rằng $\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ phải thoả mãn điều kiện:

$$f_i(\mathbf{x}^*) \leq 0, i = 1, 2, \dots, m \quad (3.22)$$

$$h_j(\mathbf{x}^*) = 0, j = 1, 2, \dots, p \quad (3.23)$$

$$\lambda_i^* \geq 0, i = 1, 2, \dots, m \quad (3.24)$$

$$\lambda_i^* f_i(\mathbf{x}^*) = 0, i = 1, 2, \dots, m \quad (3.25)$$

$$\nabla f_0(\mathbf{x}^*) + \sum_{i=1}^m \lambda_i^* \nabla f_i(\mathbf{x}^*) + \sum_{j=1}^p \nu_j^* \nabla h_j(\mathbf{x}^*) = 0 \quad (3.26)$$

Đây là *điều kiện cần* để $\mathbf{x}^*, \boldsymbol{\lambda}^*, \boldsymbol{\nu}^*$ là nghiệm của hai bài toán.

KKT conditions cho bài toán lồi

Với các bài toán lồi và *strong duality* xảy ra, các điều kiện KKT phía trên cũng là *điều kiện đủ*. Vậy với các bài toán lồi với hàm mục tiêu và hàm ràng buộc là khả vi, bất kỳ điểm nào thoả mãn các điều kiện KKT đều là *primal và dual optimal* của bài toán gốc và bài toán đối ngẫu.

Từ đây ta có thể thấy rằng: Với một bài toán lồi và điều kiện Slater thoả mãn (suy ra *strong duality*) thì các điều kiện KKT là điều kiện cần và đủ của nghiệm.

Các điều kiện KKT rất quan trọng trong tối ưu. Trong một vài trường hợp đặc biệt (chúng ta sẽ thấy trong bài Support Vector Machine sắp tới), việc giải hệ (bất) phương trình các điều kiện KKT là khả thi. Rất nhiều các thuật toán tối ưu được xây dựng dựa trên việc giải hệ điều kiện KKT.

Ví dụ: *Equality constrained convex quadratic minimization.* Xét bài toán:

$$\begin{aligned} \mathbf{x} &= \arg \min_{\mathbf{x}} \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + r \\ \text{subject to: } \mathbf{A} \mathbf{x} &= \mathbf{b} \end{aligned} \quad (3.27)$$

trong đó $\mathbf{P} \in \mathbb{S}_+^n$ (tập các ma trận đối xứng nửa xác định dương).

Lagrangian:

$$\mathcal{L}(\mathbf{x}, \boldsymbol{\nu}) = \frac{1}{2} \mathbf{x}^T \mathbf{P} \mathbf{x} + \mathbf{q}^T \mathbf{x} + r + \boldsymbol{\nu}^T (\mathbf{A} \mathbf{x} - \mathbf{b})$$

Điều kiện KKT cho bài toán này là:

$$\mathbf{A} \mathbf{x}^* = \mathbf{b} \quad (3.28)$$

$$\mathbf{P} \mathbf{x}^* + \mathbf{q} + \mathbf{A}^T \boldsymbol{\nu}^* = 0 \quad (3.29)$$

Phương trình thứ hai chính là phương trình đạo hàm của Lagrangian tại \mathbf{x}^* bằng 0.

Hệ phương trình này có thể được viết lại đơn giản là:

$$\begin{bmatrix} \mathbf{P} & \mathbf{A}^T \\ \mathbf{A} & \mathbf{0} \end{bmatrix} \begin{bmatrix} \mathbf{x}^* \\ \boldsymbol{\nu}^* \end{bmatrix} = \begin{bmatrix} -\mathbf{q} \\ \mathbf{b} \end{bmatrix}$$

đây là một phương trình tuyến tính đơn giản!

3.6 Tóm tắt

Giả sử rằng các hàm số đều khả vi:

- Các bài toán tối ưu với chỉ ràng buộc là đẳng thức có thể được giải quyết bằng phương pháp nhân tử Lagrange. Ta cũng có định nghĩa về Lagrangian. Điều kiện cần để một điểm là nghiệm của bài toán tối ưu là nó phải làm cho đạo hàm của Lagrangian bằng 0.
- Với các bài toán tối ưu có thêm ràng buộc là bất đẳng thức (không nhất thiết là lồi), chúng ta có Lagrangian tổng quát và các biến Lagrange λ, ν . Với các giá trị (λ, ν) cố định, ta có định nghĩa về **hàm đối ngẫu Lagrange** (Lagrange dual function) $g(\lambda, \nu)$ được xác định là infimum của Lagrangian khi \mathbf{x} thay đổi trên miền xác định của bài toán.
- Miền xác định và tập các điểm *feasible* thường khác nhau. *Feasible set* là tập con của tập xác định.
- Với mọi $(\boldsymbol{\lambda}, \boldsymbol{\nu})$, $g(\boldsymbol{\lambda}, \boldsymbol{\nu}) \leq p^*$.
- Hàm số $g(\boldsymbol{\lambda}, \boldsymbol{\nu})$ là **lồi** bất kể bài toán tối ưu có là lồi hay không. Hàm số này được gọi là *dual Lagrange function* hay *hàm đối ngẫu Lagrange*.
- Bài toán đi tìm giá trị lớn nhất của hàm đối ngẫu Lagrange với điều kiện $\boldsymbol{\lambda} \succeq 0$ được gọi là bài toán *đối ngẫu* (*dual problem*). Bài toán này là **lồi** bất kể bài toán gốc có lồi hay không.

- Gọi giá trị tối ưu của bài toán đối ngẫu là d^* thì ta có: $d^* \leq p^*$. Đây được gọi là *weak duality*.
- *Strong duality* xảy ra khi $d^* = p^*$. Thường thì *strong duality* không xảy ra, nhưng với các bài toán lồi thì *strong duality* thường (không luôn luôn) xảy ra.
- Nếu bài toán là lồi và điều kiện Slater thoả mãn, thì *strong duality* xảy ra.
- Nếu bài toán lồi và có *strong duality* thì nghiệm của bài toán thoả mãn các điều kiện KKT (điều kiện cần và đủ).
- Rất nhiều các bài toán tối ưu được giải quyết thông qua KKT conditions.

3.7 Kết luận

Trong ba bài 16, 17, 18, tôi đã giới thiệu sơ lược về tập lồi, hàm lồi, bài toán lồi, và các điều kiện tối ưu được xây dựng thông qua *duality*. Ý định ban đầu của tôi là tránh phần này vì khá nhiều toán, tuy nhiên trong quá trình chuẩn bị cho bài Support Vector Machine, tôi nhận thấy rằng cần phải giải thích về Lagrangian - kỹ thuật được sử dụng rất nhiều trong Tối ưu. Thêm nữa, để giải thích về Lagrangian, tôi cần nói về các bài toán lồi. Chính vì vậy tôi thấy có trách nhiệm *phải* viết về ba bài này.

Trong loạt bài tiếp theo, chúng ta sẽ lại quay lại với các thuật toán Machine Learning với rất nhiều ví dụ, hình vẽ và code mẫu. Nếu bạn nào có cảm thấy hơi đuối sau ba bài tối ưu này thì cũng đừng lo, mọi chuyện rồi sẽ ổn cả thôi.

3.8 Tài liệu tham khảo

[1] Convex Optimization – Boyd and Vandenberghe, Cambridge University Press, 2004.

[2] Lagrange Multipliers - Wikipedia.

Part II

Support Vector Machines

4

Support Vector Machine

Nếu không muốn đi sâu vào phân toán, bạn có thể bỏ qua Mục 4.3.

4.1 Giới thiệu

Trước khi đi vào phần ý tưởng chính của Support Vector Machine, tôi xin một lần nữa nhắc lại kiến thức về hình học giải tích mà chúng ta đã quá quen khi ôn thi đại học.

4.1.1 Khoảng cách từ một điểm tới một siêu mặt phẳng

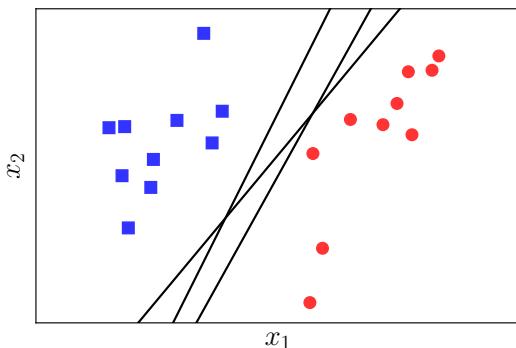
Trong không gian 2 chiều, ta biết rằng khoảng cách từ một điểm có toạ độ (x_0, y_0) tới đường thẳng có phương trình $w_1x + w_2y + b = 0$ được xác định bởi:

$$\frac{\|w_1x_0 + w_2y_0 + b\|}{\sqrt{w_1^2 + w_2^2}}$$

Trong không gian ba chiều, khoảng cách từ một điểm có toạ độ (x_0, y_0, z_0) tới một mặt phẳng có phương trình $w_1x + w_2y + w_3z + b = 0$ được xác định bởi:

$$\frac{\|w_1x_0 + w_2y_0 + w_3z_0 + b\|}{\sqrt{w_1^2 + w_2^2 + w_3^2}}$$

Hơn nữa, nếu ta bỏ dấu trị tuyệt đối ở tử số, chúng ta có thể xác định được điểm đó nằm về phía nào của đường thẳng hay mặt phẳng đang xét. Những điểm làm cho biểu thức trong dấu giá trị tuyệt đối mang dấu dương nằm về cùng 1 phía (tôi tạm gọi đây là *phía dương* của đường thẳng), những điểm làm cho biểu thức trong dấu giá trị tuyệt đối mang dấu âm nằm về phía còn lại (tôi gọi là *phía âm*). Những điểm nằm trên đường thẳng/mặt phẳng sẽ làm cho tử số có giá trị bằng 0, tức khoảng cách bằng 0.



Hình 4.1: Hai lớp dữ liệu đỏ và xanh là *phân biệt tuyến tính*. Có vô số các đường thẳng có thể phân tách chính xác hai lớp dữ liệu này.

Việc này có thể được tổng quát lên không gian nhiều chiều: Khoảng cách từ một điểm (vector) có toạ độ \mathbf{x}_0 tới *siêu mặt phẳng* (*hyperplane*) có phương trình $\mathbf{w}^T \mathbf{x} + b = 0$ được xác định bởi:

$$\frac{\|\mathbf{w}^T \mathbf{x}_0 + b\|}{\|\mathbf{w}\|_2}$$

Với $\|\mathbf{w}\|_2 = \sqrt{\sum_{i=1}^d w_i^2}$ với d là số chiều của không gian.

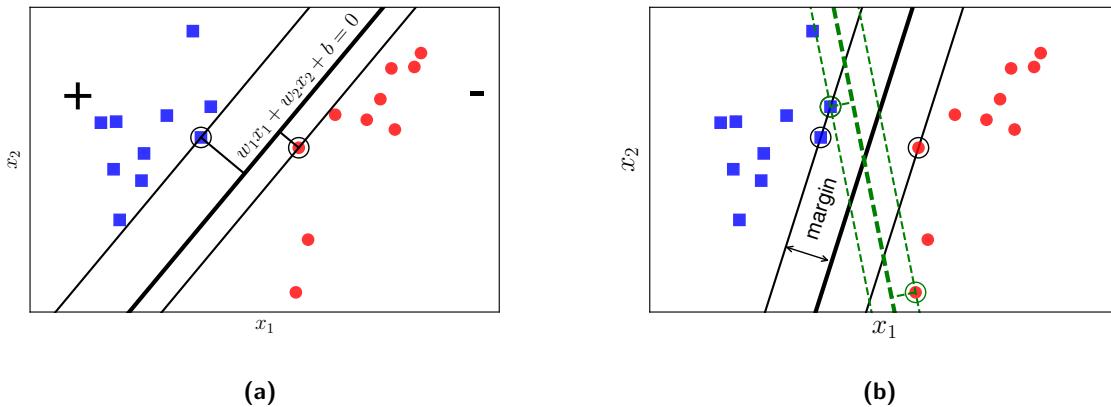
4.1.2 Nhắc lại bài toán phân chia hai classes

Chúng ta cùng quay lại với bài toán trong [Perceptron Learning Algorithm \(PLA\)](#). Giả sử rằng có hai class khác nhau được mô tả bởi các điểm trong không gian nhiều chiều, hai classes này *linearly separable*, tức tồn tại một siêu phẳng phân chia chính xác hai classes đó. Hãy tìm một siêu mặt phẳng phân chia hai classes đó, tức tất cả các điểm thuộc một class nằm về cùng một phía của siêu mặt phẳng đó và ngược phía với toàn bộ các điểm thuộc class còn lại. Chúng ta đã biết rằng, thuật toán PLA có thể làm được việc này nhưng nó có thể cho chúng ta vô số nghiệm như [Hình 4.1](#).

Câu hỏi đặt ra là: trong vô số các mặt phân chia đó, đâu là mặt phân chia tốt nhất *theo một tiêu chuẩn nào đó*? Trong ba đường thẳng minh họa trong Hình 1 phía trên, có hai đường thẳng khá *lệch* về phía class hình tròn đỏ. Điều này có thể khiến cho lớp màu đỏ *không vui vì lạnh thở* xem ra bị *lấn chiếm quá*. Liệu có cách nào để tìm được đường phân chia mà cả hai classes đều cảm thấy *công bằng* và *hạnh phúc* nhất hay không?

Chúng ta cần tìm một tiêu chuẩn để đo sự *hạnh phúc* của mỗi class.

Hãy xem [Hình 4.2](#). Nếu ta định nghĩa *mức độ hạnh phúc* của một class tỉ lệ thuận với khoảng cách gần nhất từ một điểm của class đó tới đường/mặt phân chia, thì ở [Hình 4.2a](#), class tròn đỏ sẽ *không được hạnh phúc* cho lắm vì đường phân chia gần nó hơn class vuông xanh rất nhiều. Chúng ta cần một đường phân chia sao cho khoảng cách từ điểm gần nhất của mỗi



Hình 4.2: Biên giữa hai lớp là bằng nhau và lớn nhất có thể.

class (các điểm được khoanh tròn) tới đường phân chia là như nhau, như thế thì mới *công bằng*. Khoảng cách như nhau này được gọi là *margin* (*lề*).

Đã có *công bằng* rồi, chúng ta cần *văn minh* nữa. *Công bằng* mà cả hai đều *kém hạnh phúc* như nhau thì chưa phải là *văn minh* cho lắm.

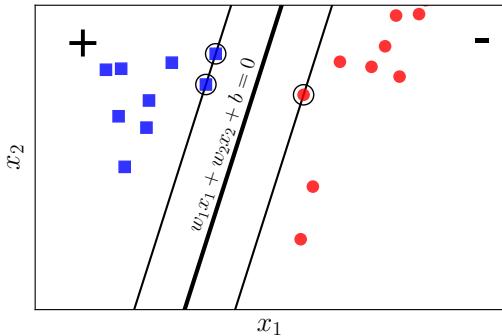
Chúng ta xét tiếp Hình 4.2b khi khoảng cách từ đường phân chia tới các điểm gần nhất của mỗi class là như nhau. Xét hai cách phân chia bởi đường nét liền màu đen và đường nét đứt màu lục, đường nào sẽ làm cho cả hai class *hạnh phúc hơn*? Rõ ràng đó phải là đường nét liền màu đen vì nó tạo ra một *margin* rộng hơn.

Việc *margin* rộng hơn sẽ mang lại hiệu ứng phân lớp tốt hơn vì *sự phân chia giữa hai classes là rạch ròi hơn*. Việc này, sau này các bạn sẽ thấy, là một điểm khá quan trọng giúp *Support Vector Machine* mang lại kết quả phân loại tốt hơn so với *Neural Network* với 1 layer, tức Perceptron Learning Algorithm.

Bài toán tối ưu trong *Support Vector Machine* (SVM) chính là bài toán đi tìm đường phân chia sao cho *margin* là lớn nhất. Đây cũng là lý do vì sao SVM còn được gọi là *Maximum Margin Classifier*. Nguồn gốc của tên gọi Support Vector Machine sẽ sớm được làm sáng tỏ.

4.2 Xây dựng bài toán tối ưu cho SVM

Giả sử rằng các cặp dữ liệu của *training set* là $(\mathbf{x}_1, y_1), (\mathbf{x}_2, y_2), \dots, (\mathbf{x}_N, y_N)$ với vector $\mathbf{x}_i \in \mathbb{R}^d$ thể hiện *đầu vào* của một điểm dữ liệu và y_i là *nhãn* của điểm dữ liệu đó. d là số chiều của dữ liệu và N là số điểm dữ liệu. Giả sử rằng *nhãn* của mỗi điểm dữ liệu được xác định bởi $y_i = 1$ (class 1) hoặc $y_i = -1$ (class 2) giống như trong PLA.

Hình 4.3: Phân tích bài toán SVM.

Để giúp các bạn dễ hình dung, chúng ta cùng xét trường hợp trong không gian hai chiều dưới đây. *Không gian hai chiều để các bạn dễ hình dung, các phép toán hoàn toàn có thể được tổng quát lên không gian nhiều chiều.*

Giả sử rằng các điểm vuông xanh thuộc class 1, các điểm tròn đỏ thuộc class -1 và mặt $\mathbf{w}^T \mathbf{x} + b = w_1 x_1 + w_2 x_2 + b = 0$ là mặt phân chia giữa hai classes (Hình 4.3). Hơn nữa, class 1 nằm về *phía dương*, class -1 nằm về *phía âm* của mặt phân chia. Nếu ngược lại, ta chỉ cần đổi dấu của \mathbf{w} và b . Chú ý rằng chúng ta cần đi tìm các hệ số \mathbf{w} và b .

Ta quan sát thấy một điểm quan trọng sau đây: với cặp dữ liệu (\mathbf{x}_n, y_n) bất kỳ, khoảng cách từ điểm đó tới mặt phân chia là:

$$\frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2}$$

Điều này có thể dễ nhận thấy vì theo giả sử ở trên, y_n luôn cùng dấu với *phía* của \mathbf{x}_n . Từ đó suy ra y_n cùng dấu với $(\mathbf{w}^T \mathbf{x}_n + b)$, và tử số luôn là 1 số không âm.

Với mặt phân chia như trên, *margin* được tính là khoảng cách gần nhất từ 1 điểm tới mặt đó (bất kể điểm nào trong hai classes):

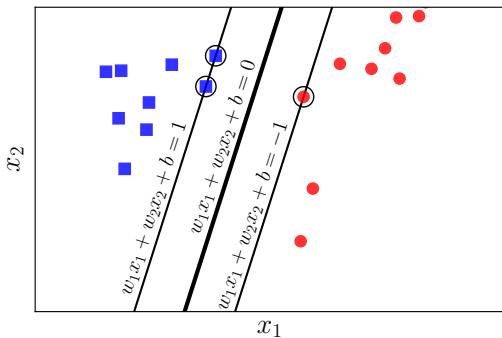
$$\text{margin} = \min_n \frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2}$$

Bài toán tối ưu trong SVM chính là bài toán tìm \mathbf{w} và b sao cho *margin* này đạt giá trị lớn nhất:

$$(\mathbf{w}, b) = \arg \max_{\mathbf{w}, b} \left\{ \min_n \frac{y_n(\mathbf{w}^T \mathbf{x}_n + b)}{\|\mathbf{w}\|_2} \right\} = \arg \max_{\mathbf{w}, b} \left\{ \frac{1}{\|\mathbf{w}\|_2} \min_n y_n(\mathbf{w}^T \mathbf{x}_n + b) \right\} \quad (4.1)$$

Việc giải trực tiếp bài toán này sẽ rất phức tạp, nhưng các bạn sẽ thấy có cách để đưa nó về bài toán đơn giản hơn.

Nhận xét quan trọng nhất là nếu ta thay vector hệ số \mathbf{w} bởi $k\mathbf{w}$ và b bởi kb trong đó k là một hằng số dương thì mặt phân chia không thay đổi, tức khoảng cách từ từng điểm đến

Hình 4.4: Phân tích bài toán SVM.

mặt phân chia không đổi, tức *margin* không đổi. Dựa trên tính chất này, ta có thể giả sử:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$$

với những điểm nằm gần mặt phân chia nhất như Hình 4.4 dưới đây:

Như vậy, với mọi n , ta có:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$$

Vậy bài toán tối ưu (4.1) có thể đưa về bài toán tối ưu có ràng buộc sau đây:

$$\begin{aligned} (\mathbf{w}, b) &= \arg \max_{\mathbf{w}, b} \frac{1}{\|\mathbf{w}\|_2} \\ \text{subject to: } &y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1, \forall n = 1, 2, \dots, N \end{aligned} \quad (4.2)$$

Bằng 1 biến đổi đơn giản, ta có thể đưa bài toán này về bài toán dưới đây:

$$\begin{aligned} (\mathbf{w}, b) &= \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \\ \text{subject to: } &1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \forall n = 1, 2, \dots, N \end{aligned} \quad (4.3)$$

Ở đây, chúng ta đã lấy nghịch đảo hàm mục tiêu, bình phương nó để được một hàm khả vi, và nhân với $\frac{1}{2}$ để biểu thức đạo hàm đẹp hơn.

Quan sát quan trọng: Trong bài toán (4.3), **hàm mục tiêu là một norm, nên là một hàm lồi**. Các hàm bất đẳng thức ràng buộc là các hàm tuyến tính theo \mathbf{w} và b , nên chúng cũng là các hàm lồi. Vậy bài toán tối ưu (4.3) có hàm mục tiêu là lồi, và các hàm ràng buộc cũng là lồi, nên nó là một bài toán lồi. Hơn nữa, nó là một **Quadratic Programming**. Thậm chí, hàm mục tiêu là *strictly convex* vì $\|\mathbf{w}\|_2^2 = \mathbf{w}^T \mathbf{I} \mathbf{w}$ và \mathbf{I} là ma trận đơn vị - là một ma trận xác định dương. Từ đây có thể suy ra nghiệm cho SVM là *duy nhất*.

Đến đây thì bài toán này có thể giải được bằng các công cụ hỗ trợ tìm nghiệm cho Quadratic Programming, ví dụ **CVXOPT**.

Tuy nhiên, việc giải bài toán này trở nên phức tạp khi số chiều d của không gian dữ liệu và số điểm dữ liệu N tăng lên cao.

Người ta thường giải **bài toán đối ngẫu** của bài toán này. Thứ nhất, bài toán đối ngẫu có những tính chất thú vị hơn khiến nó được giải hiệu quả hơn. Thứ hai, trong quá trình xây dựng bài toán đối ngẫu, người ta thấy rằng SVM có thể được áp dụng cho những bài toán mà dữ liệu không *linearly separable*, tức các đường phân chia không phải là một mặt phẳng mà có thể là các mặt có hình thù phức tạp hơn.

Xác định class cho một điểm dữ liệu mới: Sau khi tìm được mặt phân cách $\mathbf{w}^T \mathbf{x} + b = 0$, class của bất kỳ một điểm nào sẽ được xác định đơn giản bằng cách:

$$\text{class}(\mathbf{x}) = \text{sgn}(\mathbf{w}^T \mathbf{x} + b)$$

Trong đó hàm sgn là hàm xác định dấu, nhận giá trị 1 nếu đối số là không âm và -1 nếu ngược lại.

4.3 Bài toán đối ngẫu cho SVM

Nhắc lại rằng bài toán tối ưu (4.3) là một bài toán lồi. Chúng ta biết rằng: nếu một **bài toán lồi thoả mãn tiêu chuẩn Slater** thì *strong duality* thoả mãn. Và nếu *strong duality* thoả mãn thì nghiệm của bài toán chính là nghiệm của hệ **điều kiện KKT**.

4.3.1 Kiểm tra tiêu chuẩn Slater

Bước tiếp theo, chúng ta sẽ chứng minh bài toán tối ưu (4.3) thoả mãn điều kiện Slater. Điều kiện Slater nói rằng, nếu tồn tại \mathbf{w}, b thoả mãn:

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) < 0, \quad \forall n = 1, 2, \dots, N$$

thì *strong duality* thoả mãn.

Việc kiểm tra này tương đối đơn giản. Vì ta biết rằng luôn luôn có một (siêu) mặt phẳng phân chia hai classes nếu hai class đó là *linearly separable*, tức bài toán có nghiệm, nên *feasible set* của bài toán tối ưu (4.3) phải khác rỗng. Tức luôn luôn tồn tại cặp (\mathbf{w}_0, b_0) sao cho:

$$1 - y_n(\mathbf{w}_0^T \mathbf{x}_n + b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (4.4)$$

$$\Leftrightarrow 2 - y_n(2\mathbf{w}_0^T \mathbf{x}_n + 2b_0) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (4.5)$$

Vậy chỉ cần chọn $\mathbf{w}_1 = 2\mathbf{w}_0$ và $b_1 = 2b_0$, ta sẽ có:

$$1 - y_n(\mathbf{w}_1^T \mathbf{x}_n + b_1) \leq -1 < 0, \quad \forall n = 1, 2, \dots, N$$

Từ đó suy ra điều kiện Slater thoả mãn.

4.3.2 Lagrangian của bài toán SVM

Lagrangian của bài toán (4.3) là:

$$\mathcal{L}(\mathbf{w}, b, \lambda) = \frac{1}{2} \|\mathbf{w}\|_2^2 + \sum_{n=1}^N \lambda_n (1 - y_n (\mathbf{w}^T \mathbf{x}_n + b)) \quad (4.6)$$

với $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_N]^T$ và $\lambda_n \geq 0, \forall n = 1, 2, \dots, N$.

4.3.3 Hàm đối ngẫu Lagrange

Hàm đối ngẫu Lagrange được định nghĩa là:

$$g(\boldsymbol{\lambda}) = \min_{\mathbf{w}, b} \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda})$$

với $\lambda \succeq 0$.

Việc tìm giá trị nhỏ nhất của hàm này theo \mathbf{w} và b có thể được thực hiện bằng cách giải hệ phương trình đạo hàm của $\mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda})$ theo \mathbf{w} và b bằng 0:

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda})}{\partial \mathbf{w}} = \mathbf{w} - \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n = 0 \Rightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (4.7)$$

$$\frac{\partial \mathcal{L}(\mathbf{w}, b, \boldsymbol{\lambda})}{\partial b} = \sum_{n=1}^N \lambda_n y_n = 0 \quad (4.8)$$

Thay (4.7) và (4.8) vào (4.6) ta thu được $g(\boldsymbol{\lambda})$ (*phần này tôi rút gọn, coi như một bài tập nhỏ cho bạn nào muốn hiểu sâu*):

$$g(\boldsymbol{\lambda}) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \quad (4.9)$$

Hàm $g(\boldsymbol{\lambda})$ trong (4.9) là **hàm số quan trọng nhất trong SVM**, các bạn sẽ thấy rõ hơn ở bài Kernel Support Vector .

Xét ma trận:

$$\mathbf{V} = [y_1 \mathbf{x}_1, y_2 \mathbf{x}_2, \dots, y_N \mathbf{x}_N]$$

và vector $\mathbf{1} = [1, 1, \dots, 1]^T$, ta có thể viết lại $g(\lambda)$ dưới dạng:

$$g(\boldsymbol{\lambda}) = -\frac{1}{2} \boldsymbol{\lambda}^T \mathbf{V}^T \mathbf{V} \boldsymbol{\lambda} + \mathbf{1}^T \boldsymbol{\lambda}. \quad (4.10)$$

(Nếu khó tin, bạn có thể viết ra để quen dần với các biểu thức đại số tuyến tính.)

Đặt $\mathbf{K} = \mathbf{V}^T \mathbf{V}$, ta có một quan sát quan trọng: \mathbf{K} là một ma trận nửa xác định dương. Thật vậy, với mọi vector $\boldsymbol{\lambda}$, ta có:

$$\boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} = \boldsymbol{\lambda}^T \mathbf{V}^T \mathbf{V} \boldsymbol{\lambda} = \|\mathbf{V} \boldsymbol{\lambda}\|_2^2 \geq 0.$$

(Đây chính là định nghĩa của ma trận nửa xác định dương.)

Vậy $g(\boldsymbol{\lambda}) = -\frac{1}{2} \boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} + \mathbf{1}^T \boldsymbol{\lambda}$ là một hàm *concave*.

4.3.4 Bài toán đối ngẫu Lagrange

Từ đó, kết hợp hàm đối ngẫu Lagrange và các điều kiện ràng buộc của $\boldsymbol{\lambda}$, ta sẽ thu được bài toán đối ngẫu Lagrange:

$$\begin{aligned} \boldsymbol{\lambda} &= \arg \max_{\boldsymbol{\lambda}} g(\boldsymbol{\lambda}) \\ \text{subject to: } \boldsymbol{\lambda} &\succeq 0 \\ \sum_{n=1}^N \lambda_n y_n &= 0 \end{aligned} \quad (4.11)$$

Ràng buộc thứ hai được lấy từ (4.8).

Đây là một bài toán lồi vì ta đang đi tìm giá trị lớn nhất của một hàm mục tiêu là *concave* trên một *polyhedron*.

Bài toán này cũng được là một Quadratic Programming và cũng có thể được giải bằng các thư viện như CVXOPT.

Trong bài toán đối ngẫu này, số tham số (parameters) phải tìm là N , là chiều của $\boldsymbol{\lambda}$, tức số điểm dữ liệu. Trong khi đó, với bài toán gốc (4.3), số tham số phải tìm là $d+1$, là tổng số chiều của \mathbf{w} và b , tức số chiều của mỗi điểm dữ liệu cộng với 1. Trong rất nhiều trường hợp, số điểm dữ liệu có được trong *training set* lớn hơn số chiều dữ liệu rất nhiều. Nếu giải trực tiếp bằng các công cụ giải Quadratic Programming, có thể bài toán đối ngẫu còn phức tạp hơn (tốn thời gian hơn) so với bài toán gốc. Tuy nhiên, điều hấp dẫn của bài toán đối ngẫu này đến từ phần *Kernel Support Vector Machine (Kernel SVM)*, tức cho các bài toán mà dữ liệu không phải là *linearly separable* hoặc *gần linearly separable*. Phần *Kernel SVM* sẽ được tôi trình bày sau 1 hoặc 2 bài nữa. Ngoài ra, dựa vào tính chất đặc biệt của hệ điều kiện KKT mà SVM có thể được giải bằng nhiều phương pháp hiệu quả hơn.

4.3.5 Điều kiện KKT

Quay trở lại bài toán, vì đây là một bài toán lồi và *strong duality* thoả mãn, nghiệm của bài toán sẽ thoả mãn hệ [điều kiện KKT](#) sau đây với biến số là \mathbf{w}, b và λ :

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N \quad (4.12)$$

$$\lambda_n \geq 0, \quad \forall n = 1, 2, \dots, N \quad (4.13)$$

$$\lambda_n(1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0, \quad \forall n = 1, 2, \dots, N \quad (4.14)$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (4.15)$$

$$\sum_{n=1}^N \lambda_n y_n = 0 \quad (4.16)$$

Trong những điều kiện trên, điều kiện (4.14) là thú vị nhất. Từ đó ta có thể suy ra ngay, với n bất kỳ, hoặc $\lambda_n = 0$ hoặc $1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) = 0$. Trường hợp thứ hai chính là:

$$\mathbf{w}^T \mathbf{x}_n + b = y_n \quad (4.17)$$

với chú ý rằng $y_n^2 = 1, \forall n$.

Những điểm thoả mãn (4.17) chính là những điểm nằm gần mặt phân chia nhất, là những điểm được khoanh tròn trong Hình 4 phía trên. Hai đường thẳng $\mathbf{w}^T \mathbf{x}_n + b = \pm 1$ tựa lên các điểm thoả mãn (4.17). Vậy nên những điểm (vectors) thoả mãn (4.17) còn được gọi là các *Support Vectors*. Và từ đó, cái tên *Support Vector Machine* ra đời.

Một quan sát khác, số lượng những điểm thoả mãn (4.17) thường chiếm số lượng rất nhỏ trong số N điểm. Chỉ cần dựa trên những *support vectors* này, chúng ta hoàn toàn có thể xác định được mặt phân cách cần tìm. Nhìn theo một cách khác, hầu hết các λ_n bằng 0. Vậy là mặc dù vector $\lambda \in \mathbb{R}^N$ có số chiều có thể rất lớn, số lượng các phần tử khác 0 của nó rất ít. Nói cách khác, vector λ là một *sparse vector*. Support Vector Machine vì vậy còn được xếp vào *Sparse Models*. Các *Sparse Models* thường có cách giải hiệu quả (nhanh) hơn các mô hình tương tự với nghiệm là *dense* (hầu hết khác 0). Đây chính là lý do thứ hai của việc bài toán đối ngẫu SVM được quan tâm nhiều hơn là bài toán gốc.

Tiếp tục phân tích, với những bài toán có số điểm dữ liệu N nhỏ, ta có thể giải hệ điều kiện KKT phía trên bằng cách xét các trường hợp $\lambda_n = 0$ hoặc $\lambda_n \neq 0$. Tổng số trường hợp phải xét là 2^N . Với $N > 50$ (thường là như thế), đây là một con số rất lớn, giải bằng cách này sẽ không khả thi. Tôi sẽ không đi sâu tiếp vào việc giải hệ KKT như thế nào, trong phần tiếp theo chúng ta sẽ giải bài toán tối ưu (4.11) bằng CVXOPT và bằng thư viện `sklearn`.

Sau khi tìm được λ từ bài toán (4.11), ta có thể suy ra được \mathbf{w} dựa vào (4.15) và b dựa vào (4.14) và (4.16). Rõ ràng ta chỉ cần quan tâm tới $\lambda_n \neq 0$.

Gọi tập hợp $\mathcal{S} = \{n : \lambda_n \neq 0\}$ và $N_{\mathcal{S}}$ là số phần tử của tập \mathcal{S} . Với mỗi $n \in \mathcal{S}$, ta có:

$$1 = y_n(\mathbf{w}^T \mathbf{x}_n + b) \Leftrightarrow b + \mathbf{w}^T \mathbf{x}_n = y_n$$

Mặc dù từ chỉ một cặp (\mathbf{x}_n, y_n) , ta có thể suy ra ngay được b nếu đã biết \mathbf{w} , một phiên bản khác để tính b thường được sử dụng và được cho là *ổn định hơn trong tính toán (numerically more stable)* là:

$$b = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (4.18)$$

tức trung bình cộng của mọi cách tính b .

Trước đó, \mathbf{w} đã được tính bằng:

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (4.19)$$

theo (4.15).

Quan sát quan trọng: Để xác định một điểm \mathbf{x} mới thuộc vào class nào, ta cần xác định dấu của biểu thức:

$$\mathbf{w}^T \mathbf{x} + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{S}}} \sum_{n \in \mathcal{S}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right)$$

Biểu thức này phụ thuộc vào cách tính tích vô hướng giữa các cặp vector \mathbf{x} và từng $\mathbf{x}_n \in \mathcal{S}$. Nhận xét quan trọng này sẽ giúp ích cho chúng ta trong bài Kernal SVM.

4.4 Lập trình tìm nghiệm cho SVM

Trong mục này, tôi sẽ trình bày hai cách tính nghiệm cho SVM. Cách thứ nhất dựa theo bài toán (4.11) và các công thức (4.18) và (4.19). Cách thứ hai sử dụng trực tiếp thư viện `sklearn`. Cách thứ nhất chỉ là để chứng minh nãy giờ tôi không *viết nhảm*, bằng cách minh họa kết quả tìm được và so sánh với nghiệm tìm được bằng cách thứ hai.

4.4.1 Tìm nghiệm theo công thức

Trước tiên chúng ta gọi các *modules* cần dùng và tạo dữ liệu giả (dữ liệu này chính là dữ liệu tôi dùng trong các hình phía trên nên chúng ta biết chắc rằng hai classes là *linearly separable*):

```

from __future__ import print_function
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
np.random.seed(22)

means = [[2, 2], [4, 2]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N) # class 1
X1 = np.random.multivariate_normal(means[1], cov, N) # class -1
X = np.concatenate((X0.T, X1.T), axis = 1) # all data
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1) # labels

```

Tiếp theo, chúng ta giải bài toán (4.11) bằng CVXOPT:

```

from cvxopt import matrix, solvers
# build K
V = np.concatenate((X0.T, -X1.T), axis = 1)
K = matrix(V.T.dot(V)) # see definition of V, K near eq (8)

p = matrix(-np.ones((2*N, 1))) # all-one vector
# build A, b, G, h
G = matrix(-np.eye(2*N)) # for all lambda_n >= 0
h = matrix(np.zeros((2*N, 1)))
A = matrix(y) # the equality constrain is actually y^T lambda = 0
b = matrix(np.zeros((1, 1)))
solvers.options['show_progress'] = False
sol = solvers.qp(K, p, G, h, A, b)

l = np.array(sol['x'])
print('lambda = ')
print(l.T)

```

Kết quả:

```

lambda =
[[ 8.54018321e-01   2.89132533e-10   1.37095535e+00   6.36030818e-10
  4.04317408e-10   8.82390106e-10   6.35001881e-10   5.49567576e-10
  8.33359230e-10   1.20982928e-10   6.86678649e-10   1.25039745e-10
  2.22497367e+00   4.05417905e-09   1.26763684e-10   1.99008949e-10
  2.13742578e-10   1.51537487e-10   3.75329509e-10   3.56161975e-10]]

```

Ta nhận thấy rằng hầu hết các giá trị của **lambda** đều rất nhỏ, tới 10^{-9} hoặc 10^{-10} . Đây chính là các giá trị bằng 0 nhưng vì sai số tính toán nên nó khác 0 một chút. Chỉ có 3 giá trị khác 0, ta dự đoán là sẽ có 3 điểm là *support vectors*.

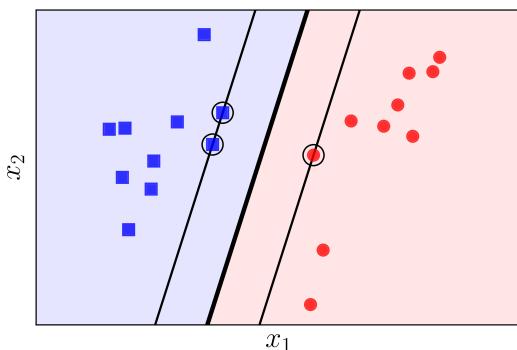
Ta đi tìm *support set* \mathcal{S} rồi tìm nghiệm của bài toán:

```

epsilon = 1e-6 # just a small number, greater than 1e-9
S = np.where(l > epsilon)[0]

VS = V[:, S]
XS = X[:, S]
yS = y[:, S]
lS = l[S]
# calculate w and b

```



Hình 4.5: Minh họa nghiệm tìm được bởi SVM. Tất cả các điểm nằm trong vùng có nền màu lam nhạt sẽ được phân vào cùng lớp với các điểm màu xanh. Điều tương tự xảy ra với các điểm nằm trên nền màu đỏ nhạt.

```
w = VS.dot(lS)
b = np.mean(yS.T - w.T.dot(XS))

print('w = ', w.T)
print('b = ', b)

w =  [-2.00984381  0.64068336]
b =  4.66856063387
```

Kết quả tìm được được minh họa trong Hình 4.5.

Đường màu đen đậm ở giữa chính là mặt phân cách tìm được bằng SVM. Từ đây có thể thấy *nhiều khả năng là các tính toán của ta là chính xác*. Để kiểm tra xem các tính toán phía trên có chính xác không, ta cần tìm nghiệm bằng các công cụ có sẵn, ví dụ như **sklearn**.

Source code cho phần này có thể được tìm thấy [ở đây](#).

4.4.2 Tìm nghiệm theo thư viện

Chúng ta sẽ sử dụng hàm **sklearn.svm.SVC** ở đây. Các bài toán thực tế thường sử dụng thư viện **libsvm** được viết trên ngôn ngữ C, có API cho Python và Matlab.

Nếu dùng thư viện thì sẽ như sau:

```
from sklearn.svm import SVC

y1 = y.reshape((2*N,))
X1 = X.T # each sample is one row
clf = SVC(kernel = 'linear', C = 1e5) # just a big number

clf.fit(X1, y1)

w = clf.coef_
b = clf.intercept_
print('w = ', w)
print('b = ', b)
```

Kết quả:

```
w = [[-2.00971102  0.64194082]]
b = [ 4.66595309]
```

Kết quả này khá giống với kết quả chúng ta tìm được ở phần trên. Có rất nhiều tuỳ chọn cho SVM, các bạn sẽ dần thấy trong các bài sau.

4.5 Tóm tắt và thảo luận

- Với bài toán binary classification mà 2 classes là *linearly separable*, có vô số các siêu mặt phẳng giúp phân biệt hai classes, tức mặt phân cách. Với mỗi mặt phân cách, ta có một *classifier*. Khoảng cách gần nhất từ 1 điểm dữ liệu tới mặt phân cách ấy được gọi là *margin* của classifier đó.
- Support Vector Machine là bài toán đi tìm mặt phân cách sao cho *margin* tìm được là lớn nhất, đồng nghĩa với việc các điểm dữ liệu *an toàn nhất* so với mặt phân cách.
- Bài toán tối ưu trong SVM là một bài toán lồi với hàm mục tiêu là *strictly convex*, nghiệm của bài toán này là duy nhất. Hơn nữa, bài toán tối ưu đó là một Quadratic Programming (QP).
- Mặc dù có thể trực tiếp giải SVM qua bài toán tối ưu gốc này, thông thường người ta thường giải bài toán đối ngẫu. Bài toán đối ngẫu cũng là một QP nhưng nghiệm là *sparse* nên có những phương pháp giải hiệu quả hơn.
- Với các bài toán mà dữ liệu *gần linearly separable* hoặc *nonlinear separable*, có những cải tiến khác của SVM để thích nghi với dữ liệu đó. Mời bạn đón đọc bài tiếp theo.
- [Source code](#).

4.6 Tài liệu tham khảo

- [1] Bishop, Christopher M. "Pattern recognition and Machine Learning.", Springer (2006). ([book](#))
- [2] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern classification. John Wiley & Sons, 2012.
- [3] [sklearn.svm.SVC](#)
- [4] [LIBSVM – A Library for Support Vector Machines](#)

Soft Margin Support Vector Machines

5.1 Giới thiệu

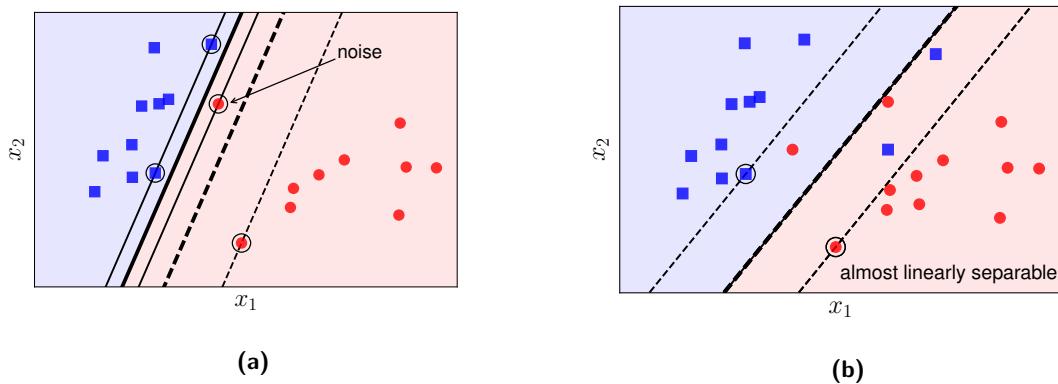
Giống như Perceptron Learning Algorithm (PLA), Support Vector Machine (SVM) *thuần* chỉ làm việc khi dữ liệu của 2 classes là *linearly separable*. Một cách tự nhiên, chúng ta cũng mong muốn rằng SVM có thể làm việc với dữ liệu *gần linearly separable* giống như Logistic Regression đã làm được.

Bạn được khuyến khích đọc bài [Support Vector Machine](#)) trước khi đọc bài này.

Xét hai ví dụ trong Hình 5.1. Có hai trường hợp dễ nhận thấy SVM làm việc không hiệu quả hoặc thậm chí không làm việc:

1. Trường hợp 1: Dữ liệu vẫn *linearly separable* như Hình 1a) nhưng có một điểm *nhiều* của lớp tròn đỏ ở gần so với lớp vuông xanh. Trong trường hợp này, nếu ta sử dụng SVM *thuần* thì sẽ tạo ra một *margin* rất nhỏ. Ngoài ra, đường phân lớp nằm quá gần lớp vuông xanh và xa lớp tròn đỏ. Trong khi đó, nếu ta *hy sinh* điểm nhiễu này thì ta được một *margin* tốt hơn rất nhiều được mô tả bởi các đường nét đứt. SVM *thuần* vì vậy còn được coi là *nhạy cảm với nhiễu* (*sensitive to noise*).
2. Trường hợp 2: Dữ liệu không *linearly separable* nhưng *gần linearly separable* như Hình 1b). Trong trường hợp này, nếu ta sử dụng SVM *thuần* thì rõ ràng bài toán tối ưu là *infeasible*, tức *feasible set* là một tập rỗng, vì vậy bài toán tối ưu SVM trở nên vô nghiệm. Tuy nhiên, nếu ta lại *chịu hy sinh một chút* những điểm ở gần biên giữa hai classes, ta vẫn có thể tạo được một đường phân chia khá tốt như đường nét đứt đậm. Các *đường support* đường nét đứt mảnh vẫn giúp tạo được một margin lớn cho bộ phân lớp này. Với mỗi điểm nằm lân sang phía bên kia của các đường support (hay *đường margin*, hoặc *đường biên*) tương ứng, ta gọi điểm đó rơi vào *vùng không an toàn*. Chú ý rằng vùng an toàn của hai classes là khác nhau, giao nhau ở phần nằm giữa hai đường support.

Trong cả hai trường hợp trên, *margin* tạo bởi đường phân chia và đường nét đứt mảnh còn được gọi là *soft margin* (*biên mềm*). Cũng theo cách gọi này, SVM *thuần* còn được gọi là *Hard Margin SVM* (*SVM biên cứng*).



Hình 5.1: Hai trường hợp khi Support Vector Machine làm việc không hiệu quả: khi (a) hai lớp vẫn linearly separable nhưng một điểm thuộc class này quá gần class kia, điểm này có thể là nhiễu; (b) dữ liệu hai lớp không linearly separable, mặc dù chúng gần linearly separable.

Trong bài này, chúng ta sẽ tiếp tục tìm hiểu một biến thể của *Hard Margin SVM* có tên gọi là *Soft Margin SVM*.

Bài toán tối ưu cho *Soft Margin SVM* có hai cách tiếp cận khác nhau, cả hai đều mang lại những kết quả thú vị và có thể phát triển tiếp thành các thuật toán SVM phức tạp và hiệu quả hơn.

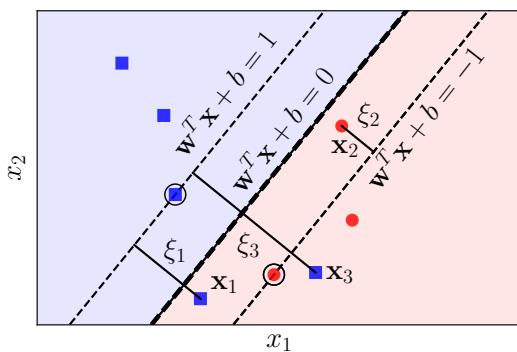
Cách giải quyết thứ nhất là giải một bài toán tối ưu có ràng buộc bằng cách giải bài toán đối ngẫu giống như *Hard Margin SVM*; cách giải dựa vào bài toán đối ngẫu này là cơ sở cho phương pháp *Kernel SVM* cho dữ liệu thực sự không *linearly separable* mà tôi sẽ đề cập trong bài tiếp theo. Hướng giải quyết này sẽ được trình bày trong Mục 3 bên dưới.

Cách giải quyết thứ hai là đưa về một bài toán tối ưu *không* ràng buộc. Bài toán này có thể giải bằng các phương pháp Gradient Descent. Nhờ đó, cách giải quyết này có thể được áp dụng cho các bài toán *large scale*. Ngoài ra, trong cách giải này, chúng ta sẽ làm quen với một hàm mất mát mới có tên là *hinge loss*. Hàm mất mát này có thể mở rộng ra cho bài toán *multi-class classification* mà tôi sẽ đề cập sau 2 bài nữa (*Multi-class SVM*). Cách phát triển từ *Soft Margin SVM* thành *Multi-class SVM* có thể so sánh với cách phát triển từ Logistic Regression thành *Softmax Regression*. Hướng giải quyết này sẽ được trình bày trong Mục 4 bên dưới.

Trước hết, chúng ta cùng đi phân tích bài toán.

5.2 Phân tích toán học

Như đã đề cập phía trên, để có một *margin* lớn hơn trong *Soft Margin SVM*, chúng ta cần *hy sinh* một vài điểm dữ liệu bằng cách chấp nhận cho chúng rơi vào vùng *không an toàn*.

**Hình 5.2:** Giới thiệu các biến slack ξ_n .

Với những điểm nằm trong *vùng an toàn*, $\xi_n = 0$. Những điểm nằm trong *vùng không an toàn* nhưng vẫn đúng phía so với đường phân chia tương ứng với các $0 < \xi_n < 1$, ví dụ \mathbf{x}_2 . Những điểm nằm ngược phía với class của chúng so với đường boundary ứng với các $\xi_n > 1$, ví dụ như \mathbf{x}_1 và \mathbf{x}_3

Tất nhiên, chúng ta phải hạn chế *sự hy sinh* này, nếu không, chúng ta có thể tạo ra một biên cực lớn bằng cách *hy sinh* hầu hết các điểm. Vậy hàm mục tiêu nên là một sự kết hợp để tối đa *margin* và tối thiểu *sự hy sinh*.

Giống như với *Hard Margin SVM*, việc tối đa *margin* có thể đưa về việc tối thiểu $\|\mathbf{w}\|_2^2$. Để xác định *sự hy sinh*, chúng ta cùng theo dõi Hình 2 dưới đây:

Với mỗi điểm \mathbf{x}_n trong tập toàn bộ dữ liệu huấn luyện, ta giới thiệu thêm một biến đo *sự hy sinh* ξ_n tương ứng. Biến này còn được gọi là *slack variable*. Với những điểm \mathbf{x}_n nằm trong *vùng an toàn*, $\xi_n = 0$. Với mỗi điểm nằm trong *vùng không an toàn* như \mathbf{x}_1 , \mathbf{x}_2 hay \mathbf{x}_3 , ta có $\xi_i > 0$.

Nhận thấy rằng nếu $y_i = \pm 1$ là nhãn của \mathbf{x}_i trong *vùng không an toàn* thì:

$$\xi_i = |\mathbf{w}^T \mathbf{x}_i + b - y_i| \quad (5.1)$$

(Bạn có nhận ra không?)

Nhắc lại bài toán tối ưu cho *Hard Margin SVM*:

$$\begin{aligned} (\mathbf{w}, b) &= \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 \\ \text{subject to: } y_n(\mathbf{w}^T \mathbf{x}_n + b) &\geq 1, \quad \forall n = 1, 2, \dots, N \end{aligned} \quad (5.2)$$

Với *Soft Margin SVM*, hàm mục tiêu sẽ có thêm một số hạng nữa giúp tối thiểu *sự hy sinh*. Từ đó ta có hàm mục tiêu:

$$\frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \quad (5.3)$$

trong đó C là một hằng số dương và $\xi = [\xi_1, \xi_2, \dots, \xi_N]$.

Hằng số C được dùng để điều chỉnh tầm quan trọng giữa *margin* và *sự hy sinh*. Hằng số này được xác định từ trước bởi người lập trình hoặc có thể được xác định bởi [cross-validation](#).

Điều kiện ràng buộc sẽ thay đổi một chút. Với mỗi cặp dữ liệu (\mathbf{x}_n, y_n) , thay vì ràng buộc $y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1$, chúng ta sẽ có ràng buộc mềm:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) \geq 1 - \xi_n \Leftrightarrow 1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \quad \forall n = 1, 2, \dots, N$$

Và ràng buộc phụ $\xi_n \geq 0, \forall n = 1, 2, \dots, N$.

Tóm lại, ta sẽ có bài toán tối ưu ở dạng chuẩn cho *Soft-margin SVM*:

$$\begin{aligned} (\mathbf{w}, b, \xi) &= \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \\ \text{subject to: } &1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \forall n = 1, 2, \dots, N \\ &-\xi_n \leq 0, \forall n = 1, 2, \dots, N \end{aligned} \tag{5.4}$$

Nhận xét:

- Nếu C nhỏ, việc *sự hy sinh* cao hay thấp không gây ảnh hưởng nhiều tới giá trị của hàm mục tiêu, thuật toán sẽ điều chỉnh sao cho $\|\mathbf{w}\|_2^2$ là nhỏ nhất, tức *margin* là lớn nhất, điều này sẽ dẫn tới $\sum_{n=1}^N \xi_n$ sẽ lớn theo. Ngược lại, nếu C quá lớn, để hàm mục tiêu đạt giá trị nhỏ nhất, thuật toán sẽ tập trung vào làm giảm $\sum_{n=1}^N \xi_n$. Trong trường hợp C rất rất lớn và hai classes là *linearly separable*, ta sẽ thu được $\sum_{n=1}^N \xi_n = 0$. Chú ý rằng giá trị này không thể nhỏ hơn 0. Điều này đồng nghĩa với việc không có điểm nào phải *hy sinh*, tức ta thu được nghiệm cho *Hard Margin SVM*. Nói cách khác, *Hard Margin SVM* chính là một trường hợp đặc biệt của *Soft Margin SVM*.
- Bài toán tối ưu (5.4) có thêm sự xuất hiện của *slack variables* ξ_n . Những $\xi_n = 0$ tương ứng với những điểm dữ liệu nằm trong *vùng an toàn*. Những $0 < \xi_n \leq 1$ tương ứng với những điểm nằm trong *vùng không an toàn* nhưng vẫn được phân loại đúng, tức vẫn nằm về đúng phía so với đường phân chia. Những $\xi_n > 1$ tương ứng với các điểm bị phân loại sai.
- Hàm mục tiêu trong bài toán tối ưu (5.4) là một hàm lồi vì nó là tổng của hai hàm lồi: hàm norm và hàm tuyến tính. Các hàm ràng buộc cũng là các hàm tuyến tính theo (\mathbf{w}, b, ξ) . Vì vậy bài toán tối ưu (5.4) là một bài toán lồi, hơn nữa nó có thể biểu diễn dưới dạng một *Quadratic Programming (QP)*.

Dưới đây, chúng ta sẽ cùng giải quyết bài toán tối ưu (5.4) bằng hai cách khác nhau.

5.3 Bài toán đối ngẫu Lagrange

Chú ý rằng bài toán này có thể giải trực tiếp bằng các toolbox hỗ trợ QP, nhưng giống như với *Hard Margin SVM*, chúng ta sẽ quan tâm hơn tới bài toán đối ngẫu.

Trước hết, ta cần kiểm tra **tiêu chuẩn Slater** cho bài toán tối ưu lồi (5.4). Nếu tiêu chuẩn này được thoả mãn, *strong duality* sẽ thoả mãn, và ta sẽ có nghiệm của bài toán tối ưu (5.4) là nghiệm của **hệ điều kiện KKT**. (Những kiến thức được đề cập trong mục này có thể được tìm thấy trong Bài 18).

5.3.1 Kiểm tra tiêu chuẩn Slater

Rõ ràng là với mọi $n = 1, 2, \dots, N$ và mọi (\mathbf{w}, b) , ta luôn có thể tìm được các số **dương** $\xi_n, n = 1, 2, \dots, N$ đủ lớn sao cho:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) + \xi_n > 1, \quad \forall n = 1, 2, \dots, N$$

Vậy nên bài toán này thoả mãn tiêu chuẩn Slater.

5.3.2 Lagrangian của bài toán Soft-margin SVM

Lagrangian cho bài toán (5.4) là:

$$\mathcal{L}(\mathbf{w}, b, \xi, \lambda, \mu) = \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n + \sum_{n=1}^N \lambda_n (1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) - \sum_{n=1}^N \mu_n \xi_n \quad (5.5)$$

với $\lambda = [\lambda_1, \lambda_2, \dots, \lambda_N]^T \succeq 0$ và $\mu = [\mu_1, \mu_2, \dots, \mu_N]^T \succeq 0$ là các biến đổi ngẫu Lagrange (vector nhân tử Lagrange).

5.3.3 Bài toán đổi ngẫu

Hàm số đổi ngẫu của bài toán tối ưu (5.4) là:

$$g(\lambda, \mu) = \min_{\mathbf{w}, b, \xi} \mathcal{L}(\mathbf{w}, b, \xi, \lambda, \mu)$$

Với mỗi cặp (λ, μ) , chúng ta sẽ quan tâm tới (\mathbf{w}, b, ξ) thoả mãn điều kiện đạo hàm của Lagrangian bằng 0:

$$\frac{\partial \mathcal{L}}{\partial \mathbf{w}} = 0 \Leftrightarrow \mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (5.6)$$

$$\frac{\partial \mathcal{L}}{\partial b} = 0 \Leftrightarrow \sum_{n=1}^N \lambda_n y_n = 0 \quad (5.7)$$

$$\frac{\partial \mathcal{L}}{\partial \xi_n} = 0 \Leftrightarrow \lambda_n = C - \mu_n \quad (5.8)$$

Từ (5.8) ta thấy rằng ta chỉ quan tâm tối đa những cặp (λ, μ) sao cho $\lambda_n = C - \mu_n$. Từ đây ta cũng suy ra $0 \leq \lambda_n, \mu_n \leq C, n = 1, 2, \dots, N$. Thay các biểu thức này vào Lagrangian ta sẽ thu được hàm đối ngẫu:

$$g(\lambda, \mu) = \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \quad (5.9)$$

Chú ý rằng hàm này không phụ thuộc vào μ nhưng ta cần lưu ý ràng buộc (5.8), ràng buộc này và điều kiện không âm của λ có thể được viết gọn lại thành $0 \leq \lambda_n \leq C$, và ta đã giảm được biến μ . Lúc này, bài toán đối ngẫu được xác định bởi:

$$\begin{aligned} \lambda &= \arg \max_{\lambda} g(\lambda) \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0 \end{aligned} \quad (5.10)$$

$$0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \quad (5.11)$$

Bài toán này gần giống với bài toán đối ngẫu của Hard Margin SVM, chỉ khác là ta có chặn trên cho mỗi λ_n . Khi C rất lớn, ta có thể coi hai bài toán là như nhau. Ràng buộc (5.11) còn được gọi là *box constraint* vì không gian các điểm λ thỏa mãn ràng buộc này giống như một hình hộp chữ nhật trong không gian nhiều chiều.

Bài toán này cũng hoàn toàn giải được bằng các công cụ giải QP thông thường, ví dụ CVXOPT như tôi đã thực hiện trong bài Hard Margin SVM.

Sau khi tìm được λ của bài toán đối ngẫu, ta vẫn phải quay lại tìm nghiệm (\mathbf{w}, b, ξ) của bài toán gốc. Để làm điều này, chúng ta cùng xem xét hệ điều kiện KKT.

5.3.4 Hệ điều kiện KKT

Hệ điều kiện KKT của bài toán tối ưu Soft Margin SVM là, với mọi $n = 1, 2, \dots, N$:

$$1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0 \quad (5.12)$$

$$-\xi_n \leq 0 \quad (5.13)$$

$$\lambda_n \geq 0 \quad (5.14)$$

$$\mu_n \geq 0 \quad (5.15)$$

$$\lambda_n(1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x}_n + b)) = 0 \quad (5.16)$$

$$\mu_n \xi_n = 0 \quad (5.17)$$

$$\mathbf{w} = \sum_{n=1}^N \lambda_n y_n \mathbf{x}_n \quad (5.6)$$

$$\sum_{n=1}^N \lambda_n y_n = 0 \quad (5.7)$$

$$\lambda_n = C - \mu_n \quad (5.8)$$

(Để cho dễ hình dung, các điều kiện (5.6) (5.7) (5.8) đã được nhắc lại trong hệ này.)

Ta có một vài quan sát như sau:

- Nếu $\lambda_n = 0$ thì từ (5.8) ta suy ra $\mu_n = C \neq 0$. Kết hợp với (5.17) ta suy ra $\xi_n = 0$. Nói cách khác, không có sự hy sinh nào xảy ra ở \mathbf{x}_n , tức \mathbf{x}_n nằm trong vùng an toàn.
- Nếu $\lambda_n > 0$, từ (5.16) ta có:

$$y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1 - \xi_n$$

- Nếu $0 < \lambda_n < C$, từ (5.8) ta suy ra $\mu_n \neq 0$ và từ (5.17) ta lại có $\xi_n = 0$. Nói cách khác, $y_n(\mathbf{w}^T \mathbf{x}_n + b) = 1$, hay những điểm \mathbf{x}_n nằm chính xác trên margin.
- Với những điểm nằm hoàn toàn trong vùng không an toàn, tức $\xi_n > 0$. Ta có thể suy ra $\mu_n = 0$ và $\lambda_n = C$.

Ngoài ra, những điểm tương ứng với $\lambda_n > 0$ bây giờ là sẽ là các *support vectors*. Mặc dù những điểm này có thể không nằm trên margins, chúng vẫn được coi là *support vectors* vì có công đóng góp cho việc tính toán \mathbf{w} thông qua phương trình (5.6).

Như vậy, dựa trên các giá trị của λ_n ta có thể dự đoán được vị trí tương đối của \mathbf{x}_n so với hai margins. Đặt $\mathcal{M} = \{n : 0 < \lambda_n < C\}$ và $\mathcal{S} = \{m : 0 < \lambda_m\}$. Tức \mathcal{M} là tập hợp các chỉ số của các điểm nằm chính xác trên margins - hỗ trợ cho việc tính \mathbf{w} , \mathbf{S} là tập hợp các chỉ số của các *support vectors* - hỗ trợ trực tiếp cho việc tính b . Tương tự như với Hard Margin SVM, các hệ số \mathbf{w}, b có thể được xác định bởi:

$$\mathbf{w} = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m \quad (5.18)$$

$$b = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} (y_n - \mathbf{w}^T \mathbf{x}_n) = \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (5.19)$$

Nhắc lại rằng mục đích cuối cùng là xác định nhãn cho một điểm mới chứ không phải là tính \mathbf{w} và b nên ta quan tâm hơn tới cách xác định giá trị của biểu thức sau với một điểm dữ liệu \mathbf{x} bất kỳ:

$$\mathbf{w}^T \mathbf{x} + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \quad (5.20)$$

Và trong cách tính này, ta chỉ cần quan tâm tới tích vô hướng của hai điểm bất kỳ. Ở bài sau các bạn sẽ thấy rõ lợi ích của việc này nhiều hơn.

5.4 Bài toán tối ưu không ràng buộc cho Soft Margin SVM

Trong mục này, chúng ta sẽ đưa bài toán tối ưu có ràng buộc (5.4) về một bài toán tối ưu không ràng buộc, và có khả năng giải được bằng các phương pháp Gradient Descent.

5.4.1 Bài toán tối ưu không ràng buộc tương đương

Để ý thấy rằng điều kiện ràng buộc thứ nhất:

$$1 - \xi_n - y_n(\mathbf{w}^T \mathbf{x} + b) \leq 0 \Leftrightarrow \xi_n \geq 1 - y_n(\mathbf{w}^T \mathbf{x} + b) \quad (5.21)$$

Kết hợp với điều kiện $\xi_n \geq 0$ ta sẽ thu được bài toán ràng buộc tương đương với bài toán (5.4) như sau:

$$(\mathbf{w}, b, \xi) = \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \xi_n \quad (5.22)$$

$$\text{subject to: } \xi_n \geq \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x} + b)), \forall n = 1, 2, \dots, N$$

Tiếp theo, để đưa bài toán (5.22) về dạng không ràng buộc, chúng ta sẽ chứng minh nhận xét sau đây bằng phương pháp phản chứng:

Nếu (\mathbf{w}, b, ξ) là nghiệm của bài toán tối ưu (5.22), tức tại đó hàm mục tiêu đạt giá trị nhỏ nhất, thì:

$$\xi_n = \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)), \forall n = 1, 2, \dots, N \quad (5.23)$$

Thật vậy, giả sử ngược lại, tồn tại n sao cho:

$$\xi_n > \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))$$

ta chọn $\xi'_n = \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))$, ta sẽ thu được một giá trị thấp hơn của hàm mục tiêu, trong khi tất cả các ràng buộc vẫn được thoả mãn. Điều này mâu thuẫn với việc hàm mục tiêu đã đạt giá trị nhỏ nhất!

Vậy nhận xét (5.23) được chứng minh.

Khi đó, ta thay toàn bộ các giá trị của ξ_n trong (5.23) vào hàm mục tiêu:

$$(\mathbf{w}, b, \xi) = \arg \min_{\mathbf{w}, b, \xi} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \quad (5.24)$$

$$\text{subject to: } \xi_n = \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)), \forall n = 1, 2, \dots, N$$

Rõ ràng biến số ξ không còn quan trọng trong bài toán này nữa, ta có thể lược bỏ nó mà không làm thay đổi nghiệm của bài toán:

$$(\mathbf{w}, b) = \arg \min_{\mathbf{w}, b} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \triangleq \arg \min_{\mathbf{w}, b} J(\mathbf{w}, b) \quad (5.25)$$

Đây là một bài toán tối ưu không ràng buộc với hàm mất mát $J(\mathbf{w}, b)$. Bài toán này có thể giải được bằng các phương pháp Gradient Descent. Nhưng trước hết, chúng ta cùng xem xét hàm mất mát này từ một góc nhìn khác, bằng định nghĩa của một hàm gọi là *hinge loss*

5.4.2 Hinge loss

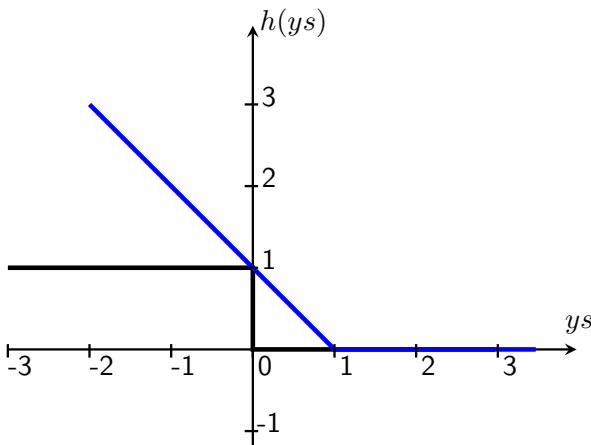
Nhắc lại một chút về hàm *cross entropy* chúng ta đã biết từ bài *Logistic Regression* và *Softmax Regression*. Với mỗi cặp hệ số (\mathbf{w}, b) và cặp dữ liệu, nhãn (\mathbf{x}_n, y_n) , đặt $z_n = \mathbf{w}^T \mathbf{x}_n + b$ và $a_n = \sigma(z_n)$ (σ là *sigmoid function*). Hàm cross entropy được định nghĩa là:

$$J_n^1(\mathbf{w}, b) = -(y_n \log(a_n) + (1 - y_n) \log(1 - a_n)) \quad (5.26)$$

Chúng ta đã biết rằng, hàm cross entropy đạt giá trị càng nhỏ nếu xác suất a_n càng gần với y_n ($0 < a_n < 1$).

Ở đây, chúng ta làm quen với một hàm số khác cũng được sử dụng nhiều trong các classifiers:

$$J_n(\mathbf{w}, b) = \max(0, 1 - y_n z_n)$$



Hình 5.3: Hinge loss (màu xanh) và zeros-one loss (màu đen). Với zero-one loss, những điểm nằm xa margin (hoành độ bằng 1) và boundary (hoành độ bằng 0) được đối xử như nhau. Trong khi đó, với hinge loss, những điểm ở xa gây ra mất mát nhiều hơn.

Hàm này có tên là *hinge loss*. Trong đó, z_n có thể được coi là *score* của \mathbf{x}_n ứng với cặp hệ số (\mathbf{w}, b) , y_n chính là đầu ra mong muốn.

Hình 5.3 mô tả hàm số *hinge loss* $f(ys) = \max(0, 1 - ys)$ và so sánh với hàm zero-one loss. Hàm zero-one loss là hàm *đếm số điểm bị misclassified*.

Trong Hình 3, biến số là y là tích của đầu ra mong muốn (ground truth) và đầu ra tính được (score). Những điểm ở phía phải của trực tung ứng với những điểm được phân loại đúng, tức s tìm được cùng dấu với y . Những điểm ở phía trái của trực tung ứng với các điểm bị phân loại sai. Ta có các nhận xét:

- Với hàm zero-one loss, các điểm có *score* ngược dấu với đầu ra mong muốn sẽ gây ra mất mát như nhau (bằng 1), bất kể chúng ở gần hay xa đường phân chia (trực tung). Đây là một hàm rời rạc, rất khó tối ưu và ta cũng khó có thể đếm được *sự hy sinh* như đã định nghĩa ở phần đầu.
- Với hàm *hinge loss*, những điểm nằm trong vùng an toàn, ứng với $ys \geq 1$, sẽ không gây ra mất mát gì. Những điểm nằm giữa margin của class tương ứng và đường phân chia tương ứng với $0 < y < 1$, những điểm này gây ra một mất mát nhỏ. Những điểm bị *misclassified*, tức $y < 0$ sẽ gây ra mất mát lớn hơn, vì vậy, khi tối thiểu hàm mất mát, ta sẽ tránh được những điểm bị *misclassified* và lấn sang phần class còn lại quá nhiều. Đây chính là một ưu điểm của hàm *hinge loss*.
- Hàm *hinge loss* là một hàm liên tục, và có *đạo hàm tại gần như mọi nơi* (*almost everywhere differentiable*) trừ điểm có hoành độ bằng 1. Ngoài ra, đạo hàm của hàm này cũng rất dễ xác định: bằng -1 tại các điểm nhỏ hơn 1 và bằng 0 tại các điểm lớn hơn 1. Tại 1, ta có thể coi như đạo hàm của nó bằng 0 giống như cách tính đạo hàm của [hàm ReLU](#).

5.4.3 Xây dựng hàm mất mát

Bây giờ, nếu ta xem xét bài toán Soft Margin SVM dưới góc nhìn hinge loss:

Với mỗi cặp (\mathbf{w}, b) , đặt:

$$L_n(\mathbf{w}, b) = \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) \quad (5.27)$$

Lấy tổng tất cả các *loss* này (giống như cách mà Logistic Regression hay Softmax Regression lấy tổng của tất cả các cross entropy loss) theo n ta được:

$$L(\mathbf{w}, b) = \sum_{n=1}^N L_n = \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b))$$

Câu hỏi đặt ra là, nếu ta trực tiếp tối ưu tổng các hinge loss này thì điều gì sẽ xảy ra?

Trong trường hợp dữ liệu trong hai class là *linearly separable*, ta sẽ có giá trị tối ưu tìm được của $L(\mathbf{w}, b)$ là bằng 0. Điều này có nghĩa là:

$$1 - y_n(\mathbf{w}^T \mathbf{x}_n + b) \leq 0, \forall n = 1, 2, \dots, N \quad (5.28)$$

Nhân cả hai về với một hằng số $a > 1$ ta có:

$$a - y_n(a\mathbf{w}^T \mathbf{x}_n + ab) \leq 0, \forall n = 1, 2, \dots, N \quad (5.29)$$

$$\Rightarrow 1 - y_n(a\mathbf{w}^T \mathbf{x}_n + ab) \leq 1 - a < 0, \forall n = 1, 2, \dots, N \quad (5.30)$$

Điều này nghĩa là $(a\mathbf{w}, ab)$ cũng là nghiệm của bài toán. Nếu không có điều kiện gì thêm, bài toán có thể dẫn tới nghiệm không ổn định vì các hệ số của nghiệm có thể lớn tùy ý!

Để tránh *bug* này, chúng ta cần thêm một số hạng nữa vào $L(\mathbf{w}, b)$ gọi là số hạng *regularization*, giống như cách chúng ta đã làm để tránh *overfitting* trong neural networks. Lúc này, ta sẽ có hàm mất mát tổng cộng là:

$$J(\mathbf{w}, b) = L(\mathbf{w}, b) + \lambda R(\mathbf{w}, b)$$

với λ là một số dương, gọi là *regularization parameter*, hàm $R()$ sẽ giúp hạn chế việc các hệ số (\mathbf{w}, b) trở nên quá lớn. Có nhiều cách chọn hàm $R()$, nhưng cách phổ biến nhất là l_2 , khi đó hàm mất mát của Soft Margin SVM sẽ là:

$$J(\mathbf{w}, b) = \sum_{n=1}^N \max(0, 1 - y_n(\mathbf{w}^T \mathbf{x}_n + b)) + \frac{\lambda}{2} \|\mathbf{w}\|_2^2 \quad (5.31)$$

Kỹ thuật này còn gọi là *weight decay*. Chú ý rằng *weight decay* thường không được áp dụng lên thành phần *bias* b .

Ta thấy rằng hàm mất mát (5.31) giống với hàm mất mát (5.25) với $\lambda = \frac{1}{C}$. Ở đây, tôi đã lấy $\lambda/2$ để biểu thức đạo hàm được *đẹp hơn*.

Trong phần tiếp theo của mục này, chúng ta sẽ quan tâm tới bài toán tối ưu hàm mất mát được cho trong (5.31).

Nhận thấy rằng ta có thể khiến biểu thức (5.24) gọn hơn một chút bằng cách sử dụng *bias trick* như đã làm trong Linear Regression hay các bài về neural networks. Bằng cách *mở rộng* thêm một thành phần bằng 1 vào các điểm dữ liệu $\mathbf{x}_n \in \mathbb{R}^d$ để được $\bar{\mathbf{x}}_n \in \mathbb{R}^{d+1}$ và kết hợp \mathbf{w}, b thành một vector $\bar{\mathbf{w}} = [\mathbf{w}^T, b]^T \in \mathbb{R}^{d+1}$ ta sẽ có một biểu thức gọn hơn. Khi đó, hàm mất mát trong (5.31) có thể được viết gọn thành:

$$J(\bar{\mathbf{w}}) = \underbrace{\sum_{n=1}^N \max(0, 1 - y_n \bar{\mathbf{w}}^T \bar{\mathbf{x}}_n)}_{\text{hinge loss}} + \underbrace{\frac{\lambda}{2} \|\mathbf{w}\|_2^2}_{\text{regularization}} \quad (5.32)$$

Các bạn có thể nhận thấy đây là một hàm lồi theo $\bar{\mathbf{w}}$ vì:

- $1 - y_n \bar{\mathbf{w}}^T \bar{\mathbf{x}}_n$ là 1 hàm tuyến tính nên nó là một hàm lồi. Hàm hằng số là một hàm lồi, max của hai hàm lồi là một hàm lồi. Vậy biểu thức hinge loss là một hàm lồi.
- Norm là một hàm lồi, vậy số hạng regularization cũng là một hàm lồi.
- Tổng của hai hàm lồi là một hàm lồi.

Vì bài toán tối ưu bây giờ là không ràng buộc, chúng ta có thể sử dụng các phương pháp Gradient Descent để tối ưu. Hơn nữa, vì tính chất lồi của hàm mất mát, nếu chọn *learning rate* không quá lớn và số vòng lặp đủ nhiều, thuật toán sẽ hội tụ tới điểm *global optimal* của bài toán.

5.4.4 Tối ưu hàm mất mát

Trước hết ta cần tính được đạo hàm của hàm mất mát theo $\bar{\mathbf{w}}$. Việc này thoảng qua có vẻ hơi phức tạp vì ta cần tính đạo hàm của hàm max, nhưng nếu chúng ta nhìn vào đạo hàm của hinge loss, ta có thể tính được đạo hàm theo $\bar{\mathbf{w}}$ một cách đơn giản.

Chúng ta tạm quên đi đạo hàm của phần regularization vì nó đơn giản bằng $\lambda \begin{bmatrix} \mathbf{w} \\ 0 \end{bmatrix}$ với thành phần 0 ở cuối chính là đạo hàm theo bias của thành phần regularization.

Với phần hinge loss, xét từng điểm dữ liệu, ta có hai trường hợp:

- TH1: Nếu $1 - y_n \bar{\mathbf{w}}^T \bar{\mathbf{x}}_n \leq 0$, ta có ngay đạo hàm theo $\bar{\mathbf{w}}$ bằng 0.
- TH2: Nếu $1 - y_n \bar{\mathbf{w}}^T \bar{\mathbf{x}}_n > 0$, đạo hàm theo \mathbf{w} chính là $-y_n \mathbf{x}_n$.

Để tính gradient cho hàm măt măt trên toàn bộ dữ liệu, chúng ta cần một chút kỹ năng biến đổi đại số tuyến tính.

Đặt:

$$\mathbf{Z} = [y_1 \bar{\mathbf{x}}_1, y_2 \bar{\mathbf{x}}_2, \dots, y_N \bar{\mathbf{x}}_N] \quad (5.33)$$

$$\mathbf{u} = [y_1 \bar{\mathbf{w}}^T \bar{\mathbf{x}}_1, y_2 \bar{\mathbf{w}}^T \bar{\mathbf{x}}_2, \dots, y_N \bar{\mathbf{w}}^T \bar{\mathbf{x}}_N] = \bar{\mathbf{w}}^T \mathbf{Z} \quad (5.34)$$

với chú ý rằng \mathbf{u} là một vector hàng.

Tiếp tục, ta cần xác định các vị trí của \mathbf{u} có giá trị nhỏ hơn 1, tức ứng với TH2 ở trên. Bằng cách đặt:

$$\mathcal{H} = \{n : u_n < 1\}$$

ta có thể suy ra cách tính đạo hàm theo $\bar{\mathbf{w}}$ của hàm măt măt là:

$$\nabla J(\bar{\mathbf{w}}) = \sum_{n \in \mathcal{H}} -y_n \bar{\mathbf{x}}_n + \lambda \begin{bmatrix} \mathbf{w} \\ 0 \end{bmatrix} \quad (5.35)$$

Các bạn sẽ thấy cách tính toán giá trị này một cách hiệu quả trong phần lập trình.

Vậy quy tắc cập nhật của $\bar{\mathbf{w}}$ sẽ là:

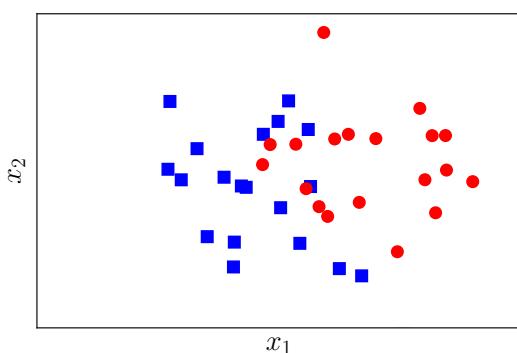
$$\bar{\mathbf{w}} = \bar{\mathbf{w}} - \eta \left(\sum_{n \in \mathcal{H}} -y_n \bar{\mathbf{x}}_n + \lambda \begin{bmatrix} \mathbf{w} \\ 0 \end{bmatrix} \right) \quad (5.36)$$

với η là *learning rate*.

Với các bài toán large-scale, ta có thể sử dụng phương pháp Mini-batch Gradient Descent để tối ưu. Đây chính là một ưu điểm của hướng tiếp cận theo hinge loss.

5.5 Kiểm chứng bằng lập trình

Trong mục này, chúng ta cùng làm hai thí nghiệm nhỏ. Thí nghiệm thứ nhất sẽ đi tìm nghiệm của một bài toán Soft Margin SVM bằng ba cách khác nhau: Sử dụng thư viện sklearn, Giải bài toán đối ngẫu bằng CVXOPT, và Tối ưu hàm măt măt không ràng buộc bằng phương pháp Gradient Descent. Nếu mọi tính toán ở trên là chính xác, nghiệm của ba cách làm này sẽ giống nhau, khác nhau có thể một chút bởi sai số trong tính toán. Ở thí nghiệm thứ hai, chúng ta sẽ thay C bởi những giá trị khác nhau và cùng xem các margin thay đổi như thế nào.



Hình 5.4: Tạo dữ liệu cho thí nghiệm. Dữ liệu của hai class là gần như linearly separable.

5.5.1 Giải bài toán Soft Margin bằng 3 cách khác nhau

Source code cho phần này có thể được tìm thấy [tại đây](#).

Khai báo thư viện và tạo dữ liệu giả

```
# generate data
# list of points
import numpy as np
import matplotlib.pyplot as plt
from scipy.spatial.distance import cdist
np.random.seed(21)
from matplotlib.backends.backend_pdf import PdfPages

means = [[2, 2], [4, 1]]
cov = [[.3, .2], [.2, .3]]
N = 10
X0 = np.random.multivariate_normal(means[0], cov, N)
X1 = np.random.multivariate_normal(means[1], cov, N)
X1[-1, :] = [2.7, 2]
X = np.concatenate((X0.T, X1.T), axis = 1)
y = np.concatenate((np.ones((1, N)), -1*np.ones((1, N))), axis = 1)
```

Hình 5.4 minh họa các điểm dữ liệu của hai classes.

Giải bài toán bằng thư viện sklearn

Ta chọn $C = 100$ trong thí nghiệm này: cáchте

```
from sklearn.svm import SVC
C = 100
clf = SVC(kernel = 'linear', C = C)
clf.fit(X, y)

w_sklearn = clf.coef_.reshape(-1, 1)
b_sklearn = clf.intercept_[0]
print(w_sklearn.T, b_sklearn)
```

Nghiệm tìm được:

```
[[ -1.87461946 -1.80697358]] 8.49691190196
```

Tìm nghiệm bằng cách giải bài toán đối ngẫu

Tương tự như việc giải bài toán Hard Margin SVM, chỉ khác rằng ta có thêm ràng buộc về chặn trên của các nhân thủ Lagrange:

```
from cvxopt import matrix, solvers
# build K
V = np.concatenate((X0.T, -X1.T), axis = 1)
K = matrix(V.T.dot(V))

p = matrix(-np.ones((2*N, 1)))
# build A, b, G, h
G = matrix(np.vstack((-np.eye(2*N), np.eye(2*N) )))

h = matrix(np.vstack((np.zeros((2*N, 1)), C*np.ones((2*N, 1)) )))
A = matrix(y.reshape((-1, 2*N)))
b = matrix(np.zeros((1, 1)))
solvers.options['show_progress'] = False
sol = solvers.qp(K, p, G, h, A, b)

l = np.array(sol['x'])
print('lambda = \n', l.T)

lambda =
[[ 1.11381472e-06   9.99999967e+01   1.10533112e-06   6.70163540e-06
  3.40838760e+01   4.73972850e-06   9.99999978e+01   3.13320446e-06
  9.99999985e+01   5.06729333e+01   9.99999929e+01   3.23564235e-06
  9.99999984e+01   9.99999948e+01   1.37977626e-06   9.99997155e+01
  3.45005660e-06   1.46190314e-06   5.50601997e-06   1.45062544e-06
  1.85373848e-06   1.14181647e-06   8.47565685e+01   9.99999966e+01
  9.99999971e+01   8.00764708e-07   2.65537193e-06   1.45230729e-06
  4.15737085e-06   9.99999887e+01   9.99999761e+01   8.98414770e-07
  9.99999979e+01   1.75651607e-06   8.27947897e-07   1.04289116e-06
  9.99999969e+01   9.07920759e-07   8.83138295e-07   9.99999971e+01]]
```

Trong các thành phần của **lambda** tìm được, có rất nhiều thành phần nhỏ tới **1e-6** hay **1e-7**. Đây chính là các **lambda_i = 0**. Có rất nhiều phần tử xấp xỉ **9.99e+01**, đây chính là các **lambda_i** bằng với **C = 100**, tương ứng với các support vectors không nằm trên margins, các sai số nhỏ xảy ra do tính toán. Các giá trị còn lại nằm giữa **0** và **100** là các giá trị tương ứng với các điểm nằm chính xác trên hai margins.

Tiếp theo, ta cần tính **w** và **b** theo công thức (15) và (16). Trước đó ta cần tìm tập hợp các điểm support và những điểm nằm trên margins.

```
S = np.where(l > 1e-5)[0] # support set
S2 = np.where(l < .999*C)[0]

M = [val for val in S if val in S2] # intersection of two lists

XT = X.T # we need each column to be one data point in this alg
VS = V[:, S]
lS = l[S]
yM = y[M]
```

```

XM = XT[:, M]

w_dual = VS.dot(lS).reshape(-1, 1)
b_dual = np.mean(yM.T - w_dual.T.dot(XM))
print(w_dual.T, b_dual)

```

Kết quả:

```
[[-1.87457279 -1.80695039]] 8.49672109815
```

Kết quả này gần giống với kết quả tìm được bằng sklearn.

Tìm nghiệm bằng giải bài toán tối ưu không ràng buộc

Trong phương pháp này, chúng ta cần tính gradient của hàm mất mát. Như thường lệ, chúng ta cần kiểm chứng này bằng cách so sánh với *numerical gradient*.

Chú ý rằng trong phương pháp này, ta cần dùng tham số `lam = 1/C`.

```

X0_bar = np.vstack((X0.T, np.ones((1, N)))) # extended data
X1_bar = np.vstack((X1.T, np.ones((1, N)))) # extended data

Z = np.hstack((X0_bar, - X1_bar)) # as in (22)
lam = 1./C

def cost(w):
    u = w.T.dot(Z) # as in (23)
    return (np.sum(np.maximum(0, 1 - u)) + \
            .5*lam*np.sum(w*w)) - .5*lam*w[-1]*w[-1] # no bias

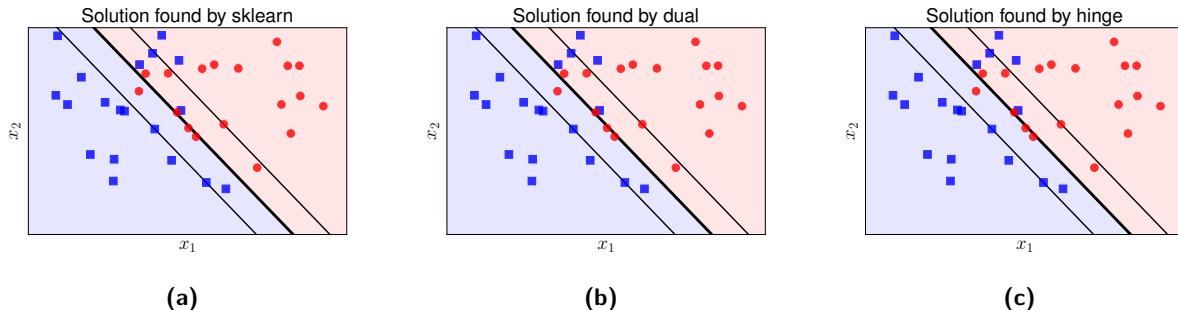
def grad(w):
    u = w.T.dot(Z) # as in (23)
    H = np.where(u < 1)[1]
    ZS = Z[:, H]
    g = (-np.sum(ZS, axis = 1, keepdims = True) + lam*w)
    g[-1] -= lam*w[-1] # no weight decay on bias
    return g

eps = 1e-6
def num_grad(w):
    g = np.zeros_like(w)
    for i in xrange(len(w)):
        wp = w.copy()
        wm = w.copy()
        wp[i] += eps
        wm[i] -= eps
        g[i] = (cost(wp) - cost(wm)) / (2*eps)
    return g

w0 = np.random.randn(X0_ext.shape[0], 1)
g1 = grad(w0)
g2 = num_grad(w0)
diff = np.linalg.norm(g1 - g2)
print('Gradient different: %f' %diff)

```

```
Gradient difference: 0.000000
```



Hình 5.5: Các đường phân chia tìm được bởi ba cách khác nhau: a) hứa viện sklearn, b) Bài toán đối ngẫu, c) Hàm hinge loss. Các kết quả tìm được là như nhau.

Vì sự khác nhau giữa hai cách tính gradient là bằng 0, ta có thể yên tâm rằng gradient tính được là chính xác.

Sau khi chắc chắn rằng gradient tìm được đã chính xác, ta có thể bắt đầu làm Gradient Descent:

```

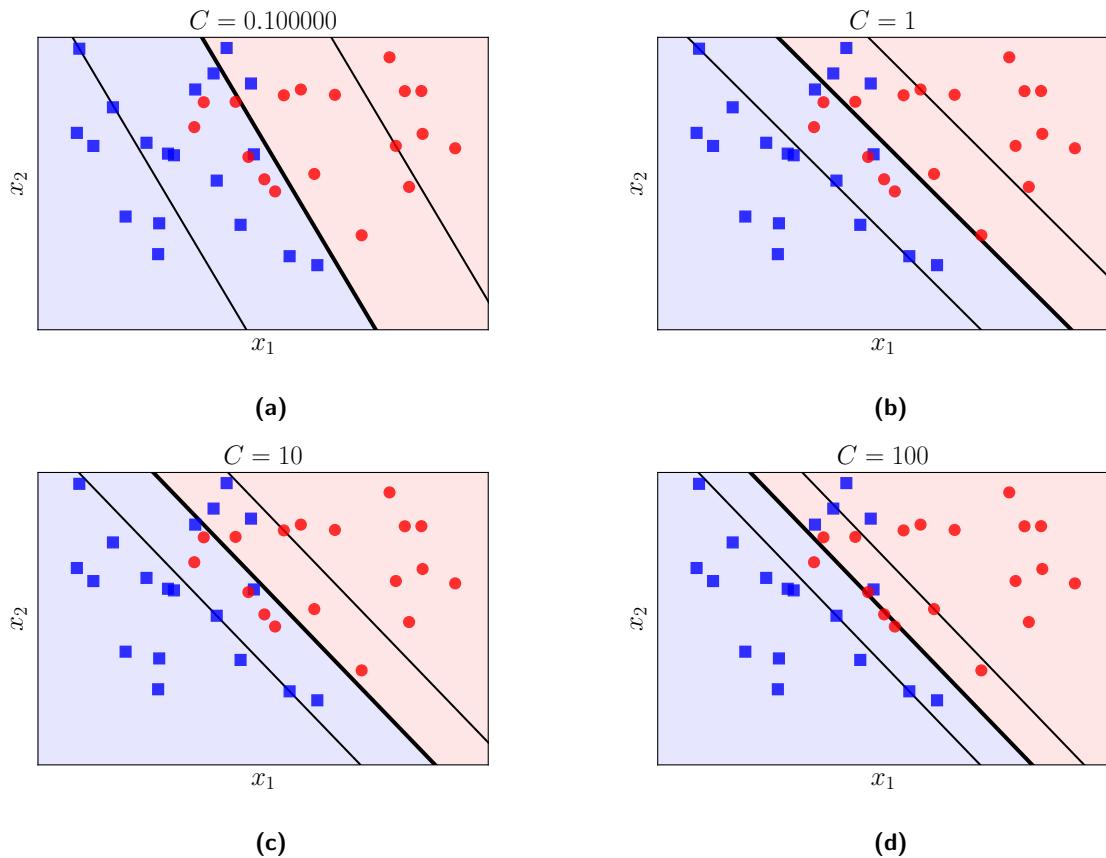
def grad_descent(w0, eta):
    w = w0
    it = 0
    while it < 100000:
        it = it + 1
        g = grad(w)
        w -= eta*g
        if (it % 10000) == 1:
            print('iter %d' %it + ' cost: %f' %cost(w))
        if np.linalg.norm(g) < 1e-5:
            break
    return w
w0 = np.random.randn(X0_ext.shape[0], 1)
w = grad_descent(w0, 0.001)
w_hinge = w[:-1].reshape(-1, 1)
b_hinge = w[-1]
print(w_hinge.T, b_hinge)

```

Kết quả:

Ta thấy rằng kết quả tìm được bằng ba cách là như nhau. Hình 5 dưới đây minh họa kết

Trong thực hành, phương pháp 1 chắc chắn được lựa chọn. Hai phương pháp còn lại được



Hình 5.6: Ảnh hưởng của C lên nghiệm của Soft Margin SVM. Khi C càng lớn thì biên càng nhỏ, và ngược lại.

5.5.2 Ảnh hưởng của C lên nghiệm

Hình 6 dưới đây minh họa nghiệm tìm được cho bài toán phía trên nhưng với các giá trị C khác nhau. Nghiệm được tìm bằng thư viện sklearn.

Chúng ta nhận thấy rằng khi C càng lớn thì biên càng nhỏ đi. Điều này phù hợp với suy luận của chúng ta ở [Mục 2](#).

5.6 Tóm tắt và thảo luận

- SVM thuần (Hard Margin SVM) hoạt động không hiệu quả khi có nhiều ở gần biên hoặc thậm chí khi dữ liệu giữa hai lớp gần *linearly separable*. Soft Margin SVM có thể giúp khắc phục điểm này.

- Trong Soft Margin SVM, chúng ta chấp nhận lỗi xảy ra ở một vài điểm dữ liệu. Lỗi này được xác định bằng khoảng cách từ điểm đó tới đường biên tương ứng. Bài toán tối ưu sẽ tối thiểu lỗi này bằng cách sử dụng thêm các biến được gọi là *slack variables*.
- Để giải bài toán tối ưu, có hai cách khác nhau. Mỗi cách có những ưu, nhược điểm riêng, các bạn sẽ thấy trong các bài tới.
- Cách thứ nhất là giải bài toán đối ngẫu. Bài toán đối ngẫu của Soft Margin SVM rất giống với bài toán đối ngẫu của Hard Margin SVM, chỉ khác ở ràng buộc chặn trên của các nhân tử Lagrange. Ràng buộc này còn được gọi là *box constraint*.
- Cách thứ hai là đưa bài toán về dạng không ràng buộc dựa trên một hàm mới gọi là *hinge loss*. Với cách này, hàm mất mát thu được là một hàm lồi và có thể giải được khá dễ dàng và hiệu quả bằng các phương pháp Gradient Descent.
- Trong Soft Margin SVM, có một hằng số phải được chọn, đó là C . Hướng tiếp cận này còn được gọi là C-SVM. Ngoài ra, còn có một hướng tiếp cận khác cũng hay được sử dụng, gọi là ν -SVM, bạn đọc có thể đọc thêm [tại đây](#).
- [Source code](#)

5.7 Tài liệu tham khảo

- [1] Bishop, Christopher M. "Pattern recognition and Machine Learning.", Springer (2006). ([book](#))
- [2] Duda, Richard O., Peter E. Hart, and David G. Stork. Pattern classification. John Wiley & Sons, 2012.
- [3] [sklearn.svm.SVC](#)
- [4] [LIBSVM – A Library for Support Vector Machines](#)
- [5] Bennett, K. P. (1992). “Robust linear programming discrimination of two linearly separable sets”. *Optimization Methods and Software* 1, 23–34.
- [6] Schölkopf, Bernhard, et al. “[New support vector algorithms](#).” Neural computation 12.5 (2000): 1207-1245.
- [7] Rosasco, L.; De Vito, E. D.; Caponnetto, A.; Piana, M.; Verri, A. (2004). “[Are Loss Functions All the Same?](#)”. *Neural Computation*. 16 (5): 1063–1076

6

Kernel Support Vector Machine

Có một sự tương ứng thú vị giữa hai nhóm thuật toán phân lớp phổ biến nhất: Neural Networks và Support Vector Machines. Chúng đều bắt đầu từ bài toán phân lớp với hai *linearly separable classes*, tiếp theo đến hai *almost linearly separable classes*, đến bài toán có nhiều classes rồi các bài toán với các classes hoàn toàn không *linearly separable*. Sự tương ứng được cho trong Bảng 6.1

Neural Networks	Support Vector Machines	Tính chất chung
PLA	Hard Margin SVM	Hai classes là <i>linearly separable</i>
Logistic Regression	Soft Margin SVM	Hai classes là <i>gần linearly separable</i>
Softmax Regression	Multi-class SVM	Bài toán phân loại nhiều classes (biên là tuyến tính)
Multi-layer Perceptron	Kernel SVM	Bài toán với dữ liệu không <i>linearly separable</i>

Bảng 6.1: Sự tương đồng giữa Neural Networks và Support Vector Machines

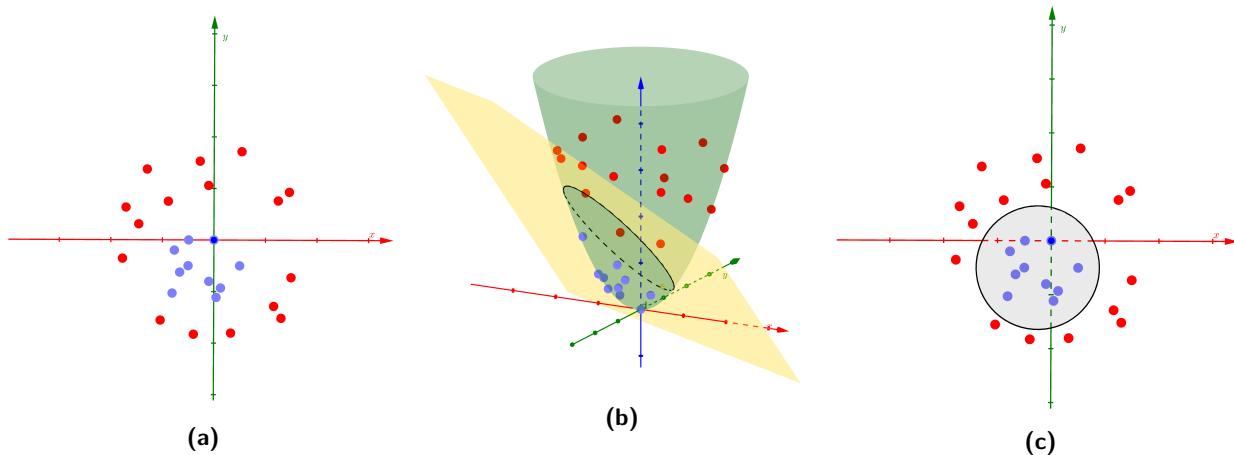
Chương này sẽ nói về Kernel SVM, tức việc áp dụng SVM lên bài toán mà dữ liệu giữa hai classes là hoàn toàn *không linear separable*. Bài toán phân biệt nhiều classes sẽ được trình bày trong Bài 22: Multiclass SVM.

Ý tưởng cơ bản của Kernel SVM và các phương pháp kernel nói chung là tìm một phép biến đổi sao cho dữ liệu ban đầu là *không phân biệt tuyến tính* được biến sang không gian mới. Ở không gian mới này, dữ liệu trở nên *phân biệt tuyến tính*.

Xét ví dụ dưới đây với việc biến dữ liệu *không phân biệt tuyến tính* trong không gian hai chiều thành *phân biệt tuyến tính* trong không gian ba chiều bằng cách giới thiệu thêm một chiều mới (xem Hình 6.1).

Để xem ví dụ này một cách sinh động hơn, bạn có thể xem [clip này](#).

Nói một cách ngắn gọn, Kernel SVM là việc đi tìm một hàm số biến đổi dữ liệu \mathbf{x} từ không gian feature ban đầu thành dữ liệu trong một không gian mới bằng hàm số $\Phi(\mathbf{x})$. Trong ví dụ này, hàm $\Phi()$ đơn giản là giới thiệu thêm một chiều dữ liệu mới (một feature mới) là một



Hình 6.1: Ví dụ về Kernel SVM. a) Dữ liệu của hai classes là *không phân biệt tuyến tính* trong không gian hai chiều. b) Nếu coi thêm chiều thứ ba là một hàm số của hai chiều còn lại $z = x^2 + y^2$, các điểm dữ liệu sẽ được phân bố trên 1 parabolic và đã trở nên *phân biệt tuyến tính*. Mặt phẳng màu vàng là mặt phân chia, có thể tìm được bởi Hard/Soft Margin SVM. c) Giao điểm của mặt phẳng tìm được và mặt parabolic là một đường ellipse, khi chiếu toàn bộ dữ liệu cũng như đường ellipse này xuống không gian hai chiều ban đầu, ta đã tìm được đường phân chia hai classes.

hàm số của các *features* đã biết. Hàm số này cần thỏa mãn mục đích của chúng ta: trong không gian mới, dữ liệu giữa hai classes là *phân biệt tuyến tính* hoặc *gần như phân biệt tuyến tính*. Khi đó, ta có thể dùng các bộ phân lớp tuyến tính thông thường như PLA, Logistic Regression, hay Hard/Soft Margin SVM.

Nếu phải so sánh, ta có thể thấy rằng hàm biến đổi $\Phi()$ tương tự như *activation functions* trong Neural Networks. Tuy nhiên, có một điểm khác biệt ở đây là: trong khi nhiệm vụ của activation function là phá vỡ tính tuyến tính của *mô hình*, hàm biến đổi $\Phi()$ đi biến dữ liệu không phân biệt tuyến tính thành phân biệt tuyến tính. Như vậy là để đạt được mục đích chung, ta có hai cách nhìn khác nhau về cách giải quyết.

Các hàm $\Phi()$ thường tạo ra dữ liệu mới có số chiều cao hơn số chiều của dữ liệu ban đầu, thậm chí là vô hạn chiều. Nếu tính toán các hàm này trực tiếp, chắc chắn chúng ta sẽ gặp các vấn đề về bộ nhớ và hiệu năng tính toán. Có một cách tiếp cận là sử dụng các *kernel functions* mô tả quan hệ giữa hai điểm dữ liệu bất kỳ trong không gian mới, thay vì đi tính toán trực tiếp từng điểm dữ liệu trong không gian mới. Kỹ thuật này được xây dựng dựa trên quan sát về **bài toán đối ngẫu của SVM**.

Trong Mục 2 dưới đây, chúng ta cùng tìm hiểu cơ sở toán học của Kernel SVM và Mục 3 sẽ giới thiệu một số hàm Kernel thường được sử dụng.

6.1 Cơ sở toán học

Tôi xin nhắc lại bài toán đối ngẫu trong Soft Margin SVM cho dữ liệu *gần phân biệt tuyến tính*:

$$\begin{aligned} \boldsymbol{\lambda} = \arg \max_{\boldsymbol{\lambda}} & \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \mathbf{x}_n^T \mathbf{x}_m \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \quad \forall n = 1, 2, \dots, N \end{aligned} \tag{6.1}$$

Trong đó:

- N : số cặp điểm dữ liệu trong tập training.
- \mathbf{x}_n : feature vector của dữ liệu thứ n trong tập training.
- y_n : nhãn của dữ liệu thứ n , bằng 1 hoặc -1.
- λ_n : nhân tử Lagrange ứng với điểm dữ liệu thứ n .
- C : hằng số dương giúp cân đối độ lớn của margin và sự hy sinh của các điểm nằm trong vùng *không an toàn*. Khi $C = \infty$ hoặc rất lớn, Soft Margin SVM trở thành Hard Margin SVM.

Sau khi giải được λ cho bài toán (6.1), nhãn của một điểm dữ liệu mới sẽ được xác định bởi dấu của biểu thức:

$$\sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x} + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \mathbf{x}_m^T \mathbf{x}_n \right) \tag{6.2}$$

Trong đó:

- $\mathcal{M} = \{n : 0 < \lambda_n < C\}$ là tập hợp những điểm nằm trên margin.
- $\mathcal{S} = \{n : 0 < \lambda_n\}$ là tập hợp các điểm support.
- $N_{\mathcal{M}}$ là số phần tử của \mathcal{M} .

Với dữ liệu thực tế, rất khó để có dữ liệu *gần phân biệt tuyến tính*, vì vậy nghiệm của bài toán (6.1) có thể không thực sự tạo ra một bộ phân lớp tốt. Giả sử rằng ta có thể tìm được

hàm số $\Phi()$ sao cho sau khi được biến đổi sang không gian mới, mỗi điểm dữ liệu \mathbf{x} trở thành $\Phi(\mathbf{x})$, và trong không gian mới này, dữ liệu trở nên *gần phân biệt tuyến tính*. Lúc này, *hy vọng rằng* nghiệm của bài toán Soft Margin SVM sẽ cho chúng ta một bộ phân lớp tốt hơn.

Trong không gian mới, bài toán (6.1) trở thành:

$$\begin{aligned} \boldsymbol{\lambda} = \arg \max_{\boldsymbol{\lambda}} & \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m \Phi(\mathbf{x}_n)^T \Phi(\mathbf{x}_m) \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \end{aligned} \quad (6.3)$$

và *nhân* của một điểm dữ liệu mới được xác định bởi dấu của biểu thức:

$$\mathbf{w}^T \Phi(\mathbf{x}) + b = \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m \Phi(\mathbf{x}_m)^T \Phi(\mathbf{x}_n) \right) \quad (6.4)$$

Như đã nói ở trên, việc tính toán trực tiếp $\Phi(\mathbf{x})$ cho mỗi điểm dữ liệu có thể sẽ tốn rất nhiều bộ nhớ và thời gian vì số chiều của $\Phi(\mathbf{x})$ thường là rất lớn, có thể là vô hạn! Thêm nữa, để tìm *nhân* của một điểm dữ liệu mới \mathbf{x} , ta lại phải tìm biến đổi của nó $\Phi(\mathbf{x})$ trong không gian mới rồi lấy tích vô hướng của nó với tất cả các $\Phi(\mathbf{x}_m)$ với m trong tập hợp support. Để tránh việc này, ta quan sát thấy một điều thú vị sau đây. Trong bài toán (6.3) và biểu thức (6.4), chúng ta không cần tính trực tiếp $\Phi(\mathbf{x})$ cho mọi điểm dữ liệu. Chúng ta chỉ cần tính được $\Phi(\mathbf{x})^T \Phi(\mathbf{z})$ dựa trên hai điểm dữ liệu \mathbf{x}, \mathbf{z} bất kỳ! Kỹ thuật này còn được gọi là **kernel trick**. Những phương pháp dựa trên kỹ thuật này, tức thay vì trực tiếp tính tọa độ của một điểm trong không gian mới, ta chỉ tính tích vô hướng giữa hai điểm trong không gian mới, được gọi chung là **kernel method**.

Lúc này, bằng cách định nghĩa *hàm kernel* $k(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$, ta có thể viết lại bài toán (6.3) và biểu thức (6.4) như sau:

$$\begin{aligned} \boldsymbol{\lambda} = \arg \max_{\boldsymbol{\lambda}} & \sum_{n=1}^N \lambda_n - \frac{1}{2} \sum_{n=1}^N \sum_{m=1}^N \lambda_n \lambda_m y_n y_m k(\mathbf{x}_n, \mathbf{x}_m) \\ \text{subject to: } & \sum_{n=1}^N \lambda_n y_n = 0 \\ & 0 \leq \lambda_n \leq C, \forall n = 1, 2, \dots, N \end{aligned} \quad (6.5)$$

và:

$$\sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}) + \frac{1}{N_{\mathcal{M}}} \sum_{n \in \mathcal{M}} \left(y_n - \sum_{m \in \mathcal{S}} \lambda_m y_m k(\mathbf{x}_m, \mathbf{x}_n) \right) \quad (6.6)$$

Ví dụ: Xét phép biến đổi 1 điểm dữ liệu trong không gian hai chiều $\mathbf{x} = [x_1, x_2]^T$ thành một điểm trong không gian 5 chiều $\Phi(\mathbf{x}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2]^T$. Ta có:

$$\Phi(\mathbf{x})^T \Phi(\mathbf{z}) = [1, \sqrt{2}x_1, \sqrt{2}x_2, x_1^2, \sqrt{2}x_1x_2, x_2^2] [1, \sqrt{2}z_1, \sqrt{2}z_2, z_1^2, \sqrt{2}z_1z_2, z_2^2]^T \quad (6.7)$$

$$= 1 + 2x_1z_1 + 2x_2z_2 + x_1^2x_2^2 + 2x_1z_1x_2z_2 + x_2^2z_2^2 \quad (6.8)$$

$$= (1 + x_1z_1 + x_2z_2)^2 = (1 + \mathbf{x}^T \mathbf{z})^2 = k(\mathbf{x}, \mathbf{z}) \quad (6.9)$$

Trong ví dụ này, rõ ràng rằng việc tính toán hàm kernel $k()$ cho hai điểm dữ liệu dễ dàng hơn việc tính từng $\Phi()$ rồi nhân chúng với nhau.

Vậy những hàm số kernel cần có những tính chất gì, và những hàm như thế nào được sử dụng trong thực tế?

6.2 Hàm số kernel

6.2.1 Tính chất của các hàm kernel

Không phải hàm $k()$ bất kỳ nào cũng được sử dụng. Các hàm kernel cần có các tính chất:

- Đổi xứng: $k(\mathbf{x}, \mathbf{z}) = k(\mathbf{z}, \mathbf{x})$. Điều này dễ nhận ra vì tích vô hướng của hai vector có tính đổi xứng.
- Về lý thuyết, hàm kernel cần thỏa mãn [điều kiện Mercer](#):

$$\sum_{n=1}^N \sum_{m=1}^N k(\mathbf{x}_m, \mathbf{x}_n) c_n c_m \geq 0, \quad \forall c_i \in \mathbb{R}, i = 1, 2, \dots, N \quad (6.10)$$

Tính chất này để đảm bảo cho việc hàm mục tiêu của bài toán đối ngẫu (6.5) là *lồi*.

- Trong thực hành, có một vài hàm số $k()$ không thỏa mãn điều kiện Mercer nhưng vẫn cho kết quả chấp nhận được. Những hàm số này vẫn được gọi là kernel. Trong bài viết này, tôi chỉ tập trung vào các hàm kernel thông dụng và có sẵn trong các thư viện.

Nếu một hàm kernel thỏa mãn điều kiện (6.10), xét $c_n = y_n \lambda_n$, ta sẽ có:

$$\boldsymbol{\lambda}^T \mathbf{K} \boldsymbol{\lambda} = \sum_{n=1}^N \sum_{m=1}^N k(\mathbf{x}_m, \mathbf{x}_n) y_n y_m \lambda_n \lambda_m \geq 0, \quad \forall \lambda_n \quad (6.11)$$

với \mathbf{K} là một ma trận đối xứng mà phần tử ở hàng thứ n cột thứ m của nó được định nghĩa bởi: $k_{nm} = y_n y_m k(\mathbf{x}_n, \mathbf{x}_m)$

Từ (6.11) ta suy ra \mathbf{K} là một ma trận nửa xác định dương. Vì vậy, bài toán tối ưu (6.5) có ràng buộc là lồi và hàm mục tiêu là một hàm lồi (một quadratic form). Vì vậy chúng ta có thể giải quyết bài toán này một cách hiệu quả.

Trong bài viết này, tôi sẽ không đi sâu vào việc giải quyết bài toán (6.5) vì nó hoàn toàn tương tự như bài toán đối ngẫu của Soft Margin SVM. Thay vào đó, tôi sẽ trình bày các hàm kernel thông dụng và hiệu năng của chúng trong các bài toán thực tế. Việc này sẽ được thực hiện thông qua các ví dụ và cách sử dụng thư viện sklearn.

6.2.2 Một số hàm kernel thông dụng

Linear

Dây là trường hợp đơn giản với kernel chính tích vô hướng của hai vector:

$$k(\mathbf{x}, \mathbf{z}) = \mathbf{x}^T \mathbf{z}$$

Hàm số này, [như đã chứng minh trong Bài 19](#), thỏa mãn điều kiện (6.10).

Khi sử dụng hàm `sklearn.svm.SVC`, kernel này được chọn bằng cách đặt `kernel = 'linear'`

Polynomial

$$k(\mathbf{x}, \mathbf{z}) = (r + \gamma \mathbf{x}^T \mathbf{z})^d \quad (6.12)$$

Với d là một số dương, là bậc của đa thức. d có thể không là số tự nhiên vì mục đích chính của ta không phải là bậc của đa thức mà là cách tính kernel. Polynomial kernel có thể dùng để mô tả hầu hết các đa thức có bậc không vượt quá d nếu d là một số tự nhiên.

Phần kiểm tra liệu hàm này có thỏa mãn điều kiện (6.10) hay không xin được bỏ qua.

Khi sử dụng thư viện `sklearn`, kernel này được chọn bằng cách đặt `kernel = 'poly'`. Thông tin cụ thể về cách sử dụng có thể xem [tại đây](#).

Radial Basic Function

Radial Basic Function (RBF) kernel hay Gaussian kernel được sử dụng nhiều nhất trong thực tế, và là lựa chọn mặc định trong sklearn. Nó được định nghĩa bởi:

$$k(\mathbf{x}, \mathbf{z}) = \exp(-\gamma \|\mathbf{x} - \mathbf{z}\|_2^2), \quad \gamma > 0 \quad (6.13)$$

Trong `sklearn, kernel = 'rbf'`.

Sigmoid

Sigmoid function cũng được sử dụng làm kernel:

$$k(\mathbf{x}, \mathbf{z}) = \tanh(\gamma \mathbf{x}^T \mathbf{z} + r) \quad (6.14)$$

Trong `sklearn, kernel = 'sigmoid'`.

Bảng tóm tắt các kernel thông dụng

Dưới đây là bảng tóm tắt các kernel thông dụng và cách sử dụng trong `sklearn`.

Tên kernel	Công thức	Thiết lập hệ số
'linear'	$\mathbf{x}^T \mathbf{z}$	không có hệ số
'poly'	$(r + \gamma \mathbf{x}^T \mathbf{z})^d$	d : <code>degree</code> , γ : <code>gamma</code> , r : <code>coef0</code>
'sigmoid'	$\tanh(\gamma \mathbf{x}^T \mathbf{z} + r)$	γ : <code>gamma</code> , r : <code>coef0</code>
'rbf'	$\exp(-\gamma \ \mathbf{x} - \mathbf{z}\ _2^2)$	$\gamma > 0$: <code>gamma</code>

Bảng 6.2: Bảng các kernel thông dụng

Nếu bạn muốn sử dụng các thư viện cho C/C++, các bạn có thể tham khảo [LIBSVM](#) và [LIBLINEAR](#).

Kernel tự định nghĩa

Ngoài các hàm kernel thông dụng như trên, chúng ta cũng có thể tự định nghĩa các kernel của mình [như trong hướng dẫn này](#).

6.3 Ví dụ minh họa

6.3.1 Bài toán XOR

Chúng ta cùng quay lại với bài toán XOR. Chúng ta biết rằng [bài toán XOR không thể giải quyết nếu chỉ dùng một bộ phân lớp tuyến tính](#). Neurrel Network cần 2 layers để giải quyết

bài toán này. Với SVM, chúng ta có cách để chỉ cần sử dụng một bộ phân lớp. Dưới đây là ví dụ:

```

import numpy as np
import matplotlib.pyplot as plt
from sklearn import svm

# XOR dataset and targets
X = np.c_[(0, 0),
           (1, 1),
           #---
           (1, 0),
           (0, 1)].T
Y = [0] * 2 + [1] * 2
# figure number
fignum = 1

# fit the model
for kernel in ('sigmoid', 'poly', 'rbf'):
    clf = svm.SVC(kernel=kernel, gamma=4, coef0 = 0)
    clf.fit(X, Y)
    with PdfPages(kernel + '2.pdf') as pdf:
        # plot the line, the points, and the nearest vectors to the plane
        fig, ax = plt.subplots()
        plt.figure(fignum, figsize=(4, 3))
        plt.clf()

        plt.scatter(clf.support_vectors_[:, 0], clf.support_vectors_[:, 1], s=80,
                    facecolors='None')
        plt.plot(X[:2, 0], X[:2, 1], 'ro', markersize = 8)
        plt.plot(X[2:, 0], X[2:, 1], 'bs', markersize = 8)

        plt.axis('tight')
        x_min, x_max = -2, 3
        y_min, y_max = -2, 3

        XX, YY = np.mgrid[x_min:x_max:200j, y_min:y_max:200j]
        Z = clf.decision_function(np.c_[XX.ravel(), YY.ravel()])

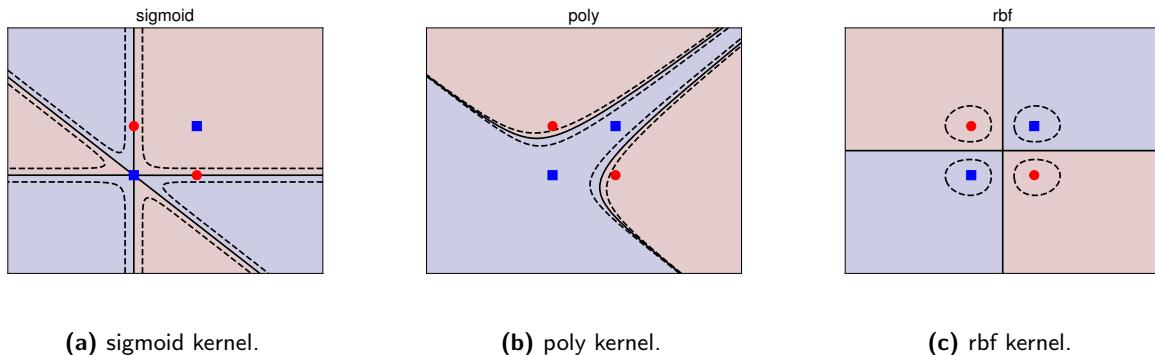
        # Put the result into a color plot
        Z = Z.reshape(XX.shape)
        plt.figure(fignum, figsize=(4, 3))
        CS = plt.contourf(XX, YY, np.sign(Z), 200, cmap='jet', alpha = .2)
        plt.contour(XX, YY, Z, colors=['k', 'k', 'k'], linestyles=['--', '--', '--'],
                    levels=[-.5, 0, .5])
        plt.title(kernel, fontsize = 15)
        plt.xlim(x_min, x_max)
        plt.ylim(y_min, y_max)

        plt.xticks(())
        plt.yticks(())
        fignum = fignum + 1
        pdf.savefig()

plt.show()

```

Kết quả được cho trong Hình 6.2.



Hình 6.2: Sử dụng kernel SVM để giải quyết bài toán XOR. a) sigmoid kernel. b) polynomial kernel. c) RBF kernel. Các đường nét liền là các đường phân lớp, ứng với giá trị của biểu thức (6.6) bằng 0. Các đường nét đứt là các đường đồng mức ứng với giá trị của biểu thức (6.6) bằng ± 0.5 . Trong ba phương pháp, RBF cho kết quả tốt nhất vì chúng cho kết quả đối xứng, hợp lý với dữ liệu bài toán.

Ta có các nhận xét đối với mỗi kernel như sau:

- **sigmoid:** nghiệm tìm được không thật tốt vì có 3 trong 4 điểm nằm chính xác trên đường phân chia. Nói cách khác, nghiệm này rất *nhạy cảm với nhiễu*.
- **poly:** Nghiệm này có tốt hơn nghiệm của **sigmoid** nhưng kết quả có phần giống với **overfitting**.
- **rbf:** Dữ liệu được tạo ra một cách đối xứng, đường phân lớp tìm được cũng tạo ra các vùng đối xứng với mỗi class. Nghiệm này được cho là *hợp lý hơn*. Trên thực tế, các **rbf** kernel được sử dụng nhiều nhất và cũng là lựa chọn mặc định trong hàm `sklearn.svm.SVC`.

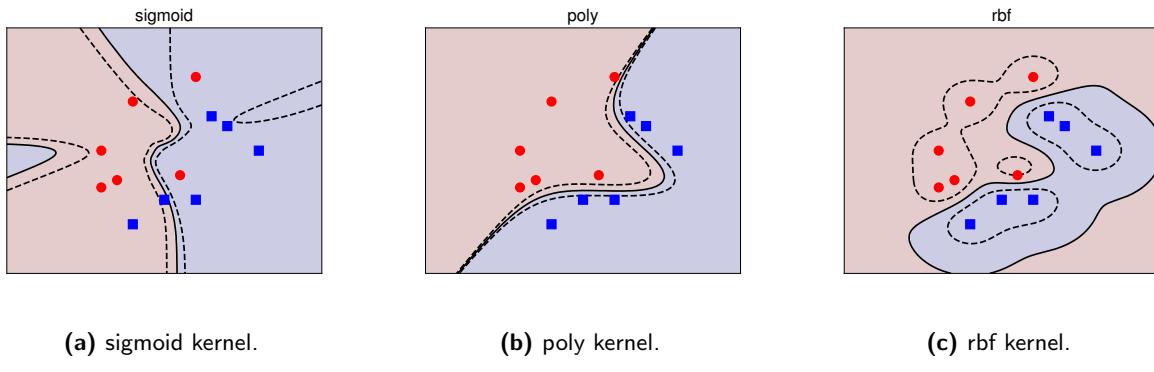
6.3.2 Dữ liệu gần phân biệt tuyến tính

Xét một ví dụ khác với dữ liệu giữa hai classes là *gần phân biệt tuyến tính* như Hình 6.3.

Trong ví dụ này, `kernel = 'poly'` cho kết quả tốt hơn `kernel = 'rbf'` vì trực quan cho ta thấy rằng nửa bên phải của mặt phẳng nên hoàn toàn thuộc vào class xanh. **sigmoid** kernel cho kết quả không thực sự tốt và ít được sử dụng.

6.3.3 Bài toán phân biệt giới tính

Bài toán này đã được đề cập ở Bài 12 với dữ liệu đầu vào là các ảnh khuôn mặt. Vì tôi không được phép phân phối cơ sở dữ liệu gốc này, tôi sẽ chia sẻ cho các bạn về dữ liệu đã qua xử lý, được lưu trong file `myARgender.mat`, có thể được [download tại đây](#). Dưới đây là ví dụ về cách sử dụng thư viện `sklearn.svm.SVC` để giải quyết bài toán:



Hình 6.3: Sử dụng kernel SVM để giải quyết bài toán với dữ liệu *gần phân biệt tuyến tính*. a) sigmoid kernel. b) polynomial kernel. c) RBF kernel. Các đường nét liền là các đường phân lớp, ứng với giá trị của biểu thức (6) bằng 0. Các đường nét đứt là các đường đồng mức ứng với giá trị của biểu thức (6) bằng ± 0.5 . Với bài toán này, polynomial kernel cho kết quả tốt hơn.

```

import scipy.io as sio
from sklearn.svm import SVC

A = sio.loadmat('myARgender.mat')
X_train = A['Y_train'].T
X_test = A['Y_test'].T
N = 700
y_train = A['label_train'].reshape(N)
y_test = A['label_test'].reshape(N)

clf = SVC(kernel='poly', degree = 3, gamma=1, C = 100)
clf.fit(X_train, y_train)
y_pred = clf.predict(X_test)
print("Accuracy: %.2f %%" %(100*accuracy_score(y_test, y_pred)))

```

Accuracy: 92.86 %

Kết quả không tệ! Các bạn thử thay các **kernel** và thiết lập các tham số khác xem kết quả thay đổi như thế nào. Vì dữ liệu giữa hai classes là *gần phân biệt tuyến tính* nên không có sự khác nhau nhiều giữa các kernel.

6.4 Tóm tắt

- Nếu dữ liệu của hai lớp là *không phân biệt tuyến tính*, chúng ta có thể tìm cách biến đổi dữ liệu sang một không gian mới sao cho trong không gian mới ấy, dữ liệu của hai lớp là *phân biệt tuyến tính* hoặc *gần phân biệt tuyến tính*.
- Việc tính toán trực tiếp hàm $\Phi()$ đòi hỏi phức tạp và tốn nhiều bộ nhớ. Thay vào đó, ta có thể sử dụng **kernel trick**. Trong cách tiếp cận này, ta chỉ cần tính tích vô hướng của hai vector bất kỳ trong không gian mới: $k(\mathbf{x}, \mathbf{z}) = \Phi(\mathbf{x})^T \Phi(\mathbf{z})$.

- Thông thường, các hàm $k()$ thỏa mãn điều kiện Mercer, và được gọi là *kernel*. Cách giải bài toán SVM với kernel hoàn toàn giống với cách giải bài toán Soft Margin SVM.
- Có 4 loại kernel thông dụng: **linear**, **poly**, **rbf**, **sigmoid**. Trong đó, **rbf** được sử dụng nhiều nhất và là lựa chọn mặc định trong các thư viện SVM.
- Với dữ liệu *gần phân biệt tuyến tính*, **linear** và **poly** kernels cho kết quả tốt hơn.
- [Source code](#).

6.5 Tài liệu tham khảo

- [1] Bishop, Christopher M. “[Pattern recognition and Machine Learning.](#)”, Springer (2006).
- [2] Duda, Richard O., Peter E. Hart, and David G. Stork. [Pattern classification](#). John Wiley & Sons, 2012.
- [3] [sklearn.svm.SVC](#)
- [4] [LIBSVM – A Library for Support Vector Machines](#)
- [5] Bennett, K. P. (1992). "Robust linear programming discrimination of two linearly separable sets". *Optimization Methods and Software* 1, 23–34.
- [6] Schölkopf, Bernhard, et al. "[New support vector algorithms.](#)" *Neural computation* 12.5 (2000): 1207-1245.
- [7] Rosasco, L.; De Vito, E. D.; Caponnetto, A.; Piana, M.; Verri, A. (2004). "[Are Loss Functions All the Same?](#)". *Neural Computation*. 16 (5): 1063–1076
- [8] [slearn Kernel functions](#)
- [9] [Kernel method](#)
- [10] <http://www.support-vector-machines.org/>

Multi-class Support Vector Machine

7.1 Giới thiệu

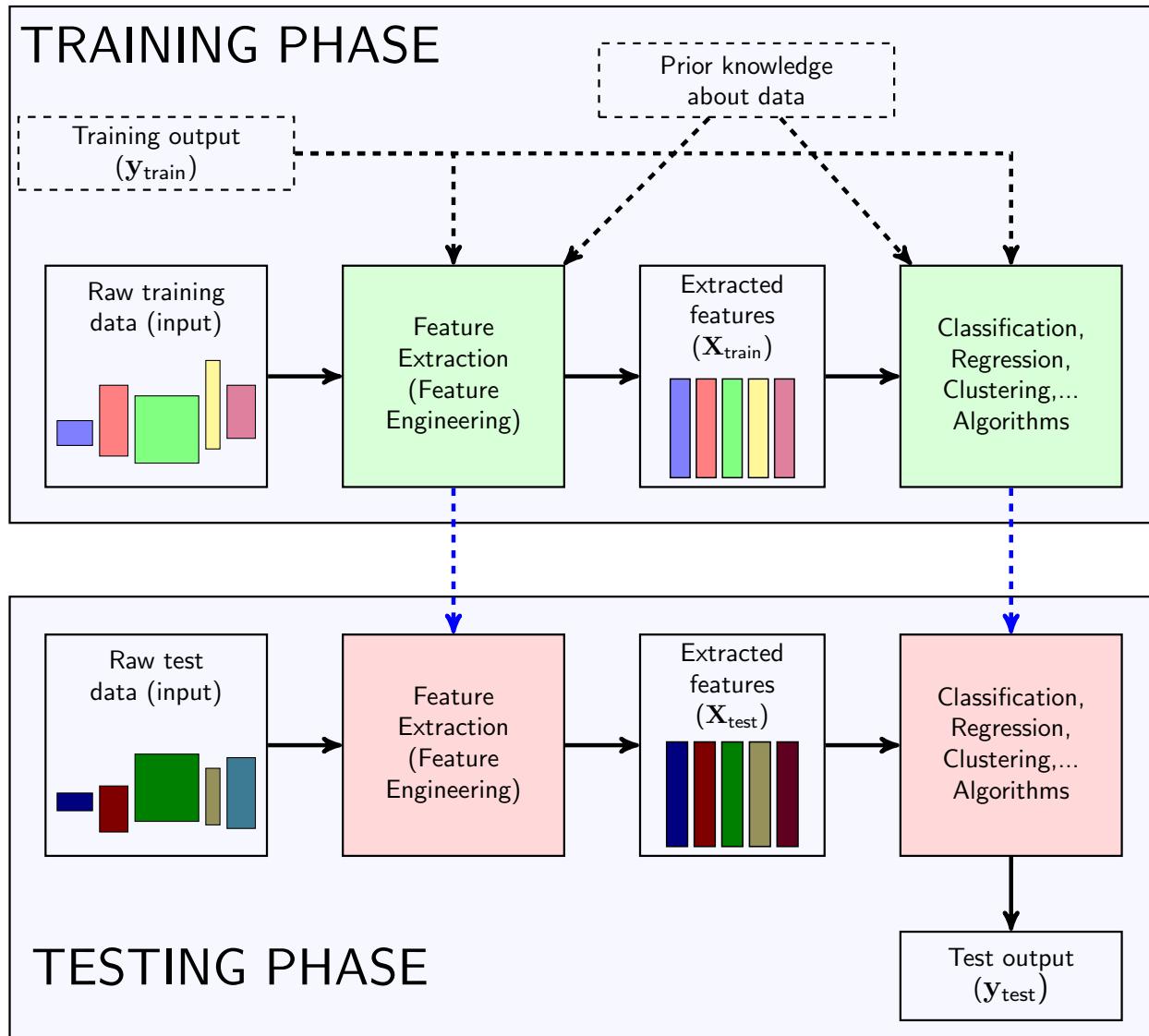
7.1.1 Từ Binary classification tới multi-class classification

Các phương pháp Support Vector Machine đã đề cập (Hard Margin, Soft Margin, Kernel) đều được xây dựng nhằm giải quyết bài toán [Binary Classification](#), tức bài toán phân lớp với chỉ hai classes. Việc này cũng tương tự như [Perceptron Learning Algorithm](#) hay [Logistic Regression](#) vậy. Các mô hình làm việc với bài toán có 2 classes còn được gọi là [Binary classifiers](#). Một cách tự nhiên để mở rộng các mô hình này áp dụng cho các bài toán multi-class classification, tức có nhiều classes dữ liệu khác nhau, là [sử dụng nhiều binary classifiers và các kỹ thuật như one-vs-one hoặc one-vs-rest](#). Cách làm này có những hạn chế như đã trình bày trong bài [Softmax Regression](#).

7.1.2 Mô hình end-to-end

Softmax Regression là mở rộng của Logistic Regression cho bài toán multi-class classification, có thể được coi là một layer của Neural Networks. Nhờ đó, Softmax Regression thường được sử dụng rất nhiều trong các bộ phân lớp hiện nay. Các bộ phân lớp cho kết quả cao nhất thường là một Neural Network với rất nhiều layers và layer cuối là một softmax regression, đặc biệt là các Convolutional Neural Networks. Các layer trước thường là kết hợp của các Convolutional layers và các nonlinear activation functions và pooling, các bạn tạm thời chưa cần quan tâm đến các layers phía trước này, tôi sẽ giới thiệu khi có dịp. Có thể coi các layer trước layer cuối là một công cụ giúp trích chọn đặc trưng của dữ liệu (Feature extraction), layer cuối là softmax regression, là một bộ phân lớp tuyến tính đơn giản nhưng rất hiệu quả. Bằng cách này, ta có thể coi là nhiều one-vs-rest classifiers được huấn luyện cùng nhau, hỗ trợ lẫn nhau, vì vậy, một cách tự nhiên, sẽ có thể tốt hơn là huấn luyện từng classifier riêng lẻ.

Sự hiệu quả của Softmax Regression nói riêng và Convolutional Neural Networks nói chung là cả *bộ trích chọn đặc trưng* (feature extractor) và *bộ phân lớp* (classifier) được *huấn luyện*



Hình 7.1: Mô hình chung cho các bài toán Machine Learning.

đồng thời. Điều này nghĩa là hai *bộ phận* này bổ trợ cho nhau trong quá trình huấn luyện. Classifier giúp tìm ra các hệ số hợp lý phù hợp với feature vector tìm được, ngược lại, feature extractor lại điều chỉnh các hệ số của các convolutional layer sao cho feature thu được là tuyến tính, phù hợp với classifier ở layer cuối cùng.

Tôi viết đến đây không phải là để giới thiệu về Softmax Regression, mà là đang nói chung đến các mô hình phân lớp *hiện đại*. Đặc điểm chung của chúng là feature extractor và classifier được huấn luyện một cách đồng thời. Những mô hình như thế này còn được gọi là *end-to-end*. Cùng xem lại mô hình chung cho các bài toán Machine Learning mà chúng ta đã thảo luận ở [Bài 11: Feature Engineering](#) (xem Hình 7.1).

Trong Hình 7.1, phần TRAINING PHASE, chúng ta có thể thấy rằng có hai khối chính là *Feature Extraction* và *Classification/Regression/Clustering...*. Các phương pháp *truyền thống*

thường xây dựng hai khối này qua các bước riêng rẽ. Phần Feature Extraction với dữ liệu ảnh có thể dùng các feature descriptor như **SIFT**, **SURF**, **HOG**; với dữ liệu văn bản thì có thể là **Bag of Words** hoặc **TF-IDF**. Nếu là các bài toán classification, phần còn lại có thể là SVM thông thường hay các bộ phân lớp *truyền thống* khác.

Với sự phát triển của Deep Learning trong những năm gần đây, người ta cho rằng các hệ thống *end-to-end* (từ đầu đến cuối) mang lại kết quả tốt hơn nhờ và việc các hai khối phía trên được huấn luyện cùng nhau, bổ trợ lẫn nhau. Thực tế cho thấy, các phương pháp *state-of-the-art* thường là các mô hình *end-to-end*.

Support Vector Machine được chứng minh là nhìn chung tốt hơn Logistic Regression vì chúng có quan tâm đến việc tạo *margin* lớn nhất giữa các classes. Câu hỏi đặt ra là:

Liệu có cách nào giúp kết hợp SVM với Neural Networks để tạo ra một bộ phân lớp tốt với bài toán multi-class classification? Hơn nữa, toàn bộ hệ thống có thể được huấn luyện theo kiểu *end-to-end*?

Câu trả lời sẽ được tìm thấy trong bài viết này, bằng một phương pháp được gọi là *Multi-class Support Vector Machine*.

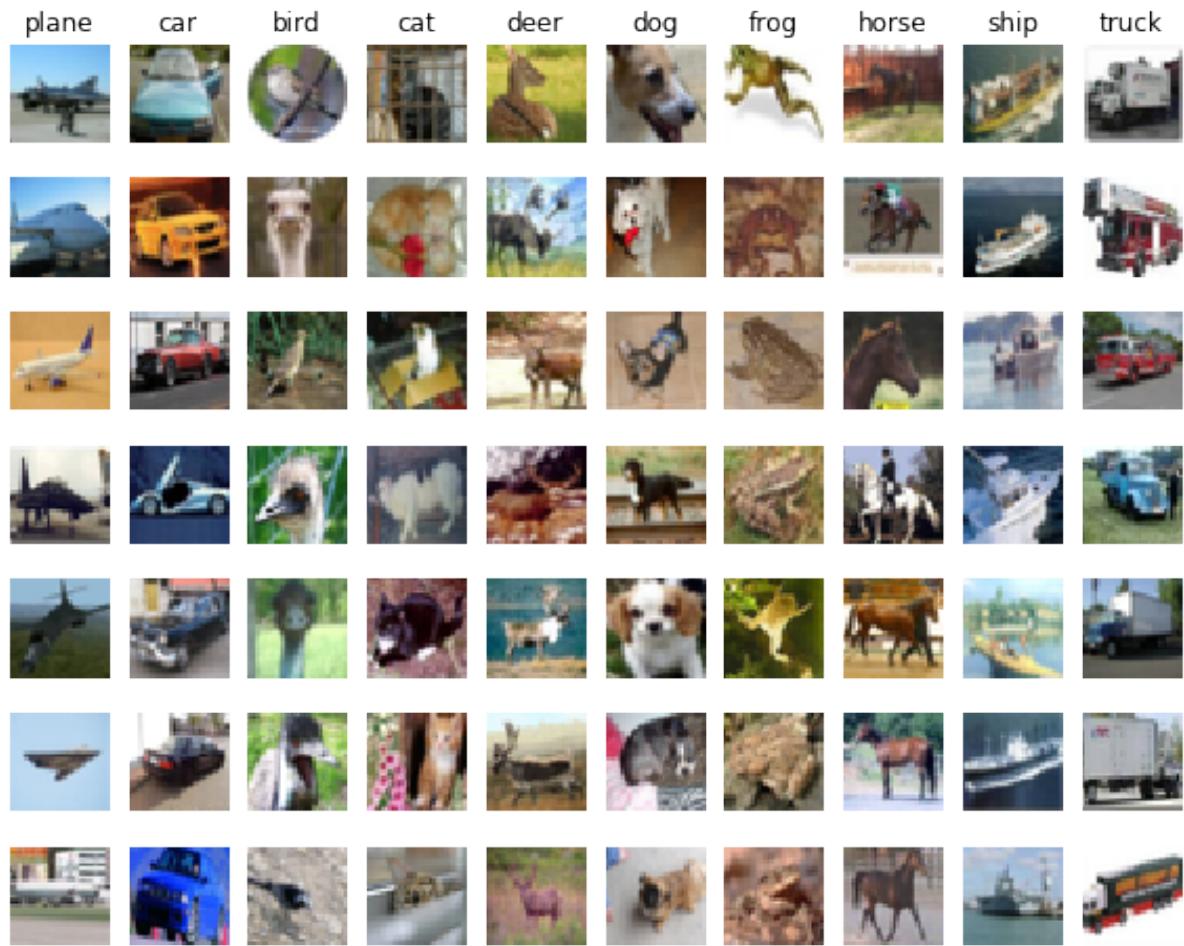
Và để cho bài viết hấp dẫn hơn, tôi xin giới thiệu luôn, ở phần cuối, chúng ta sẽ cùng lập trình từ đầu đến cuối để giải quyết bài toán phân lớp với bộ cơ sở dữ liệu nổi tiếng: CIFAR10.

7.1.3 Bộ cơ sở dữ liệu CIFAR10

Bộ cơ sở dữ liệu CIFAR10 gồm 51000 ảnh khác nhau thuộc 10 classes: *plane, car, bird, cat, deer, dog, frog, horse, ship, và truck*. Mỗi bức ảnh có kích thước 32×32 pixel. Một vài ví dụ cho mỗi class được cho trong Hình 7.2. 50000 ảnh được sử dụng cho training, 1000 ảnh còn lại được dùng cho test. Trong số 50000 ảnh training, 1000 ảnh sẽ được lấy ra ngẫu nhiên để làm **validation set**.

Đây là một bộ cơ sở dữ liệu tương đối khó vì ảnh nhỏ và object trong cùng một class cũng biến đổi rất nhiều về màu sắc, hình dáng, kích thước. **Thuật toán tốt nhất hiện nay cho bài toán này** đã đạt được độ chính xác trên 90%, sử dụng một Convolutional Neural Network nhiều lớp kết hợp với Softmax regression ở layer cuối cùng. Trong bài này, chúng ta sẽ sử dụng một mô hình neural network đơn giản không có hidden layer nào để giải quyết, kết quả đạt được là khoảng 40%, nhưng cũng là đã rất ấn tượng. Layer cuối là một layer Multi-class SVM. Tôi sẽ hướng dẫn các bạn lập trình cho mô hình này *từ đầu đến cuối* mà không sử dụng một thư viện đặc biệt nào ngoài numpy.

Bài toán này cũng như nội dung chính của bài viết được lấy từ Lecture notes: **Linear Classifier II** và **Assignment #1** trong khoá học **CS231n: Convolutional Neural Networks for Visual Recognition** kỳ Winter 2016 của Stanford.



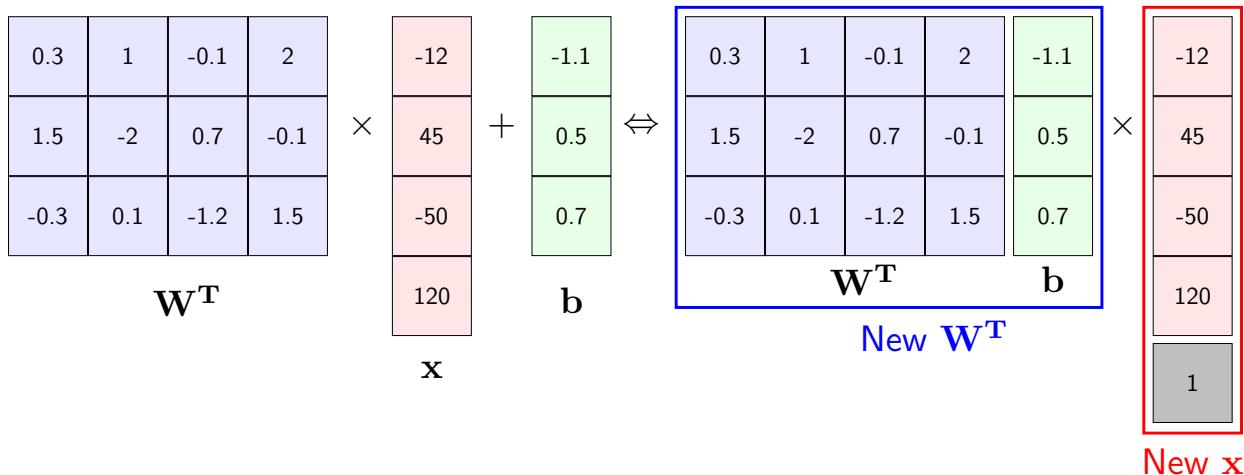
Hình 7.2: Ví dụ về các bức ảnh trong 10 classes trong bộ dữ liệu CIFAR10.

Trước khi đi vào mục xây dựng hàm măt măt cho Multi-class SVM, tôi muôn nhăc lại một chút về một chút *feature engineering* cho ảnh trong CIFAR-10 và **bias trick** nói chung trong Neural Networks.

7.1.4 Image data preprocessing

Để cho mọi thứ được đơn giản và có được một mô hình hoàn chỉnh, chúng ta sẽ sử dụng phương pháp *feature engineering* đơn giản nhất: lấy trực tiếp tất cả các pixel trong mỗi ảnh và thêm một chút normalization.

- Mỗi ảnh của CIFAR-10 đã có kích thước giống nhau 32×32 pixel, vì vậy việc đầu tiên chúng ta cần làm là kéo dài mỗi trong ba channels Red, Green, Blue của bức ảnh ra thành một vector có kích thước là $3 \times 32 \times 32 = 3072$.

**Hình 7.3:** Bias trick.

- Vì mỗi pixel có giá trị là một số tự nhiên từ 0 đến 255 nên chúng ta cần một chút **chuẩn hóa dữ liệu**. Trong Machine Learning, một cách đơn giản nhất để chuẩn hóa dữ liệu là **center data**, tức là làm cho mỗi feature có trung bình cộng bằng 0. Một cách đơn giản để làm việc này là ta tính trung bình cộng của tất cả các ảnh trong tập training để được *ảnh trung bình*, sau đó trừ từ tất cả các ảnh đi *ảnh trung bình* này. Tương tự, ta cũng dùng *ảnh trung bình* này để chuẩn hóa dữ liệu trong *validation set* và *test set*.

7.1.5 Bias trick

Thông thường, với một ma trận hệ số $\mathbf{W} \in \mathbb{R}^{d \times C}$, một đầu vào $\mathbf{x} \in \mathbb{R}^d$ và vector bias $\mathbf{b} \in \mathbb{R}^C$, chúng ta có thể tính được đầu ra của layer này là:

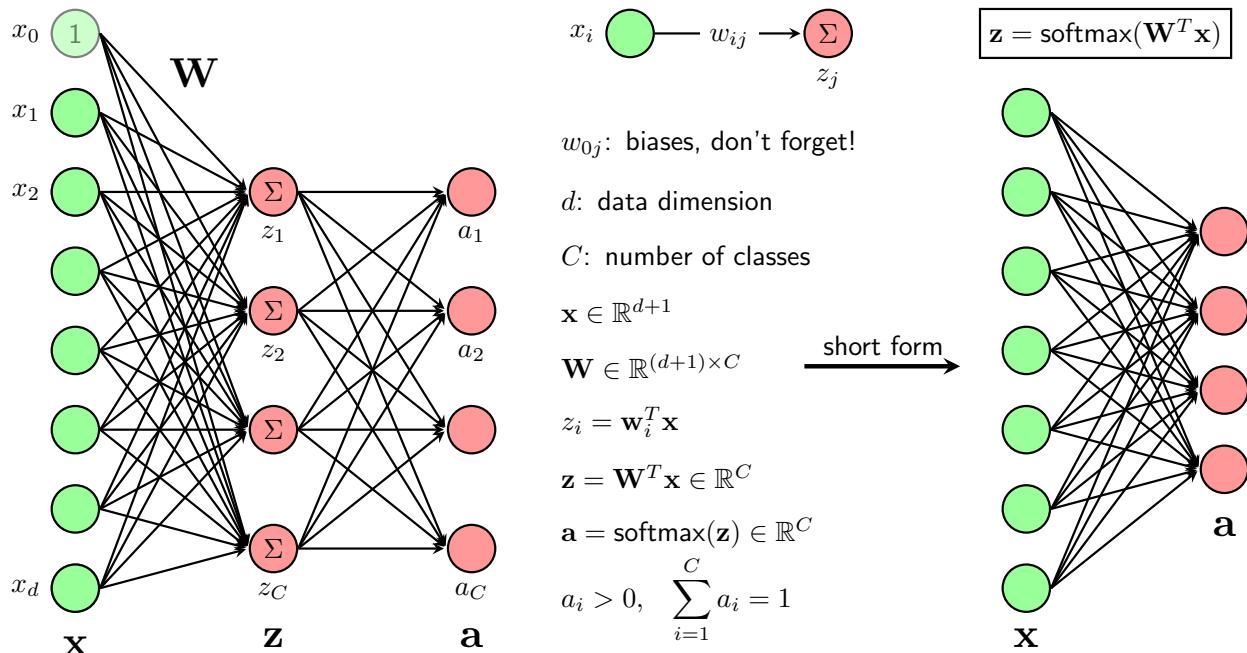
$$f(\mathbf{x}, \mathbf{W}, \mathbf{b}) = \mathbf{W}^T \mathbf{x} + \mathbf{b} \quad (7.1)$$

Để cho biểu thức trên đơn giản hơn, ta có thể thêm một phần tử bằng 1 vào cuối của \mathbf{x} và *ghép* vector \mathbf{b} vào ma trận \mathbf{W} như ví dụ trong [Hình 7.3](#).

Bây giờ thì ta chỉ còn một biến dữ liệu là \mathbf{W} thay vì hai biến dữ liệu như trước. Từ giờ trở đi, khi viết \mathbf{W} và \mathbf{x} , chúng ta ngầm hiểu là biến mới và dữ liệu mới như ở phần bên phải của [Hình 7.3](#).

7.2 Xây dựng hàm mất mát cho Multi-class Support Vector Machine

Chúng ta cùng quay lại một chút với ý tưởng của Softmax Regression với hàm mất mát Cross-entropy. Sau đó, chúng ta sẽ làm quen với Multi-class SVM với hàm mất mát hinge loss mở rộng.



Hình 7.4: Mô hình Softmax Regression dưới dạng Neural Network.

7.2.1 Nhắc lại Softmax Regression.

Chúng ta cùng xem lại [Softmax layer đã được trình bày trong Bài 13](#).

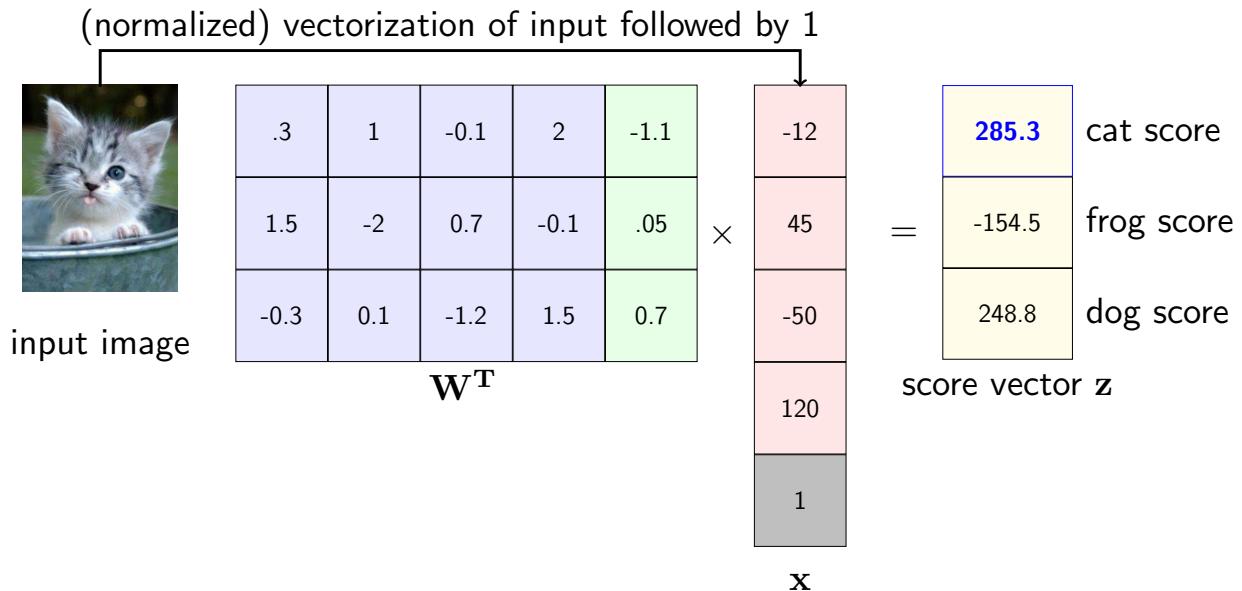
Trong Hình 4 ở trên, dữ liệu trong lớp màu xanh lục được coi như *feature vector* của dữ liệu. Với dữ liệu CIFAR-10, nếu ta coi mỗi feature là giá trị của từng pixel trong ảnh, tổng số chiều của *feature vector* cho mỗi bức ảnh là $32 \times 32 \times 3 + 1 = 3073$, với 3 là số channels trong bức ảnh (Red, Green, Blue).

Qua ma trận hệ số \mathbf{W} , dữ liệu ban đầu trở thành $\mathbf{z} = \mathbf{W}^T \mathbf{x}$.

Lúc này, ứng với mỗi một trong C classes, chúng ta nhận được một giá trị tương ứng z_i ứng với class thứ i . Giá trị z_i này còn được gọi là *score* của dữ liệu \mathbf{x} ứng với class thứ i .

Ý tưởng chính trong Softmax Regression là đi tìm ma trận hệ số \mathbf{W} , mỗi cột của ma trận này ứng với một class, sao cho *score vector* \mathbf{z} đạt giá trị lớn nhất tại phần tử tương ứng với class chính xác của nó. Sau khi mô hình đã được *trained*, *nhận* của một điểm dữ liệu mới được tính là vị trí của thành phần score có giá trị lớn nhất trong *score vector*. Xem ví dụ trong Hình 5 dưới đây:

Để huấn luyện trên tập các cặp (*dữ liệu, nhãn*), Softmax Regression sử dụng hàm softmax để đưa *score vector* về dạng phân phối xác suất có các phần tử là dương và có tổng bằng 1. Sau đó dùng hàm cross entropy để ép vector xác suất này gần với vector xác suất *thật sự* của dữ liệu - tức one-hot vector mà chỉ có đúng 1 phần tử bằng 1 tại class tương ứng, các phần tử còn lại bằng 0.



Hình 7.5: Ví dụ về cách tính score vector. Khi test, nhãn của dữ liệu được xác định dựa trên class có score cao nhất.

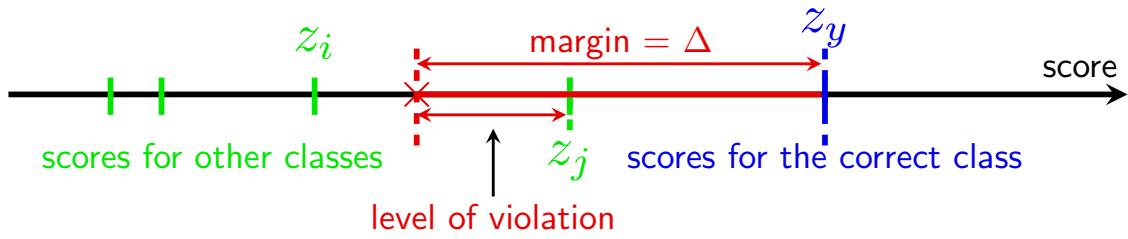
7.2.2 Hinge lossss tổng quát cho Multi-class SVM

Với Multi-class SVM, trong khi test, class của một input cũng được xác định bởi thành phần có giá trị lớn nhất trong score vector. Điều này giống với Softmax Regression.

Softmax Regression sử dụng cross-entropy để ép hai vector xác suất bằng nhau, tức ép phần tử tương ứng với *correct class* trong vector xác suất gần với 1, đồng thời, các phần tử còn lại trong vector đó gần với 0. Nói cách khác, cách làm này khiến cho phần tử tương ứng với *correct class* càng lớn hơn các phần tử còn lại càng tốt. Trong khi đó, Multi-class SVM sử dụng một chiến thuật khác cho mục đích tương tự dựa trên *score vector*. Điểm khác biệt là Multi-class SVM xây dựng hàm mất mát dựa trên định nghĩa của *bien an toàn*, giống như trong Hard/Soft Margin vậy. Multi-class SVM muốn thành phần ứng với *correct class* của *score vector* lớn hơn các phần tử khác, không những thế, nó còn lớn hơn một đại lượng $\Delta > 0$ gọi là *bien an toàn*. Hãy xem Hình 6 dưới đây:

Với cách xác định biên như trên, Multi-class SVM sẽ *cho qua* những scores nằm về phía trước vùng màu đỏ. Những điểm có scores nằm phía phải của *ngưỡng* (chữ x màu đỏ) sẽ bị *xử phạt*, và càng vi phạm nhiều sẽ càng bị xử lý ở mức cao.

Để mô tả các mức vi phạm này dưới dạng toán học, trước hết ta giả sử rằng các thành phần của score vector được đánh số thứ tự từ 1. Các classes cũng được đánh số thứ tự từ 1. Giả sử rằng điểm dữ liệu \mathbf{x} đang xét thuộc class y và score vector của nó là vector \mathbf{z} . Thế thì score của *correct class* là z_y , scores của các classes khác là các $z_i, i \neq y$. Xét ví dụ như trong Hình 6 với hai score z_i trong vùng an toàn và z_j trong vùng vi phạm.



Hình 7.6: Mô tả hinge loss cho Multi-class Support Vector Machine. Multi-class SVM *muốn* score của *correct class*, được minh họa bởi điểm màu lam, cao hơn các scores khác, minh họa bởi các điểm màu lục, một khoảng cách an toàn Δ là đoạn màu đỏ. Những scores khác nằm trong vùng an toàn (phía trái của điểm x màu đỏ) sẽ không gây ra mất mát gì, những scores nằm trong hoặc bên phải vùng màu đỏ đã *vi phạm* quy tắc và cần được *xử phạt*.

- Với mỗi score z_i trong vùng an toàn, *loss* bằng 0.
- Với mỗi score z_j vượt quá điểm an toàn (điểm x đỏ), *loss* do nó gây ra được tính bằng lượng vượt quá so với điểm x đỏ, đại lượng này có thể tính được là: $z_j - (z_y - \Delta) = \Delta - z_y + z_j$.

Tóm lại, với một score $z_j, j \neq y$, *loss* do nó gây ra có thể được viết gọn thành:

$$\max(0, \Delta - z_y + z_j) = \max(0, \Delta - \mathbf{w}_y^T \mathbf{x} + \mathbf{w}_j^T \mathbf{x}) \quad (7.2)$$

trong đó \mathbf{w}_j là *cột* thứ j của ma trận hệ số \mathbf{W} .

Như vậy, với một điểm dữ liệu $\mathbf{x}_n, n = 1, 2, \dots, N$, tổng cộng *loss* do nó gây ra là:

$$\mathcal{L}_n = \sum_{j \neq y_n} \max(0, \Delta - z_{y_n}^n + z_j^n)$$

trong đó $\mathbf{z}^n = \mathbf{w}^T \mathbf{x}_n = [z_1^n, z_2^n, \dots, z_C^n]^T \in \mathbb{R}^{C \times 1}$ là scores tương ứng với điểm dữ liệu \mathbf{x}_n ; y_n là *correct class* của điểm dữ liệu đó.

Với toàn bộ các điểm dữ liệu $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_N]$, *loss* tổng cộng là:

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, \Delta - z_{y_n}^n + z_j^n) \quad (7.3)$$

với $\mathbf{y} = [y_1, y_2, \dots, y_N]$ là vector chứa *corect class* của toàn bộ các điểm dữ liệu trong *training set*. Hệ số $\frac{1}{N}$ tính trung bình của *loss* để tránh việc biểu thức này quá lớn gây tràn số máy tính.

Có một *bug* trong lỗi này, chúng ta cùng phân tích tiếp.

7.2.3 Regularization

Điều gì sẽ xảy ra nếu nghiệm tìm được \mathbf{w} là *hoàn hảo*, tức không có score nào *vi phạm* và biểu thức (2) đạt giá trị bằng 0? Nói cách khác:

$$\Delta - z_{y_n}^n + z_j^n = \leq 0 \Leftrightarrow \Delta \leq \mathbf{w}_{y_n}^T \mathbf{x}_n - \mathbf{w}_j^T \mathbf{x}_n \quad \forall n = 1, 2, \dots, N; j = 1, 2, \dots, C; j \neq y_n$$

Điều này có nghĩa là $k\mathbf{W}$ cũng là một nghiệm của bài toán với $k > 1$ bất kỳ. Việc bài toán có vô số nghiệm và có những nghiệm có những phần tử tiên tới vô cùng khiến cho bài toán rất *unstable* khi giải. Một phương pháp quen thuộc để tránh hiện tượng này là cộng thêm số hạng *regularization* vào hàm mất mát. Số hạng này giúp *ngăn chặn* việc các hệ số của \mathbf{W} trở nên quá lớn. Và để cho hàm mất mát vẫn có đạo hàm đơn giản, chúng ta lại sử dụng l_2 regularization:

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \underbrace{\frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, \Delta - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n)}_{\text{data loss}} + \underbrace{\frac{\lambda}{2} \|\mathbf{W}\|_F^2}_{\text{regularization loss}} \quad (7.4)$$

với $\|\bullet\|_F$ là *Frobenius norm*, và λ là một giá trị dương giúp cân bằng giữa *data loss* và *regularization loss*, thường được chọn bằng *cross-validation*.

7.2.4 Chọn giá trị Δ

Có hai *hyperparameter* trong hàm mất mát (7.4) là Δ và λ , câu hỏi đặt ra là làm thế nào để chọn ra cặp giá trị hợp lý nhất cho từng bài toán. Liệu chúng ta có cần làm *cross-validation* cho từng giá trị không?

Trong thực tế, người ta nhận thấy rằng Δ có thể được chọn bằng 1 mà không ảnh hưởng nhiều tới chất lượng của nghiệm. Thực tế cho thấy cả hai tham số Δ và λ đều giúp cân bằng giữa *data loss* và *regularization loss*. Thực vậy, độ lớn của các hệ số trong \mathbf{W} có tác động trực tiếp lên các *score vectors*, và vì vậy ảnh hưởng tới sự khác nhau giữa chúng. Khi chúng ta giảm các hệ số của \mathbf{W} , sự khác nhau giữa các scores cũng giảm một tỉ lệ tương tự; và khi ta tăng các hệ số của \mathbf{W} , sự khác nhau giữa các scores cũng tăng lên. Bởi vậy, giá trị chính xác Δ của *margin* giữa các scores trở nên không quan trọng vì chúng ta có thể tăng hoặc giảm \mathbf{W} một cách tùy ý. Việc quan trọng hơn là hạn chế việc \mathbf{W} trở nên quá lớn. Việc này đã được điều chỉnh bởi tham số λ .

Cuối cùng, chúng ta sẽ đi tối ưu hàm mất mát sau đây cho Multi-class SVM:

$$\mathcal{L}(\mathbf{X}, \mathbf{y}, \mathbf{W}) = \frac{1}{N} \sum_{n=1}^N \sum_{j \neq y_n} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) + \frac{\lambda}{2} \|\mathbf{W}\|_F^2 \quad (7.5)$$

Một lần nữa, chúng ta có thể dùng [Gradient Descent](#) để tối ưu bài toán tối ưu không ràng buộc này. Chúng ta sẽ đi sâu vào việc tính đạo hàm của hàm mất mát *một cách hiệu quả* ở mục 3.

Trước hết, có một nhận xét thú vị:

7.2.5 Soft Margin SVM là một trường hợp đặc biệt của Multi-class SVM

Phát biểu này có vẻ hiển nhiên vì bài toán phân lớp với hai classes là một trường hợp đặc biệt của bài toán phân lớp với nhiều classes! Nhưng điều tôi muốn nói đến là cách xây dựng hàm mất mát. Điều này có thể được nhận ra bằng cách xét từng điểm dữ liệu.

Trong (7.5), nếu $C = 2$ (số classes bằng 2), hàm mất mát tại mỗi điểm dữ liệu trở thành (tạm bỏ qua *regularization loss*):

$$\mathcal{L}_n = \sum_{j \neq y_n} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) \quad (7.6)$$

Xét hai trường hợp:

- $y_n = 1 \Rightarrow \mathcal{L}_n = \max(0, 1 - \mathbf{w}_1^T \mathbf{x}_n + \mathbf{w}_2^T \mathbf{x}_n) = \max(0, 1 - (1)(\mathbf{w}_1 - \mathbf{w}_2)^T \mathbf{x})$
- $y_n = 2 \Rightarrow \mathcal{L}_n = \max(0, 1 - \mathbf{w}_2^T \mathbf{x}_n + \mathbf{w}_1^T \mathbf{x}_n) = \max(0, 1 - (-1)(\mathbf{w}_1 - \mathbf{w}_2)^T \mathbf{x})$

Nếu ta thay $y_n = -1$ cho dữ liệu thuộc class 2, và đặt $\bar{\mathbf{w}} = \mathbf{w}_1 - \mathbf{w}_2$, hai trường hợp trên có thể được viết gọn thành:

$$\mathcal{L}_n = \max(0, 1 - y_n \bar{\mathbf{w}}^T \mathbf{x}_n)$$

tức chính là Hinge loss cho Soft Margin SVM.

7.3 Tính toán hàm mất mát và đạo hàm của nó

Để tối ưu hàm mất mát, chúng ta sử dụng phương pháp Stochastic Gradient Method. Điều này có nghĩa là chúng ta cần tính gradient tại mỗi vòng lặp. Đồng thời, *loss* sau mỗi vòng lặp cũng cần được tính để kiểm tra liệu thuật toán có hoạt động như ý muốn hay không.

Việc tính toán *loss* và *gradient* này không những cần phải chính xác mà còn cần được thực hiện càng nhanh càng tốt. Trong khi việc tính *loss* thường dễ thực hiện, việc tính *gradient* cần phải được kiểm tra kỹ càng hơn.

Để đảm bảo rằng *loss* và *gradient* được tính một cách chính xác và nhanh, chúng ta sẽ làm từng bước một. Bước thứ nhất là đảm bảo rằng các tính toán là chính xác, dù cách tính có rất chậm. Bước thứ hai là phải đảm bảo có cách tính hiệu quả để thuật toán chạy nhanh

hơn. Hai bước này cần được thực hiện trên một lượng dữ liệu nhỏ để đảm bảo chúng được tính chính xác trước khi áp dụng thuật toán vào dữ liệu thật, thường có số điểm dữ liệu lớn và mỗi điểm dữ liệu cũng có số chiều lớn.

Hai mục nhỏ tiếp theo sẽ mô tả hai bước đã nêu ở trên.

7.3.1 Tính hàm mất mát và đạo hàm của nó bằng cách *naive*

Naive dịch tạm ra tiếng Việt có nghĩa là *ngây thơ*, hoặc *ngây ngô*. Trong Machine Learning, từ này cũng hay được sử dụng với ý chỉ sự đơn giản.

Dưới đây là cách tính đơn giản *loss* và *gradient* của hàm mất mát trong (7.5). Chú ý thành phần *regularization*.

```
import numpy as np
from random import shuffle

# naive way to calculate loss and grad
def svm_loss_naive(W, X, y, reg):
    d, C = W.shape
    _, N = X.shape

    ## naive loss and grad
    loss = 0
    dW = np.zeros_like(W)
    for n in xrange(N):
        xn = X[:, n]
        score = W.T.dot(xn)
        for j in xrange(C):
            if j == y[n]:
                continue
            margin = 1 - score[y[n]] + score[j]
            if margin > 0:
                loss += margin
                dW[:, j] += xn
                dW[:, y[n]] -= xn

        loss /= N
        loss += 0.5*reg*np.sum(W * W) # regularization

    dW /= N
    dW += reg*W # gradient off regularization
    return loss, dW

# random, small data
N, C, d = 10, 3, 5
reg = .1
W = np.random.randn(d, C)
X = np.random.randn(d, N)
y = np.random.randint(C, size = N)

# sanity check
print 'loss without regularization:', svm_loss_naive(W, X, y, 0)[0]
print 'loss with regularization:', svm_loss_naive(W, X, y, .1)[0]
```

loss without regularization: 4.68441457903 loss with regularization: 6.25136675351

Cách tính với hai vòng for lồng nhau như trên mô tả lại chính xác *loss* trong (7.5) nên sai sót, nếu có, ở đây có thể được kiểm tra và sửa lại dễ dàng. Việc kiểm tra ở cuối cho cái nhìn ban đầu về hàm mất mát: dương và không có *regularization* sẽ có *loss* tổng cộng nhỏ hơn.

Về cách tính *gradient* cho phần *data loss*, mặc dù **hàm max là convex** nhưng nó không có đạo hàm tại mọi nơi. Cụ thể:

$$\frac{\partial}{\partial \mathbf{w}_{y_n}} \max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) = \begin{cases} 0 & \text{if } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n < 0 \\ -\mathbf{x}_n & \text{if } 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n > 0 \end{cases} \quad (7.7)$$

$$\frac{\partial}{\partial \mathbf{w}_j} \max(0, 1 - \mathbf{w}_j^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n) = \begin{cases} 0 & \text{if } 1 - \mathbf{w}_j^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n < 0 \\ \mathbf{x}_n & \text{if } 1 - \mathbf{w}_j^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n > 0 \end{cases} \quad (7.8)$$

Rõ ràng là các đạo hàm này không xác định tại các điểm mà $1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n = 0$. Tuy nhiên, khi thực hành, ta có thể giả sử rằng tại 0, các đạo hàm này cũng bằng 0.

Dể kiểm tra lại cách tính đạo hàm như trên dựa vào (7.7) và (7.8) có chính xác không, chúng ta sẽ làm một bước quen thuộc là so sánh nó với *numerical gradient*. Nếu sự sai khác là nhỏ, nhỏ hơn **1e-7** thì ta có thể coi là *gradient* tính được là chính xác:

```
f = lambda W: svm_loss_naive(W, X, y, .1)[0]

# for checking if calculated grad is correct
def numerical_grad_general(W, f):
    eps = 1e-6
    g = np.zeros_like(W)
    # flattening variable -> 1d. Then we need
    # only one for loop
    W_flattened = W.flatten()
    g_flattened = np.zeros_like(W_flattened)

    for i in xrange(W.size):
        W_p = W_flattened.copy()
        W_n = W_flattened.copy()
        W_p[i] += eps
        W_n[i] -= eps

        # back to shape of W
        W_p = W_p.reshape(W.shape)
        W_n = W_n.reshape(W.shape)
        g_flattened[i] = (f(W_p) - f(W_n)) / (2*eps)

    # convert back to original shape
    return g_flattened.reshape(W.shape)

# compare two ways of computing gradient
g1 = svm_loss_naive(W, X, y, .1)[1]
g2 = numerical_grad_general(W, f)
print 'gradient difference: %f' % np.linalg.norm(g1 - g2)
# this should be very small
```

$$\begin{array}{c}
 \mathbf{Z} = \mathbf{W}^T \mathbf{X} \\
 \left[\begin{array}{c} \mathbf{w}_1^T \\ \mathbf{w}_2^T \\ \mathbf{w}_3^T \\ \mathbf{w}_4^T \end{array} \right] \left[\begin{array}{c} \mathbf{x}_1 \mathbf{x}_2 \mathbf{x}_3 \end{array} \right] = \left[\begin{array}{ccc} 2 & 0.1 & -0.2 \\ 1.5 & 1.5 & 2.5 \\ -0.2 & 2.5 & 3.0 \\ 1.7 & 1.8 & 1.0 \end{array} \right] \xrightarrow{\max(0, 1 - z_{y_n}^n + z_j^n)} \left[\begin{array}{ccc} 0 & 0 & 0 \\ 0.5 & 0 & 0 \\ 0 & 0 & 1.5 \\ 0.7 & 0.3 & 0 \end{array} \right] \xrightarrow{\mathcal{L}_{\text{data}} = 0.5 + 0.7 + 0.3 + 1.5 = 3.0} \left[\begin{array}{ccc} -2 & 0 & 0 \\ 1 & 0 & -1 \\ 0 & -1 & 1 \\ 1 & 1 & 0 \end{array} \right] \rightarrow \frac{\partial \mathcal{L}_{\text{data}}}{\partial \mathbf{w}_1} = -2\mathbf{x}_1 \\
 \mathbf{y} = [1, 3, 2] \quad \rightarrow \frac{\partial \mathcal{L}_{\text{data}}}{\partial \mathbf{w}_2} = \mathbf{x}_1 - \mathbf{x}_3 \quad \rightarrow \frac{\partial \mathcal{L}_{\text{data}}}{\partial \mathbf{w}_3} = -\mathbf{x}_2 + \mathbf{x}_3 \quad \rightarrow \frac{\partial \mathcal{L}_{\text{data}}}{\partial \mathbf{w}_4} = \mathbf{x}_1 + \mathbf{x}_2
 \end{array}$$

Hình 7.7: Mô phỏng cách tính *loss* và *gradient* của Multi-class SVM.

Kết quả:

```
gradient difference: 0.000000
```

Sự sai khác là xấp xỉ 0, vậy chúng ta có thể yên tâm khi nói rằng cách tính *gradient* đã thỏa mãn sự *chính xác*, chúng ta cần tính nó một cách *hiệu quả* nữa.

Các cách tính hiệu quả thường không chứa các vòng **for** mà được viết gọn lại dưới dạng ma trận và vector, việc này đòi hỏi các kỹ năng về Đại số tuyến tính và **numpy** một chút. Cách tính này thường được gọi là *vectorized*.

7.3.2 Tính hàm mất mát và đạo hàm của nó bằng cách *vectorized*

Dễ dẽ hình dung, chúng ta cùng quan sát Hình 7.7.

Ở đây, chúng ta tạm quên phần *regularization loss* đi vì cả *loss* và *gradient* của phần này đều có cách tính đơn giản. Với phần *data loss*, chúng ta cũng bỏ qua hệ số $\frac{1}{N}$ đi cho dễ hình dung.

Giả sử rằng có bốn classes và mini-batch gồm có ba điểm dữ liệu $\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3$. Ba điểm này lần lượt thuộc vào các class 1, 3, 2. Các ô có nền màu đỏ nhạt ở mỗi cột tương ứng với *correct class* của điểm dữ liệu của cột đó. Các bước tính *loss* và *gradient* có thể được hình dung như sau:

- **Bước 1:** Tính *score matrix* $\mathbf{Z} = \mathbf{W}^T \mathbf{X}$.
- **Bước 2:** Với mỗi ô, tính $\max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \mathbf{w}_j^T \mathbf{x}_n)$. Chú ý rằng ta không cần tính các ô có nền màu đỏ nhạt vì có thể coi chúng bằng 0 do trong biểu thức *data loss*. Sau khi tính được giá trị của từng ô, ta chỉ quan tâm tới các ô có giá trị lớn hơn 0 - là các ô được tô nền màu xanh lục. Lấy tổng của tất cả các phần tử ở các ô xanh lục, ta sẽ được *data loss*. (*Có thể bạn sẽ phải dừng lại một chút để hiểu. Không sao, take your time*).

- **Bước 3:** Theo công thức (7.8), với ô màu xanh lục ở hàng 2, cột 1, thì đạo hàm theo vector hệ số \mathbf{w}_2 sẽ được cộng thêm một lượng \mathbf{x}_1 và đạo hàm theo vector hệ số \mathbf{w}_1 sẽ được trừ đi một lượng \mathbf{x}_1 . Tương tự với các ô màu xanh lục còn lại. Với các ô màu đỏ ở hàng 1 cột 1, chúng ta chú ý rằng trong cùng cột 1, có bao nhiêu ô màu xanh lục thì có bấy nhiêu lần đạo hàm của \mathbf{w}_1 bị trừ đi một lượng \mathbf{x}_1 . Điều này được suy ra từ (7.7). Từ đó suy ra trong khối ô vuông thứ ba, giá trị của ô màu đỏ sẽ bằng đối số của tổng số lượng các ô màu xanh lục. Vậy nên ô màu đỏ ở hàng 1 cột 1 phải bằng -2.
- **Bước 4:** Bây giờ cộng theo các hàng, ta sẽ được đạo hàm theo hệ số của class tương ứng.

Trong đoạn code dưới đây, `correct_class_score` chính là tập hợp các giá trị trong các ô màu đỏ ở khối thứ nhất.

```
# more efficient way to compute loss and grad
def svm_loss_vectorized(W, X, y, reg):
    d, C = W.shape
    _, N = X.shape
    loss = 0
    dW = np.zeros_like(W)

    Z = W.T.dot(X)

    correct_class_score = np.choose(y, Z).reshape(N, 1).T
    margins = np.maximum(0, Z - correct_class_score + 1)
    margins[y, np.arange(margins.shape[1])] = 0
    loss = np.sum(margins, axis = (0, 1))
    loss /= N
    loss += 0.5 * reg * np.sum(W * W)

    F = (margins > 0).astype(int)
    F[y, np.arange(F.shape[1])] = np.sum(~F, axis = 0)
    dW = X.dot(F.T)/N + reg*W
    return loss, dW
```

Sau khi đã viết đoạn code mà chúng ta *cho rằng* đã hiệu quả (không còn vòng **for** nào) này, chúng ta cần phải kiểm chứng hai điều:

1. Quy trình 4 bước tôi nêu ở trên có chính xác không. Việc này được kiểm chứng bằng cách so sánh đạo hàm này với đạo hàm nhận được bằng cách tính *naive*. 2. Cách tính thứ hai này liệu có thực sự *hiệu quả*, tức có nhanh hơn cách *naive* nhiều không.

```
N, C, d = 49000, 10, 3073
reg = .1
W = np.random.randn(d, C)
X = np.random.randn(d, N)
y = np.random.randint(C, size = N)

import time
t1 = time.time()
l1, dW1 = svm_loss_naive(W, X, y, reg)
t2 = time.time()
print 'Naive      : run time:', t2 - t1, '(s)'

t1 = time.time()
```

```

12, dW2 = svm_loss_vectorized(W, X, y, reg)
t2 = time.time()
print 'Vectorized: run time:', t2 - t1, '(s)'
print 'loss difference:', np.linalg.norm(l1 - l2)
print 'gradient difference:', np.linalg.norm(dW1 - dW2)

```

Naive : run time: 34.326472044 (s) Vectorized: run time: 0.267823934555 (s) loss difference: 3.63797880709e-12 gradient difference: 2.70855454684e-14

Kết quả nhận được cho chúng ta thấy rằng cách tính bằng *vectorized* nhanh hơn rất nhiều (khoảng 120 lần) so với cách tính *naive*. Hơn nữa, sự chênh lệch giữa kết quả của hai cách tính là rất nhỏ, đều nhỏ hơn **1e-10** (tức 10^{-10}). Vậy thì chúng ta có thể yên tâm sử dụng cách *vectorized* này để cập nhật nghiệm.

7.3.3 Gradient Descent cho Multi-class SVM

Mọi việc giờ thật là đơn giản, giống như mọi phương pháp giải bằng Gradient Descent tôi đã nêu trước đây:

```

# Mini-batch gradient descent
def multiclass_svm_GD(X, y, Winit, reg, lr=.1, \
                      batch_size = 100, num_iters = 1000, print_every = 100):
    W = Winit
    loss_history = np.zeros((num_iters))
    for it in xrange(num_iters):
        # randomly pick a batch of X
        idx = np.random.choice(X.shape[1], batch_size)
        X_batch = X[:, idx]
        y_batch = y[idx]

        loss_history[it], dW = \
            svm_loss_vectorized(W, X_batch, y_batch, reg)

        W -= lr*dW
        if it % print_every == 1:
            print 'it %d/%d, loss = %f' \
                  %(it, num_iters, loss_history[it])

    return W, loss_history

N, C, d = 49000, 10, 3073
reg = .1
W = np.random.randn(d, C)
X = np.random.randn(d, N)
y = np.random.randint(C, size = N)

W, loss_history = multiclass_svm_GD(X, y, W, reg)

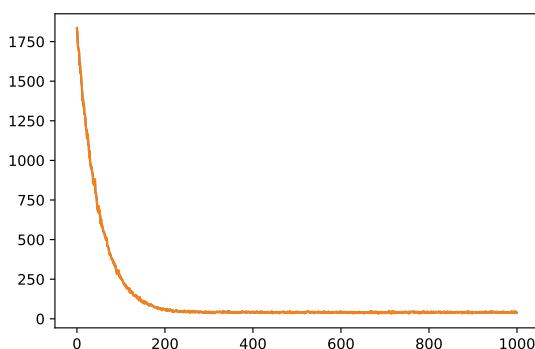
```

Kết quả:

```

it 1/1000, loss = 1802.750975
it 101/1000, loss = 251.495825
it 201/1000, loss = 62.021015

```



Hình 7.8: Lịch sử *loss* qua các vòng lặp. Ta thấy rằng *loss* có xu hướng giảm và hội tụ khá nhanh.

```
it 301/1000, loss = 45.626031
it 401/1000, loss = 38.334262
it 501/1000, loss = 43.666638
it 601/1000, loss = 45.649841
it 701/1000, loss = 35.401936
it 801/1000, loss = 36.211475
it 901/1000, loss = 41.676211
```

Chúng ta thử *visualize* giá trị của *loss* sau mỗi vòng lặp:

```
import matplotlib.pyplot as plt
# plot loss as a function of iteration
plt.plot(loss_history)
plt.show()
```

Lịch sử của hàm mât mát được cho trong Hình 7.8.

Từ *lịch sử loss* này ta thấy rằng giá trị của *loss* sau mỗi vòng lặp có xu hướng giảm và hội tụ, đây chính là điều mà chúng ta mong muốn.

Phần code còn lại để giải quyết bài toán phân loại cho cơ sở dữ liệu CIFAR-10 có thể tìm thấy trong [ipython notebook này](#).

(đây chính là lời giải của tôi cho Assignment #1 của CS231n, Winter 2016, Stanford.)

Kết quả đạt được cho CIFAR-10 là khoảng 40%. Như thế là đã rất tốt với một bài toán khó với 10 classes như thế này, nhất là khi chúng ta chưa phải làm thêm bước feature engineering phức tạp nào. Kết quả của Softmax Regression là khoảng 35%, các bạn cũng có thể tìm thấy [source code tại đây](#).

Chú ý: Trong các bài tập này, dữ liệu được tính toán theo dạng hàng, tức mỗi hàng của \mathbf{X} là một điểm dữ liệu. Khi đó, *score* được tính theo công thức: $\mathbf{Z} = \mathbf{X}\mathbf{W}$. Các phép biến đổi có khác một chút so với trường hợp dữ liệu ở dạng cột. Hy vọng các bạn không gặp khó khăn nhiều.



Hình 7.9: Minh họa hệ số tìm được dưới dạng các bức ảnh.

7.3.4 Minh họa nghiệm tìm được

Để ý rằng mỗi \mathbf{w}_i có chiều giống như chiều của dữ liệu, trong trường hợp này, chúng là các bức ảnh. Bằng cách *sắp xếp* lại các điểm trong mỗi trong 10 vector hệ số tìm được, chúng ta sẽ thu được *bức ảnh* cũng có kích thước $3 \times 32 \times 32$ như mỗi ảnh nhỏ trong cơ sở dữ liệu. Hình 7.9 mô tả hệ số tìm được của mỗi \mathbf{w}_i :

Từ đây chúng ta sẽ thấy một điều thú vị.

Hệ số tương ứng với mỗi class đều mang những tính chất giống với các bức ảnh trong class đó, ví dụ như *car* và *truck* trông khá giống với các bức ảnh trong class *car* và *truck*. Hệ số của *ship* và *plane* có mang màu xanh của nước biển và bầu trời. Trong khi *horse* trông giống như 1 con ngựa 2 đầu; điều này dễ hiểu vì trong tập training, các con ngựa có thể quay đầu về hai phía. Có thể nói theo một cách khác rằng các hệ số tìm được được coi như là các *ảnh đại diện* cho mỗi class. Vì sao chúng ta có thể nói như vậy?

Nếu chúng ta cùng xem lại cách xác định class cho một dữ liệu mới được thực hiện bằng cách tìm vị trí của giá trị lớn nhất trong *score vector* $\mathbf{W}^T \mathbf{x}$, tức:

$$\text{class}(\mathbf{x}) = \arg \max_{i=1,2,\dots,C} \mathbf{w}_i^T \mathbf{x}$$

Nếu bạn để ý chút nữa thì tích vô hướng chính là đại lượng đo sự tương quan giữa hai vector. Đại lượng này càng lớn thì sự tương quan càng cao, tức hai vector càng giống nhau. Như vậy, việc đi tìm class của một bức ảnh mới chính là việc đi tìm xem hệ số tìm được nào gần với bức ảnh đó nhất. Nói thêm một cách nữa, đây chính là *K-nearest neighbors*, nhưng thay vì thực hiện KNN trên toàn bộ training data, chúng ta chỉ thực hiện trên 10 *bức ảnh* đại diện tìm được bằng Multi-class SVM (hoặc Softmax Regression). Chính vì vậy, hai phương pháp này có thể coi là cách đi tìm mỗi điểm dữ liệu đại diện cho mỗi class!

7.4 Thảo luận

- Giống như Softmax Regression, Multi-class SVM vẫn được coi là một bộ phân lớp tuyến tính vì đường phân chia giữa các class là các đường tuyến tính.

- Kernel SVM cũng hoạt động khá tốt, nhưng việc tính toán ma trận kernel có thể tốn nhiều thời gian và bộ nhớ. Hơn nữa, việc mở rộng nó ra cho bài toán multi-class classification thường không hiệu quả bằng Multi-class SVM. Một ưu điểm nữa của Multi-class SVM là nó có thể được tối ưu bằng (Stochastic) Gradient Descent, tức là nó phù hợp với các bài toán large-scale. Việc boundary giữa các class là tuyến tính có thể được giải quyết bằng cách kết hợp nó với Deep Neural Networks. Bạn đọc có thể so sánh hiệu quả của hai phương pháp này bằng cách giải quyết bài toán CIFAR-10 bằng thư viện sklearn như tôi đã trình bày trong bài trước. Tôi đã thử, Kernel cho kết quả thấp và tốn hơn 1 giờ để huấn luyện, so với vài phút của Multi-class SVM. Có thể tôi chưa lựa chọn các tham số hợp lý, nhưng chắc chắn một điều rằng, Kernel SVM tốn nhiều thời gian huấn luyện hơn.
- Có một cách nữa mở rộng *hinge loss* cho bài toán multi-class classification là dùng *loss*: $\max(0, 1 - \mathbf{w}_{y_n}^T \mathbf{x}_n + \max_{j \neq y_n} \mathbf{w}_j^T \mathbf{x}_n)$. Đây chính là *vi phạm lớn nhất*, so với *tổng vi phạm* mà chúng ta sử dụng trong bài này.
- Trên thực tế, **Multi-class SVM và Softmax Regression có hiệu quả tương đương nhau**. Có thể trong một bài toán cụ thể, phương pháp này tốt hơn phương pháp kia, nhưng điều ngược lại xảy ra trong các bài toán khác. Khi thực hành, nếu có thể, chúng ta có thể thử cả hai phương pháp rồi chọn phương pháp có kết quả tốt hơn.

7.5 Tài liệu tham khảo

[1] CS231n Convolutional Neural Networks for Visual Recognition

[2] Hinge loss - Wikipedia

Part III

Phụ lục

Ôn tập Đại số tuyến tính

51.1 Lưu ý về ký hiệu

Trong các bài viết của tôi, các số vô hướng được biểu diễn bởi các chữ cái viết ở dạng không in đậm, có thể viết hoa, ví dụ x_1, N, y, k . Các vector được biểu diễn bằng các chữ cái thường in đậm, ví dụ \mathbf{y}, \mathbf{x}_1 . Nếu không giải thích gì thêm, các vector được mặc định hiểu là các vector cột. Các ma trận được biểu diễn bởi các chữ viết hoa in đậm, ví dụ $\mathbf{X}, \mathbf{Y}, \mathbf{W}$.

Đối với vector, $\mathbf{x} = [x_1, x_2, \dots, x_n]$ được hiểu là một vector hàng. Trong khi $\mathbf{x} = [x_1; x_2; \dots; x_n]$ được hiểu là vector cột. Chú ý sự khác nhau giữa dấu phẩy (,) và dấu chấm phẩy (;). Đây chính là ký hiệu được Matlab sử dụng.

Tương tự, trong ma trận, $\mathbf{X} = [\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_n]$ được hiểu là các vector cột \mathbf{x}_j được đặt cạnh nhau theo thứ tự từ trái qua phải để tạo ra ma trận \mathbf{X} . Trong khi $\mathbf{X} = [\mathbf{x}_1; \mathbf{x}_2; \dots; \mathbf{x}_m]$ được hiểu là các vector \mathbf{x}_i được đặt chồng lên nhau theo thứ tự từ trên xuống dưới để tạo ra ma trận \mathbf{X} . Các vector được ngầm hiểu là có kích thước phù hợp để có thể xếp cạnh hoặc xếp chồng lên nhau.

Cho một ma trận \mathbf{W} , nếu không giải thích gì thêm, chúng ta hiểu rằng \mathbf{w}_i là **vector cột** thứ i của ma trận đó. Chú ý sự tương ứng giữa ký tự viết hoa và viết thường.

51.2 Norms (chuẩn)

Trong không gian một chiều, việc đo khoảng cách giữa hai điểm đã rất quen thuộc: lấy trị tuyệt đối của hiệu giữa hai giá trị đó. Trong không gian hai chiều, tức mặt phẳng, chúng ta thường dùng khoảng cách Euclid để đo khoảng cách giữa hai điểm. Khoảng cách này chính là cái chúng ta thường nói bằng ngôn ngữ thông thường là *đường chim bay*. Đôi khi, để đi từ một điểm này tới một điểm kia, con người chúng ta không thể đi bằng đường chim bay được mà còn phụ thuộc vào việc đường đi nối giữa hai điểm có dạng như thế nào nữa.

Việc đo khoảng cách giữa hai điểm dữ liệu nhiều chiều, tức hai vector, là rất cần thiết trong Machine Learning. Chúng ta cần đánh giá xem điểm nào là điểm gần nhất của một điểm khác; chúng ta cũng cần đánh giá xem độ chính xác của việc ước lượng; và trong rất nhiều ví dụ khác nữa.

Và đó chính là lý do mà khái niệm norm (chuẩn) ra đời. Có nhiều loại norm khác nhau mà các bạn sẽ thấy ở dưới đây:

Để xác định khoảng cách giữa hai vector \mathbf{y} và \mathbf{z} , người ta thường áp dụng một hàm số lên vector hiệu $\mathbf{x} = \mathbf{y} - \mathbf{z}$. Một hàm số được dùng để đo các vector cần có một vài tính chất đặc biệt.

51.2.1 Định nghĩa

Một hàm số $f(\cdot)$ ánh xạ một điểm \mathbf{x} từ không gian n chiều sang tập số thực một chiều được gọi là norm nếu nó thỏa mãn ba điều kiện sau đây:

1. $f(\mathbf{x}) \geq 0$. Điều bằng xảy ra $\Leftrightarrow \mathbf{x} = \mathbf{0}$.
2. $f(\alpha \mathbf{x}) = \|\alpha\| f(\mathbf{x})$, $\forall \alpha \in \mathbb{R}$
3. $f(\mathbf{x}_1) + f(\mathbf{x}_2) \geq f(\mathbf{x}_1 + \mathbf{x}_2)$, $\forall \mathbf{x}_1, \mathbf{x}_2 \in \mathbb{R}^n$

Điều kiện thứ nhất là dễ hiểu vì khoảng cách không thể là một số âm. Hơn nữa, khoảng cách giữa hai điểm \mathbf{y} và \mathbf{z} bằng 0 nếu và chỉ nếu hai điểm nó trùng nhau, tức $\mathbf{x} = \mathbf{y} - \mathbf{z} = \mathbf{0}$.

Điều kiện thứ hai cũng có thể được lý giải như sau. Nếu ba điểm \mathbf{y}, \mathbf{v} và \mathbf{z} thẳng hàng, hơn nữa $\mathbf{v} - \mathbf{y} = \alpha(\mathbf{v} - \mathbf{z})$ thì khoảng cách giữa \mathbf{v} và \mathbf{y} sẽ gấp $\|\alpha\|$ lần khoảng cách giữa \mathbf{v} và \mathbf{z} .

Điều kiện thứ ba chính là bất đẳng thức tam giác nếu ta coi $\mathbf{x}_1 = \mathbf{w} - \mathbf{y}, \mathbf{x}_2 = \mathbf{z} - \mathbf{w}$ với \mathbf{w} là một điểm bất kỳ trong cùng không gian.

51.2.2 Một số chuẩn thường dùng

Giả sử các vectors $\mathbf{x} = [x_1; x_2; \dots; x_n], \mathbf{y} = [y_1; y_2; \dots; y_n]$.

Nhận thấy rằng khoảng cách Euclid chính là một norm, norm này thường được gọi là **norm 2**:

$$\|\mathbf{x}\|_2 = \sqrt{x_1^2 + x_2^2 + \dots + x_n^2} \quad (51.1)$$

Với p là một số không nhỏ hơn 1 bất kỳ, hàm số sau đây:

$$\|\mathbf{x}\|_p = (\|x_1\|^p + \|x_2\|^p + \dots \|x_n\|^p)^{\frac{1}{p}} \quad (51.2)$$

được chứng minh thỏa mãn ba điều kiện bên trên, và được gọi là **norm p**.

Nhận thấy rằng khi $p \rightarrow 0$ thì biểu thức bên trên trở thành *số các phần tử khác 0 của \mathbf{x}* . Hàm số (51.2) khi $p = 0$ được gọi là giả chuẩn (pseudo-norm) 0. Nó không phải là norm vì nó không thỏa mãn điều kiện 2 và 3 của norm. Giả-chuẩn này, thường được ký hiệu là $\|\mathbf{x}\|_0$, khá quan trọng trong Machine Learning vì trong nhiều bài toán, chúng ta cần có ràng buộc "sparse", tức số lượng thành phần "active" của \mathbf{x} là nhỏ.

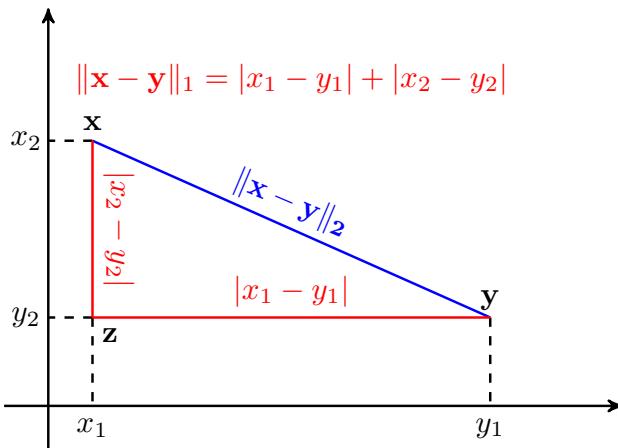
Có một vài giá trị của p thường được dùng:

1. Khi $p = 2$ chúng ta có norm 2 như ở trên.

2. Khi $p = 1$ chúng ta có:

$$\|\mathbf{x}\|_1 = \|x_1\| + \|x_2\| + \dots + \|x_n\| \quad (51.3)$$

là tổng các trị tuyệt đối của từng phần tử của \mathbf{x} . Norm 1 thường được dùng như xấp xỉ của norm 0 trong các bài toán có ràng buộc "sparse". Dưới đây là một ví dụ so sánh norm 1 và norm 2 trong không gian hai chiều:



Hình 51.1: Minh họa norm 1 và norm 2

Norm 2 (màu xanh) chính là đường thẳng "chim bay" nối giữa hai vector \mathbf{x} và \mathbf{y} . Khoảng cách norm 1 giữa hai điểm này (màu đỏ) có thể diễn giải như là đường đi từ \mathbf{x} tới \mathbf{y} trong một thành phố mà đường phố tạo thành hình bàn cờ. Chúng ta chỉ có cách đi dọc theo cạnh của bàn cờ mà không được đi thẳng.

3. Khi $p \rightarrow \infty$, ta có norm p chính là trị tuyệt đối của phần tử lớn nhất của vector đó:

$$\|\mathbf{x}\|_\infty = \max_{i=1,2,\dots,n} \|x_i\| \quad (51.4)$$

51.2.3 Chuẩn của ma trận

Với một ma trận $\mathbf{A} \in \mathbb{R}^{m \times n}$, chuẩn thường được dùng nhất là chuẩn Frobenius, ký hiệu là $\|\mathbf{A}\|_F$ là căn bậc hai của tổng bình phương tất cả các phần tử của ma trận đó.

$$\|\mathbf{A}\|_F = \sqrt{\sum_{i=1}^m \sum_{j=1}^n a_{ij}^2}$$

51.3 Đạo hàm của hàm nhiều biến

Trong mục này, chúng ta sẽ giả sử rằng các đạo hàm tồn tại. Chúng ta sẽ xét hai trường hợp: i) Hàm số nhận giá trị là ma trận (vector) và cho giá trị là một số thực vô hướng; và ii) Hàm số nhận giá trị là một số vô hướng hoặc vector và cho giá trị là một vector.

51.3.1 Hàm cho giá trị là một số vô hướng

Đạo hàm (gradient) của một hàm số $f(\mathbf{x}) : \mathbb{R}^n \rightarrow \mathbb{R}$ theo vector \mathbf{x} được định nghĩa như sau:

$$\nabla_{\mathbf{x}} f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \\ \vdots \\ \frac{\partial f(\mathbf{x})}{\partial x_n} \end{bmatrix} \in \mathbb{R}^n \quad (51.5)$$

trong đó $\frac{\partial f(\mathbf{x})}{\partial x_i}$ là đạo hàm của hàm số theo thành phần thứ i của vector \mathbf{x} . Đạo hàm này được lấy khi giả sử tất cả các biến còn lại là hằng số.

Nếu không có thêm biến nào trong hàm số, $\nabla_{\mathbf{x}} f(\mathbf{x})$ thường được viết gọn là $\nabla f(\mathbf{x})$.

Điều quan trọng cần nhớ: **đạo hàm của hàm số này là một vector có cùng chiều với vector đang lấy đạo hàm**. Tức nếu vector viết ở dạng cột thì đạo hàm cũng phải viết ở dạng cột.

Đạo hàm bậc hai (second-order gradient) của hàm số trên còn được gọi là *Hessian* và được định nghĩa như sau:

$$\nabla^2 f(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 x_n} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_2 x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_n x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_n x_2} & \cdots & \frac{\partial^2 f(\mathbf{x})}{\partial x_n^2} \end{bmatrix} \in \mathbb{S}^n \quad (51.6)$$

với $\mathbb{S}^n \in \mathbb{R}^{n \times n}$ là tập các ma trận vuông đối xứng có số cột là n .

Đạo hàm của một hàm số $f(\mathbf{X}) : \mathbb{R}^{n \times m} \rightarrow \mathbb{R}$ theo ma trận \mathbf{X} được định nghĩa là:

$$\nabla f(\mathbf{X}) = \begin{bmatrix} \frac{\partial f(\mathbf{X})}{\partial x_{11}} & \frac{\partial f(\mathbf{X})}{\partial x_{12}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{1m}} \\ \frac{\partial f(\mathbf{X})}{\partial x_{21}} & \frac{\partial f(\mathbf{X})}{\partial x_{22}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{2m}} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f(\mathbf{X})}{\partial x_{n1}} & \frac{\partial f(\mathbf{X})}{\partial x_{n2}} & \cdots & \frac{\partial f(\mathbf{X})}{\partial x_{nm}} \end{bmatrix} \in \mathbb{R}^{n \times m} \quad (51.7)$$

Một lần nữa, đạo hàm của một hàm số theo ma trận là một ma trận có chiều giống với ma trận đó.

Hiểu một cách đơn giản, đạo hàm của một hàm số (có đầu ra là 1 số vô hướng) theo một ma trận được tính như sau. Trước tiên, tính đạo hàm của hàm số đó theo từng thành phần của ma trận *khi toàn bộ các thành phần khác được giả sử là hằng số*. Tiếp theo, ta ghép các đạo hàm thành phần tính được thành một ma trận đúng theo thứ tự như trong ma trận đó. Chú ý rằng vector là một trường hợp của ma trận.

Ví dụ: Xét hàm số: $f : \mathbb{R}^2 \rightarrow \mathbb{R}$, $f(\mathbf{x}) = x_1^2 + 2x_1x_2 + \sin(x_1) + 2$.

Đạo hàm bậc nhất theo \mathbf{x} của hàm số đó là:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} \frac{\partial f(\mathbf{x})}{\partial x_1} \\ \frac{\partial f(\mathbf{x})}{\partial x_2} \end{bmatrix} = \begin{bmatrix} 2x_1 + 2x_2 + \cos(x_1) \\ 2x_1 \end{bmatrix}$$

Đạo hàm bậc hai theo \mathbf{x} , hay *Hessian* là:

$$\nabla^2 f(\mathbf{x}) = \begin{bmatrix} \frac{\partial^2 f(\mathbf{x})}{\partial x_1^2} & \frac{\partial^2 f(\mathbf{x})}{\partial x_1 x_2} \\ \frac{\partial^2 f(\mathbf{x})}{\partial x_2 x_1} & \frac{\partial^2 f(\mathbf{x})}{\partial x_2^2} \end{bmatrix} = \begin{bmatrix} 2 - \sin(x_1) & 2 \\ 2 & 0 \end{bmatrix}$$

Chú ý rằng *Hessian* luôn là một ma trận đối xứng.

51.3.2 Hàm cho giá trị là một vector

Những hàm số cho giá trị là một vector được gọi là *vector-valued function* trong tiếng Anh.

Giả sử một hàm số với **đầu vào là một số thực** $v(x) : \mathbb{R} \rightarrow \mathbb{R}^n$:

$$v(x) = \begin{bmatrix} v_1(x) \\ v_2(x) \\ \vdots \\ v_n(x) \end{bmatrix} \quad (51.8)$$

Đạo hàm của nó là một **vector hàng** như sau:

$$\nabla v(x) \triangleq \left[\frac{\partial v_1(x)}{\partial x} \frac{\partial v_2(x)}{\partial x} \dots \frac{\partial v_n(x)}{\partial x} \right] \quad (51.9)$$

Đạo hàm bậc hai của hàm số này có dạng:

$$\nabla^2 v(x) \triangleq \left[\frac{\partial^2 v_1(x)}{\partial x^2} \frac{\partial^2 v_2(x)}{\partial x^2} \dots \frac{\partial^2 v_n(x)}{\partial x^2} \right] \quad (51.10)$$

Ví dụ: Cho vector $\mathbf{a} \in \mathbb{R}^n$ và *vector-valued function* $v(x) = x\mathbf{a}$, thế thì:

$$\nabla v(x) = \mathbf{a}^T, \quad \nabla^2 v(x) = \mathbf{0} \in \mathbb{R}^{n \times n} \quad (51.11)$$

với $\mathbf{0}$ là ma trận với các thành phần đều là 0.

Xét một *vector-valued function* với **đầu vào là một vector** $h(\mathbf{x}) : \mathbb{R}^k \rightarrow \mathbb{R}^n$, đạo hàm bậc nhất của nó là:

$$\nabla h(\mathbf{x}) \triangleq \begin{bmatrix} \frac{\partial h_1(\mathbf{x})}{\partial x_1} & \frac{\partial h_2(\mathbf{x})}{\partial x_1} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_1} \\ \frac{\partial h_1(\mathbf{x})}{\partial x_2} & \frac{\partial h_2(\mathbf{x})}{\partial x_2} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_2} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial h_1(\mathbf{x})}{\partial x_k} & \frac{\partial h_2(\mathbf{x})}{\partial x_k} & \dots & \frac{\partial h_n(\mathbf{x})}{\partial x_k} \end{bmatrix} \quad (51.12)$$

$$= [\nabla h_1(\mathbf{x}) \nabla h_2(\mathbf{x}) \dots \nabla h_n(\mathbf{x})] \in \mathbb{R}^{k \times n} \quad (51.13)$$

Một quy tắc dễ nhớ ở đây là nếu một hàm số $g : \mathbb{R}^m \rightarrow \mathbb{R}^n$ thì đạo hàm của nó là một ma trận thuộc $\mathbb{R}^{m \times n}$.

Đạo hàm bậc hai của hàm số trên là một *ma trận ba chiều*, tôi xin không đề cập ở đây.

<hr> Với các hàm số *matrix-valued* nhận giá trị đầu vào là ma trận, tôi cũng xin không đề cập ở đây. Tuy nhiên, ở phần dưới, khi tính toán đạo hàm cho các hàm cho giá trị là số thực, chúng ta vẫn có thể sử dụng khái niệm này.

Trước khi đến phần tính đạo hàm của các hàm số thường gặp, chúng ta cần biết hai tính chất quan trọng khá giống với đạo hàm của hàm một biến được học trong chương trình cấp ba.

51.3.3 Hai tính chất quan trọng

Product rules

Để cho tổng quát, ta giả sử biến đầu vào là một ma trận (vector và số thực là các trường hợp đặc biệt của ma trận). Giả sử rằng các hàm số có chiều phù hợp để các phép nhân thực hiện được. Ta có:

$$\nabla (f(\mathbf{X})^T g(\mathbf{X})) = (\nabla f(\mathbf{X})) g(\mathbf{X}) + (\nabla g(\mathbf{X})) f(\mathbf{X}) \quad (51.14)$$

Biểu thức này giống như biểu thức chúng ta đã quá quen thuộc:

$$(f(x)g(x))' = f'(x)g(x) + g'(x)f(x)$$

Chú ý rằng với vector và ma trận, chúng ta không được sử dụng tính chất giao hoán.

Chain rules

Khi có các hàm hợp thì:

$$\nabla_{\mathbf{X}} g(f(\mathbf{X})) = \nabla_{\mathbf{X}} f^T \nabla_f g \quad (51.15)$$

Quy tắc này cũng giống với quy tắc trong hàm một biến:

$$(g(f(x)))' = f'(x)g'(f)$$

Nhắc lại rằng khi tính toán với ma trận, chúng ta cần chú ý tới chiều của các ma trận, và nhân ma trận không có tính chất giao hoán.

51.3.4 Đạo hàm của các hàm số thường gặp

$$f(\mathbf{x}) = \mathbf{a}^T \mathbf{x}$$

Giả sử $\mathbf{a}, \mathbf{x} \in \mathbb{R}^n$, ta viết lại:

$$f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} = a_1 x_1 + a_2 x_2 + \cdots + a_n x_n$$

Có thể nhận thấy rằng:

$$\frac{\partial f(\mathbf{x})}{\partial x_i} = a_i, \quad \forall i = 1, 2, \dots, n$$

Vậy nên:

$$\nabla f(\mathbf{x}) = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{bmatrix} = \mathbf{a} \quad (51.16)$$

Thêm nữa, vì $\mathbf{a}^T \mathbf{x} = \mathbf{x}^T \mathbf{a}$ nên:

$$\nabla(\mathbf{x}^T \mathbf{a}) = \mathbf{a}$$

$$f(\mathbf{x}) = \mathbf{A}\mathbf{x}$$

Đây là một *vector-valued function* $f : \mathbb{R}^n \rightarrow \mathbb{R}^m$ với $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{m \times n}$. Giả sử rằng \mathbf{a}_i là hàng thứ i của ma trận \mathbf{A} . Ta có:

$$\mathbf{A}\mathbf{x} = \begin{bmatrix} \mathbf{a}_1\mathbf{x} \\ \mathbf{a}_2\mathbf{x} \\ \vdots \\ \mathbf{a}_m\mathbf{x} \end{bmatrix}$$

Theo định nghĩa (51.13), và công thức (51.16), ta có thể suy ra:

$$\nabla_{\mathbf{x}}(\mathbf{A}\mathbf{x}) = [\mathbf{a}_1^T \ \mathbf{a}_2^T \ \dots \ \mathbf{a}_m^T] = \mathbf{A}^T \quad (51.17)$$

Từ đây ta có thể suy ra đạo hàm của hàm số $f(\mathbf{x}) = \mathbf{x} = \mathbf{I}\mathbf{x}$, với \mathbf{I} là ma trận đơn vị với chiều phù hợp, là:

$$\nabla_{\mathbf{x}} = \mathbf{I}$$

$$f(\mathbf{x}) = \mathbf{x}^T \mathbf{A}\mathbf{x}$$

với $\mathbf{x} \in \mathbb{R}^n$, $\mathbf{A} \in \mathbb{R}^{n \times n}$. Áp dụng Product rules (14) ta có:

$$\begin{aligned} \nabla f(\mathbf{x}) &= \nabla((\mathbf{x}^T)(\mathbf{A}\mathbf{x})) \\ &= (\nabla(\mathbf{x}))\mathbf{A}\mathbf{x} + (\nabla(\mathbf{A}\mathbf{x}))\mathbf{x} \\ &= \mathbf{I}\mathbf{A}\mathbf{x} + \mathbf{A}^T\mathbf{x} \\ &= (\mathbf{A} + \mathbf{A}^T)\mathbf{x} \end{aligned} \quad (51.18)$$

Từ (51.18) và (51.17), ta có thể suy ra: $\nabla^2 \mathbf{x}^T \mathbf{A}\mathbf{x} = \mathbf{A}^T + \mathbf{A}$

Nếu \mathbf{A} là một ma trận đối xứng, ta sẽ có:

$$\nabla \mathbf{x}^T \mathbf{A}\mathbf{x} = 2\mathbf{A}\mathbf{x}, \quad \nabla^2 \mathbf{x}^T \mathbf{A}\mathbf{x} = 2\mathbf{A} \quad (51.19)$$

Nếu \mathbf{A} là ma trận đơn vị, tức $f(\mathbf{x}) = \mathbf{x}^T \mathbf{I}\mathbf{x} = \mathbf{x}^T \mathbf{x} = \|\mathbf{x}\|_2^2$, ta có:

$$\nabla \|\mathbf{x}\|_2^2 = 2\mathbf{x}, \quad \nabla^2 \|\mathbf{x}\|_2^2 = 2\mathbf{I} \quad (51.20)$$

$$f(\mathbf{x}) = \|\mathbf{Ax} - \mathbf{b}\|_2^2$$

Có hai cách tính đạo hàm của hàm số này:

Cách 1: Trước hết, biến đổi:

$$\begin{aligned} f(\mathbf{x}) &= \|\mathbf{Ax} - \mathbf{b}\|_2^2 = (\mathbf{Ax} - \mathbf{b})^T(\mathbf{Ax} - \mathbf{b}) = (\mathbf{x}^T \mathbf{A}^T - \mathbf{b}^T)(\mathbf{Ax} - \mathbf{b}) \\ &= \mathbf{x}^T \mathbf{A}^T \mathbf{Ax} - 2\mathbf{b}^T \mathbf{Ax} + \mathbf{b}^T \mathbf{b} \end{aligned}$$

Lấy đạo hàm cho từng số hạng rồi cộng lại ta có:

$$\nabla \|\mathbf{Ax} - \mathbf{b}\|_2^2 = 2\mathbf{A}^T \mathbf{Ax} - 2\mathbf{A}^T \mathbf{b} = 2\mathbf{A}^T(\mathbf{Ax} - \mathbf{b})$$

Cách 2: Dùng Chain rule. Sử dụng $\nabla(\mathbf{Ax} - \mathbf{b}) = \mathbf{A}^T$ và $\nabla \|\mathbf{x}\|_2^2 = 2\mathbf{x}$ và công thức chain rules (51.15), ta sẽ thu được kết quả tương tự.

$$f(\mathbf{x}) = \mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}$$

Bằng cách viết lại $f(\mathbf{x}) = (\mathbf{a}^T \mathbf{x})(\mathbf{x}^T \mathbf{b})$, ta có thể dùng Product rules (14) và ra kết quả:

$$\nabla(\mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}) = \mathbf{a} \mathbf{x}^T \mathbf{b} + \mathbf{b} \mathbf{a}^T \mathbf{x} = \mathbf{a} \mathbf{b}^T \mathbf{x} + \mathbf{b} \mathbf{a}^T \mathbf{x} = (\mathbf{a} \mathbf{b}^T + \mathbf{b} \mathbf{a}^T) \mathbf{x}$$

trong đây tôi đã sử dụng tính chất $\mathbf{y}^T \mathbf{z} = \mathbf{z}^T \mathbf{y}$ và tích của một số thực với một vector cũng bằng tích của vector và số thực đó.

51.3.5 Bảng các đạo hàm thường gặp

Cho vector

$f(\mathbf{x})$	$\nabla f(\mathbf{x})$
$\mathbf{a}^T \mathbf{x}$	\mathbf{a}
$\mathbf{x}^T \mathbf{A} \mathbf{x}$	$(\mathbf{A} + \mathbf{A}^T) \mathbf{x}$
$\mathbf{x}^T \mathbf{x} = \ \mathbf{x}\ _2^2$	$2\mathbf{x}$
$\ \mathbf{Ax} - \mathbf{b}\ _2^2$	$2\mathbf{A}^T(\mathbf{Ax} - \mathbf{b})$
$\mathbf{a}^T \mathbf{x}^T \mathbf{x} \mathbf{b}$	$2\mathbf{a}^T \mathbf{b} \mathbf{x}$
$\mathbf{a}^T \mathbf{x} \mathbf{x}^T \mathbf{b}$	$(\mathbf{a} \mathbf{b}^T + \mathbf{b} \mathbf{a}^T) \mathbf{x}$

Bảng 51.1: Đạo hàm theo vector

$f(\mathbf{X})$	$\nabla f(\mathbf{X})$
$\ \mathbf{X}\ _F^2$	$2\mathbf{X}$
\mathbf{AX}	\mathbf{A}^T
$\ \mathbf{AX} - \mathbf{B}\ _F^2$	$2\mathbf{A}^T(\mathbf{AX} - \mathbf{B})$
$\ \mathbf{XA} - \mathbf{B}\ _F^2$	$2(\mathbf{XA} - \mathbf{B})\mathbf{A}^T$
$\mathbf{a}^T \mathbf{X}^T \mathbf{X} \mathbf{b}$	$\mathbf{X}(\mathbf{ab}^T + \mathbf{ba}^T)$
$\mathbf{a}^T \mathbf{X} \mathbf{X}^T \mathbf{b}$	$(\mathbf{ab}^T + \mathbf{ba}^T)\mathbf{X},$
$\mathbf{a}^T \mathbf{Y} \mathbf{X}^T \mathbf{b}$	$\mathbf{ba}^T \mathbf{Y}$
$\mathbf{a}^T \mathbf{Y}^T \mathbf{X} \mathbf{b}$	\mathbf{Yab}^T
$\mathbf{a}^T \mathbf{X} \mathbf{Y}^T \mathbf{b}$	$\mathbf{ab}^T \mathbf{Y}$
$\mathbf{a}^T \mathbf{X}^T \mathbf{Y} \mathbf{b}$	\mathbf{Yba}^T

Bảng 51.2: Đạo hàm theo ma trận

Cho ma trận

(Xem Bảng 51.2)

51.3.6 Tài liệu tham khảo

[1] Matrix calculus

Index

- α -sublevel sets, 18
- affine functions, 15
- bias trick, 107
- complementary slackness, 54
- constraints, 5
- contours, 16
- convex, 4
 - combination, 11
 - functions, 12
 - first-order condition, 20
 - Second-order condition, 22
 - hull, 11
 - optimization problems, 31
 - sets, 5
 - strictly convex functions, 13
- CVXOPT, 34
- duality, 45
- ellipsoids, 9
- feasible points, 5
- feasible sets, 5
- Geometric Programming, 40
 - convex form, 42
- halfspace, 8
- hinge loss, 81
- hinge loss tổng , 109
- hyperplane, 7
- infeasible sets, 5
- Kernel, 95
 - Kernel trick, 95
 - Linear, 97
- Mercer conditions, 96
- Polynomial, 97
- Radial Basic Function (RBF), 97
- Sigmoid, 98
- KKT conditions, 55
- Lagrange/Lagrangian
 - auxiliary function, 46
 - dual function, 48
 - dual functions, 48
 - dual problems, 51
 - multiplier method, 46
- level sets, 16
- Linear Programming, 34
 - General form, 34
 - Standard form, 35
- Mahalanobis norm, 9
- monomials, 40
- norm balls, 8
- posynomials, 40
- quadratic
 - forms, 15
- Quadratic Programming, 37
- quasiconvex, 20
- Separating hyperplane theorem, 12
- Slater's constraint qualification, 53
- strong duality, 53
- Support Vector Machine, 60
 - Hard Margin SVM, 60
 - Kernel SVM, 92
 - Multi-class SVM, 103
 - Soft Margin SVM, 73
- weak duality, 52